UNIVERSIDADE FEDERAL DE MINAS GERAIS

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Algoritmos e Estruturas de Dados III
Trabalho Prático 2 – Índice Invertido

Caio Felipe Zanatelli

Belo Horizonte
14 de junho de 2017

1 Introdução

Recuperação de Informação (RI) é um campo de grande relevância no âmbito de Ciência da Computação, pois permite o armazenamento de dados e recuperação de informação a eles associados de forma automática, possuindo aplicações em diversas áreas. Neste contexto, este trabalho apresenta a implementação de um Índice Invertido, o qual é fundamental para o tratamento de grandes volumes de informação por uma máquina de busca.

Um índice invertido é uma estrutura de índice reseponsável por armazenar registros que mapeiam um conteúdo em documentos de uma base de dados. Nesta estrutura, aqui implementada para a busca em uma coleção de documentos de conteúdo textual, cada ocorrência de uma palavra em um dado documento é listado no índice invertido na forma $< w, d, f, p >$, onde w é a palavra, d é o documento em que a palavra ocorre, f é a frequência da palavra naquele documento e p é a posição, em *bytes*, da palavra no documento. Além disso, tal estrutura de índice é ordenada lexicograficamente para as palavras, como primeiro critério, e os demais termos ordenados como valores inteiros não-negativos.

Por fim, o ponto fundamental deste trabalho está relacionado ao fato de que o índice invertido pode ser grande o suficiente para não caber em memória principal. Assim, a ordenação dos registros foi efetuada através de um algoritmo de ordenação externa, mais especificamente o *Quicksort Externo*. Dessa forma, é importante considerar ainda que esta implementação utiliza uma quantidade limitada de memória principal, sendo essa limitação ajustada pelo usuário.

2 Solução do Problema

Esta seção tem como objetivo apresentar a solução implementada neste trabalho. Para tanto, inicialmente será introduzida a modelagem proposta, ilustrando as etapas da resolução do problema. Em seguida será apresentado o algoritmo *Quicksort Externo*, utilizado para a ordenação do índice invertido. Por fim, a estrutura de código utilizada é apresentada.

2.1 Modelagem do Problema

A criação do índice invertido foi dividida em três etapas principais. Porém, antes de introduzí-las, é necessário ressaltar que o índice invertido possui registros da forma $< w, d, f, p >$, como abordado anteriormente. Como o tamanho dos registros é fixo, possuindo exatamente 32 *bytes*, distribuídos em 20 *bytes* para a palavra e 12 *bytes* para os inteiros d, f e p, todo o processo de análise e manipulação dos arquivos foi efetuada utilizando arquivos binários. Esta questão de projeto foi adotada devido ao fato de arquivos binários armazenarem registros na forma de blocos de *bytes* de tamanho fixo e em forma sequencial, permitindo o fácil acesso a qualquer registro do arquivo. A Figura 1 ilustra a disposição de registros com os campos $< w, d, f, p >$ em um arquivo binário, e em seguida são apresentadas as etapas do algoritmo.

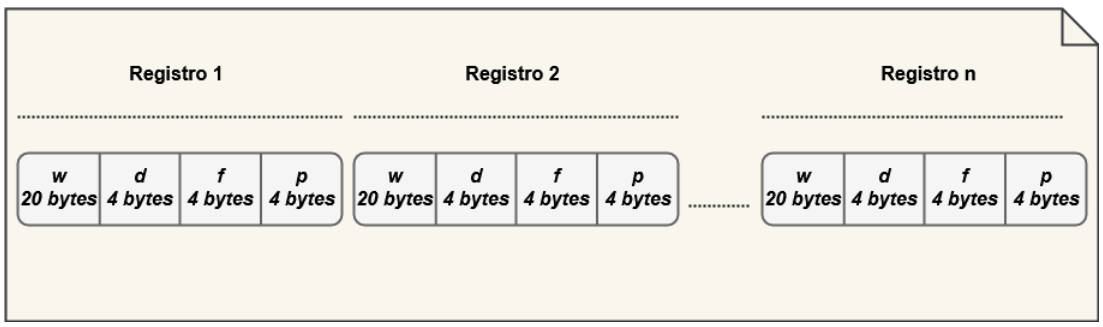

Figura 1 – Distribuição dos registros em um arquivo binário

.

A primeira etapa consiste na criação de um único arquivo, o qual conterá as ocorrências dos registros de todos os documentos da coleção. Esses registros são gravados no formato $< w, d, f, p >$ em um arquivo binário. Neste ponto, a palavra, o documento e a posição referentes aos registros são conhecidas, pois, uma vez que a palavra foi lida de um arquivo, essa identificação é trivial, bastando verificar o arquivo de leitura e a posição do ponteiro de leitura. O campo de frequência, por sua vez, é desconhecido, pois não se sabe, de início, quantas vezes determinada palavra w ocorreu no arquivo d. Sabe-se, porém, que ela apareceu pelo menos uma vez, pois a leitura foi efetuada. Assim, na primeira etapa a frequência é fixada com valor 1 e gravada no arquivo inicial, deixando a atualização deste campo para as etapas posteriores.

Com o arquivo inicial gerado, a segunda etapa consiste em ordená-lo em memória secundária. O critério de ordenação definido é primeiro a palavra, em ordem lexicográfica, e em seguida os inteiros que representam o número do documento, a frequência da palavra e a posição da palavra naquele documento, respectivamente. Entretanto, é importante observar que a frequência da palavra é indiferente na definição da ordem dos registros. Para visualizar isso, sejam $R_1 = \{w_1, d_1, f_1, p_1\}$ e $R_2 = \{w_2, d_2, f_2, p_2\}$ dois registros distintos. Se $w_1 < w_2$, então R_1 vem antes de R_2. Se $w_1 > w_2$, então R_2 vem antes de R_1 na ordenação. Porém, se $w_1 = w_2$, então o segundo critério a ser a analisado é o documento. Na mesma lógica, se $d_1 < d_2$, então R_1 vem antes de R_2, e se $d_1 > d_2$, R_2 vem antes de R_1. Agora, se $d_1 = d_2$, então obviamente $w_1 = w_2$, ou seja, a mesma palavra w ocorre em um mesmo documento d, tal que a frequência de ambas é a mesma, isto é, $f_1 = f_2$. Mas como são duas ocorrências diferentes, é fácil observar que as posições, em *bytes*, dessas palavras no documento d são distintas, o que leva à afirmação anterior de que apenas as variáveis w, d e p são relevantes para a ordenação. Dito isso, o arquivo gerado na etapa anterior é ordenado através do algoritmo *Quicksort Externo*, o qual foi escolhido por não precisar da utilização de fitas (arquivos) adicionais, como é o caso do método de intercalação, efetuando toda a ordenação *in-place*. A implementação utilizando tal algoritmo se mostrou bastante eficiente para a ordenação dos documentos, como será apresentado posteriormente na análise experimental.

Por fim, com o arquivo inicial já ordenado pela etapa anterior, a contagem da frequência é efetuada e o índice invertido final é gerado. Para tanto, o arquivo binário com os registros devidamente ordenados é percorrido e a contagem é analisada da seguinte forma. Em primeiro lugar, definimos que um registro $R_a = \{w_a, d_a, f_a, p_a\}$ é ocorrência de um registro $R_b = \{w_b, d_b, f_b, p_b\}$ se $w_a = w_b$ e $d_a = d_b$. Asim, seja R_i um registro lido no momento i, e seja P_i a posição desse registro no arquivo ordenado. Seja $R_j \neq R_i$ um novo registro e P_j sua posição no arquivo, tal que $R_{i+1}, R_{i+2}, ..., R_{j-1}$ sejam ocorrências de R_i. Os registros compreendidos entre R_i e R_j são contabilizados, compondo a frequência de w_i no documento d_i. Uma vez identificado R_j, e com as posições P_i e P_j guardadas em memória, o ponteiro de leitura do arquivo é movido para P_i e os registros até P_{j-1}, inclusive, são lidos novamente, alterando suas frequências para a nova frequência calculada. Entretanto, essa alteração não é realizada no arquivo binário, e sim armazenada em um arquivo texto chamado *index*, o qual é o índice invertido final requisitado. Em outras palavras, a cada novo registro identificado, sua frequência é calculada através da leitura dos registros posteriores até que um registro que não seja ocorrência deste seja encontrado, e as frequências dos registros lidos são alterados e esses registros são então armazenados no arquivo final. Este processo se repete até que a frequência de todos os registros sejam alteradas e o índice invertido final gerado. Vale ressaltar, neste ponto, que, como a posição da primeira ocorrência de um registro, bem como a posição de sua última ocorrência são armazenados, este processo de contagem de frequência é feito em tempo linear, pois com o arquivo binário é possível iniciar a leitura exatamente na primeira ocorrência do registro tratado no momento, levando a exatamente duas passadas no arquivo ordenado para a geração do arquivo final.

Com as etapas supracitadas, o índice invertido é gerado, resolvendo o problema proposto. Assim, de forma a possibilitar uma melhor visualização acerca dessas etapas, a Figura 2 ilustra todo este processo.

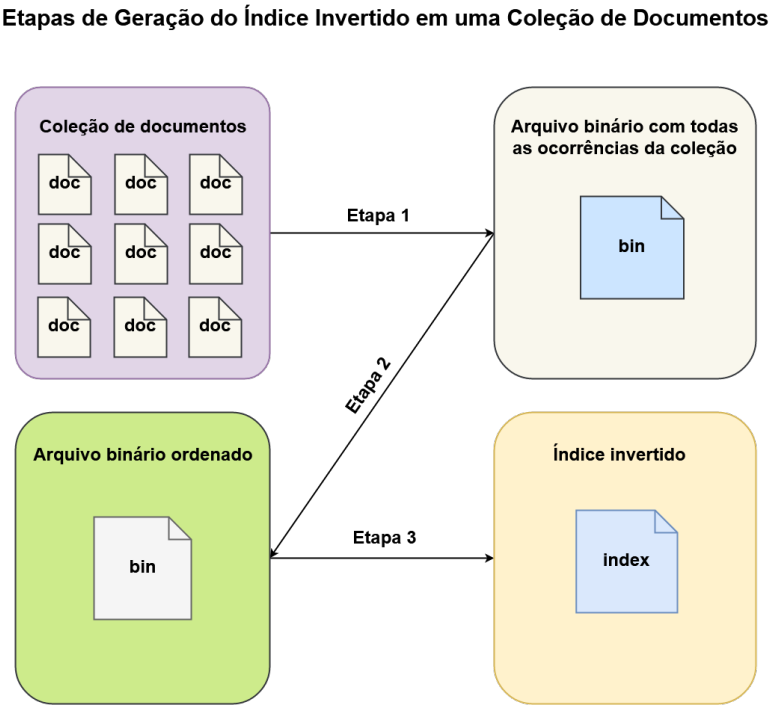

Figura 2 – Etapas de geração do índice invertido

.

2.2 Ordenação Externa

Como já abordado, o algoritmo de ordenação externa utilizado neste trabalho é o *QuickSort Externo*. Essa escolha foi feita baseado no fato de que o algoritmo em questão não necessita de arquivos auxiliares para executar a ordenação, ou seja, se trata de uma ordenação *in-place*, o que diminui o tempo gasto em termos de acesso ao disco. A seguir serão descritas as etapas do algoritmo.

O funcionamento do *Quicksort Externo* é parecido com o do *Quicksort* interno, sendo o pivô não mais um elemento, e sim um *buffer* com uma determinada quantidade de registros. Assim,são necessários quatro ponteiros: leitura inferior, leitura superior, escrita inferior e escrita superior. Para melhor entendimento, seja M a quantidade de registros que cabem no *buffer*. O ponteiro de leitura inferior é inicializado no primeiro registro do arquivo, e o ponteiro de leitura superior é inicializado no último registro. A primeira etapa do algoritmo é ler M registros do arquivo, alternando entre os ponteiros de leitura inferior e leitura superior. Com este processo, serão lidos $\frac{M}{2}$ registros em ambos os lados. Em seguida, esse *buffer* é ordenado por algum algoritmo de ordenação interna, no contexto deste trabalho, o *Quicksort* interno.

A próxima etapa é efetuar uma leitura balanceada da parte inferior e da parte superior do arquivo, através da alternância entre os ponteiros de leitura. Com o registro lido, se este for menor do que o menor registro do *buffer*, ele é gravado na posição apontada pelo ponteior de escrita inferior. Se o registro em questão for maior do que o maior registro do *buffer*, ele é gravado na posição apontada pelo ponteior de escrita superior. Senão, o registro está entre o mínimo e o máximo elemento do *buffer*. Neste caso, o menor ou o maior registro do *buffer* deve ser escrito no arquivo, e o registro analisado (que estava entre o mínimo e o máximo) deve ser trazido para o *buffer*. Entretanto, essa escolha deve ser balanceada, para não utilizar mais um ponteiro de escrita do que o outro, ou seja, se o ponteiro de escrita inferior foi mais utilizado até o momento, essa gravação é realizada pelo ponteior de escrita superior, e vice-versa. Após isso, o *buffer* deve ser novamente ordenado, uma vez que um de seus registros foi substituído. Neste ponto, como o vetor

está semi-ordenado, foi utilizado o algoritmo *Insertion Sort*, pois, para esse caso, a ordenação é feita em tempo linear.

Após as etapas anteriores, o *buffer* é descarregado no arquivo utilizando o ponteiro de escrita da esquerda. Já estando o *buffer* ordenado, quando este for gravado no arquivo, também será em ordem. Uma vez que isso foi feito, essa porção do arquivo já está ordenada e não será avaliada novamente. Em seguida, recursivamente a partição à esquerda e à direita do *buffer* também é ordenada. No fim deste processo, o arquivo todo está ordenado, sem a necessidade, em momento algum, de arquivos auxiliares para tal, como ocorre no processo de ordenação por intercalação, por exemplo. Por fim, a Figura 3 ilustra um exemplo de ordenação, onde o tamanho do *buffer* é $M = 3$ registros e o arquivo a ser ordenado possui quatro registros. Para melhor visualização, os ponteiros de leitura inferior e superior foram chamados de li e ls, respectivamente, e os ponteiros de escrita inferior e superior foram chamados de ei e es, respectivamente.

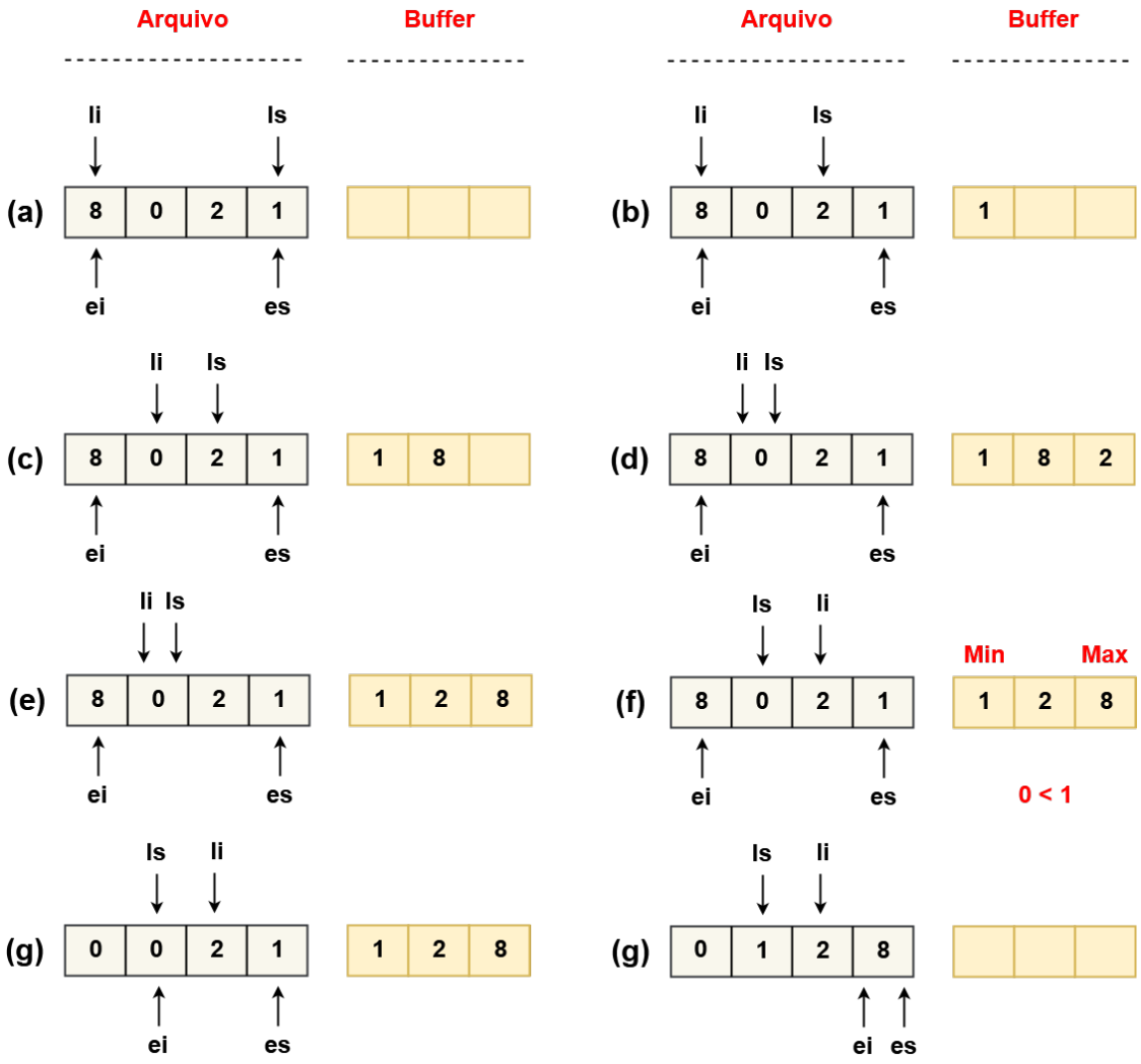

Figura 3 – Exemplo de ordenação através do *QuickSort Externo*

.

2.3 Organização do Código

Definido o problema e a modelagem para a sua resolução, esta seção tem por objetivo apresentar a estrutura de código dos módulos que compõem a implementação deste trabalho.

2.3.1 TAD Register

Como discutido anteriormente na modelagem do problema, arquivos binários foram manipulados para analisar e processar os registros. Assim, foi criado um tipo abstrato de dados (TAD) *Register*, o qual contém os seguintes campos:

- **word**: *string* que contém a palavra do registro.
- **document**: inteiro responsável por armazenar o documento que que a palavra *word* ocorre.
- **frequency**: inteiro que identifica a frequência em que a palavra *word* ocorre no documento *document*.
- **position**: inteiro responsável por armazenar a posição, em *bytes*, da palavra *word* no documento *document*.

Além disso, duas funcionalidades também foram implementadas, a qual são introduzidas na sequência.

- **newRegister**: inicializa os campos de um registro.
- **compareRegisters**: função implementada para comparar dois registros, a qual é utilizada nas funções de ordenação. A função executada com os parâmetros $(regA, regB)$ retorna um valor < 0, se $regA < regB$; 0, se $regA = regB$; e um valor > 0, se $regA > regB$. Essas comparações são efetuadas com primeiro critério a ordem lexicográfica das palavras e depois o documento, a frequência e a posição das palavras.

2.3.2 Biblioteca InvertedIndex.h

Nesta biblioteca estão implementadas as funções responsáveis pela criação do arquivo invertido, as quais são brevemente introduzidas a seguir.

- **createInvertedIndex**: função principal invocada pelo usuário, responsável por efetuar as três etapas descritas na modelagem do problema. Primeiro, a função *mergeInitialFiles* é chamada, a qual irá criar um arquivo binário contendo todas as ocorrências da coleção de documentos. Em seguida, a função *externalSort*, da biblioteca *ExternalSort.h*, é chamada, a qual irá ordenar o arquivo gerado na etapa anterior. Por último, a função *createFrequencieFile* é chamada, criando o índice invertido com as frequências devidamente contabilizadas.
- **mergeInitialFiles**: responsável por criar um único arquivo binário contendo as ocorências de todas as palavras dos documentos da coleção. Essas ocorrências são armazenadas através do tipo *Register*, o qual contém os campos $< w, d, f, p >$, discutido anteriormente.
- **createFrequenciesFile**: função responsável por, através do arquivo ordenado, contabilizar a frequência de cada registro e gerar o índice invertido final, criando o arquivo de saída *index*.

2.3.3 Biblioteca InternalSort.h

Esta biblioteca implementa os métodos de ordenação interna utilizados pelo *QuickSort Externo*, são eles:

- **insertionSort**: ordena um vetor de elementos do tipo *Register* através do método de ordenação interna por inserção.
- **internalQuickSort**: ordena um vetor de elementos do tipo *Register* através do método de

ordenação interna *QuickSort*.

2.3.4 Biblioteca ExternalSort.h

Esta biblioteca implementa os métodos responsáveis pela ordenação externa através do *QuickSort Externo*. As funções presentes neste módulo utilizam o tipo de dados *Buffer*, um tipo abstrato de dados criado para a manipulação da área de memória disponível para a ordenação externa, uma vez que este fator é ajustado pelo usuário. Assim, o tipo *Buffer* possui os seguintes campos:

- **data**: vetor do tipo *Register*, alocado dinamicamente. Contém os registros do arquivo para efetuar a ordenação interna e transferí-los de volta para o arquivo.
- **capacity**: inteiro que armazena a capacidade máxima de memória principal que pode ser utilizada. Esta é a capacidade do vetor *data*, e seu valor é $\frac{M}{32}$, ou seja, a quantidade M de *bytes* informada pelo usuário dividido por 32, uma vez que cada registro possui 32 *bytes*.
- **size**: inteiro que armazena o número de registros presentes na memória principal, isto é, a quantidade de elementos do vetor *data* que foram utilizados até o momento.

Por fim, as funções implementadas nessa biblioteca são:

- **initBuffer**: inicializa uma variável do tipo *Buffer*, atribuindo os devidos valores aos campos *size* e *capacity*. Também aloca o vetor *data* com um número de elementos definido pelo campo *capacity*.
- **freeBuffer**: libera a memória alocada para a estrutura do tipo *Buffer*.
- **externalSort**: abre os arquivos para ajustar os quatro ponteiros utilizados pela ordenação e aloca a memória para armazenar os registros em memória principal, através de uma variável do tipo *Buffer*. Em seguida chama o método *externalQuickSort*, o qual irá ordenar o arquivo utilizando o algoritmo *QuickSort Externo*. Por último, desaloca a memória alocada para o *buffer*, através da função *freeBuffer*, e então fecha os arquivos.
- **externalQuickSort**: ordena a partição atual e faz chamadas recursivas para ordenar as partições da direita e da esquerda.
- **externalPartition**: implementa a partição para o funcionamento do *QuickSort Externo*. Preenche o *buffer* com registros do arquivo, alternando a leitura entre os ponteiros de leitura superior e inferior, e ordena o *buffer* utilizando o algoritmo *QuickSort* interno, da biblioteca *InternalSort.h*, apresentada anteriormente. Responsável também por mover registros do arquivo para o *buffer* e do *buffer* para o arquivo, de forma a possibilitar a ordenação do mesmo. Cada vez que um registro é substituído, o *buffer* é novamente ordenado através do algoritmo *Insertion Sort*, também implementado pela biblioteca *InternalSort.h*.
- **switchRegistersBuffer**: se um dado registro for maior do que o menor registro e menor do que o maior registro do *buffer*, o registro do *buffer* é escrito no arquivo e então o registro do *buffer* é substituído por aquele utilizado na comparação. Isso é feito variando o ponteiro de escrita superior e inferior, de forma a equilibrar a escrita.
- **readTop**: lê o próximo registro do arquivo, utilizando o ponteiro de leitura superior.
- **readBottom**: lê o próximo registro do arquivo, utilizando o ponteiro de leitura inferior.
- **writeMax**: se um dado registro for maior do que o maior registro do *buffer*, esse registro é gravado no arquivo, utilizando o ponteiro de escrita superior.
- **writeMin**: se um dado registro for menor do que o menor registro do *buffer*, esse registro é gravado no arquivo, utilizando o ponteiro de escrita inferior.
- **fillBuffer**: preenche o *buffer* com registros lidos do arquivo, alternando a leitura entre o ponteiro de leitura superior e inferior.

3 Análise de Complexidade

Nesta seção serão apresentadas as análises de complexidade de tempo e espaço do algoritmo implementado.

3.1 Complexidade Temporal

Como se trata de uma implementação cuja essência é trabalhar algoritmos e acessos em memória secundária, na teoria o tempo gasto pelos algoritmos executados em memória principal é considerado constante. Isso decorre do fato de que acessos à memória secundária está a ordens de grandeza, em termos de ciclos de *clock* necessários para a execução das instruções, acima daqueles à memória primária, sendo esta diferença cerca de 10^4. Com base nisso, todas as funções que demandam apenas processamento em memória principal, como por exemplo o *QuickSort* interno e o *Insertion Sort*, são aqui avaliadas com complexidade assintótica de tempo $O(1)$. Vale ressaltar, ainda, que para as análises feitas nesta seção, são considerados n o número total de registros presentes na coleção de documentos, m a memória máxima, em *bytes*, que o algoritmo pode utilizar, sendo esta variável informada pelo usuário, e b o tamanho de cada registro, neste caso, 32 *bytes*.

A primeira biblioteca a ser avaliada é a *InvertedIndex.h*. Nela estão presentes três funções: *createInvertedIndex*, *mergeInitialFiles* e *createFrequenciesFile*. A primeira delas basicamente invoca as funções *mergeInitialFiles* e *createFrequenciesFile*, bem como a função que executa o *QuickSort Externo* – *externalSort*, a qual será avaliada posteriormente. A função *mergeInitalFiles* é responsável por efetuar a leitura dos documentos da coleção e gerar um único arquivo binário com todo o conteúdo lido. Sendo n o número de registros, o custo desta função é $O(n)$. A função *createFrequenciesFile*, por sua vez, é responsável por gerar o arquivo final que contém o índice invertido. Essa geração é feita baseada na contagem de frequências dos registros salvos em um arquivo binário. Como, quando encontrada a frequência f de um registro, são lidos novamente exatamente f registros e então gravados em um arquivo texto, tal que um registro nunca é avaliado mais de uma vez, através de uma análise amortizada conclui-se que este processo precisa de $2n$ acessos à memória secundária, o que caracteriza uma complexidade temporal de $O(n)$.

Em relação à biblioteca *ExternalSort.h*, as funções *initBuffer* e *freeBuffer* levam tempo constante para ser executadas, pois apenas alocam e desalocam o *buffer* responsável por armazenar os registros em memória principal, respectivamente. As funções *readTop*, *readBottom*, *writeTop* e *writeBottom* são executadas em $O(1)$, uma vez que necessitam apenas acessar uma posição do arquivo para efetuar a leitura ou escrita de um registro.

Por fim, a função *externalSort*, que executa o *QuickSort Externo* para a ordenação de um arquivo binário com n registros, utiliza o paradigma de Divisão e Conquista, gerando partições cada vez menores e ordenando-as, sem retornar a uma partição já avaliada. A cada chamada recursiva, m elementos de b *bytes* são trazidos para a memória, sendo m o número de registros que cabem em memória principal. São realizadas, então, $log\frac{n}{m}$ chamadas recursivas, o que, no caso médio, resulta em uma complexidade equivalente a $O(\frac{n}{b} \cdot log\frac{n}{m})$. Já no melhor caso, quando o vetor já está ordenado, não são criadas novas partições, pois os registros do *buffer* são sempre substituídos, resultando em uma complexidade de $O(\frac{n}{b})$. Por fim, no pior caso, o qual ocorre quando o particionamento é realizado de tal modo que uma das partições fique vazia e a outra com todos os registros, o que caracteriza uma árvore de recursão totalmente degenerada. Neste caso, a complexidade do *QuickSort Externo* é $O(\frac{n^2}{m})$.

Tendo em vista as complexidades supracitadas, tem-se que a complexidade temporal total desta implementação é aquela de maior ordem, ou seja, a do *QuickSort Externo*. Dessa forma, em suma, o melhor caso é $O(\frac{n}{b})$, o caso médio é $O(\frac{n}{b} \cdot log\frac{n}{m})$ e o pior caso é $O(\frac{n^2}{m})$.

3.2 Complexidade Espacial

Como se trata de um algoritmo essencialmente de ordenação externa, com capacidade de memória principal disponível limitada, o cerne da análise de complexidade espacial está justamente no espaço utilizado pelo método de ordenação, pois é o único que utiliza alocação dinâmica. Assim, sendo m o número máximo de *bytes* que podem ser utilizados em memória principal, a complexidade espacial do algoritmo implementado neste trabalho é $O(m)$.

4 Avaliação Experimental

Para analisar o comportamento do algoritmo na prática, foram realizados dois experimentos. Para ambos, foi criado um gerador de texto randômico, no qual são configurados o número de *bytes* total que este deverá ter, sendo este parâmetro que definirá a quantidade de palavras. Cada linha do arquivo possui um número aleatório de palavras, variando de uma a trinta palavras. Cada palavra, por sua vez, possui um número aleatório de caracteres, o qual varia de um a vinte. Para cada teste, o número de documentos se manteve constante, uma vez que este fator levaria mais tempo apenas para a leitura do arquivo. Sendo o algoritmo de ordenação externa bem mais custoso do que uma simples leitura, esta estratégia é válida. O primeiro experimento realizado foi fixar o número de memória que o programa pode utilizar e então variar o tamanho das conversas. O resultado obtido (Figura 4) mostra o que já era esperado: como o número de palavras aumenta de forma randômica, gerando casos médios, o gráfico *tamanho das conversas x tempo de processamento* é próximo de $\frac{n}{b} \cdot log\frac{n}{m}$.

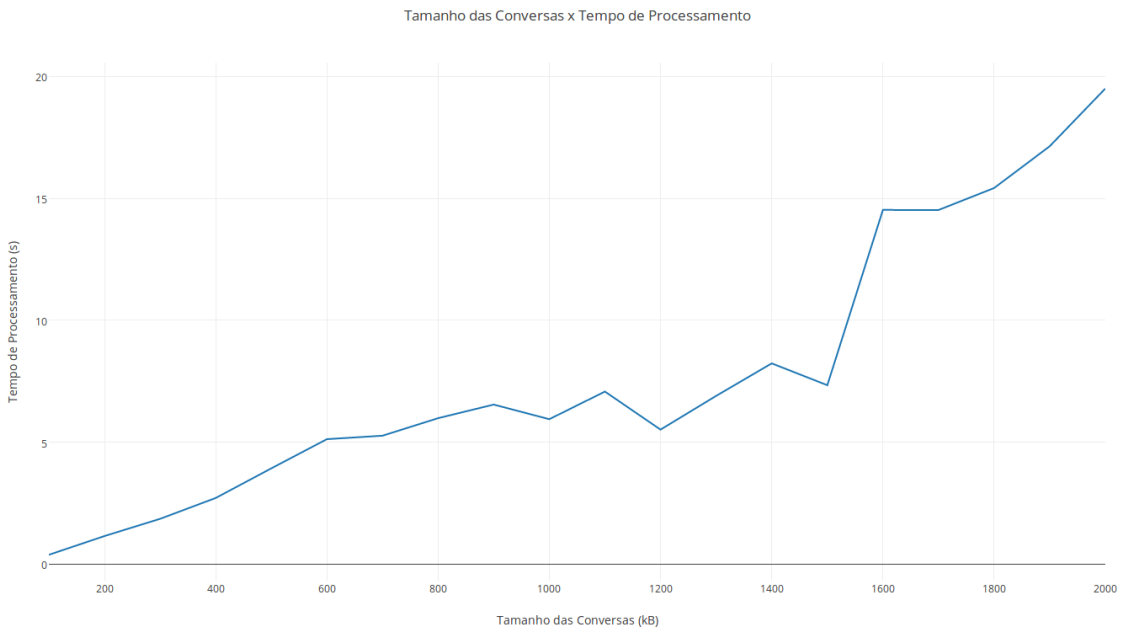

Figura 4 – Experimento com memória fixada e variação no tamanho das conversas

.

Por fim, o segundo experimento realizado foi fixar o tamanho das conversas e variar a memória disponível para o algoritmo. O resultado obtido, ilustrado pela Figura 5, apresenta um fator importante. Embora o tamanho do *buffer* aumente, o ganho pode não ser tão significativo, uma vez que, quanto maior esse *buffer*, mais operações para ordená-lo serão necessárias. Assim, o gráfico apresenta vários picos, sendo o aumento bom para a eficiência em alguns pontos e não em outros. Porém, em geral, o aumento de memória torna o algoritmo mais rápido, como é possível notar pela curva decrescente. Este fator depende também de como as partições são feitas, uma vez que o *QuickSort Externo* possui diferentes complexidades para cada um deles.

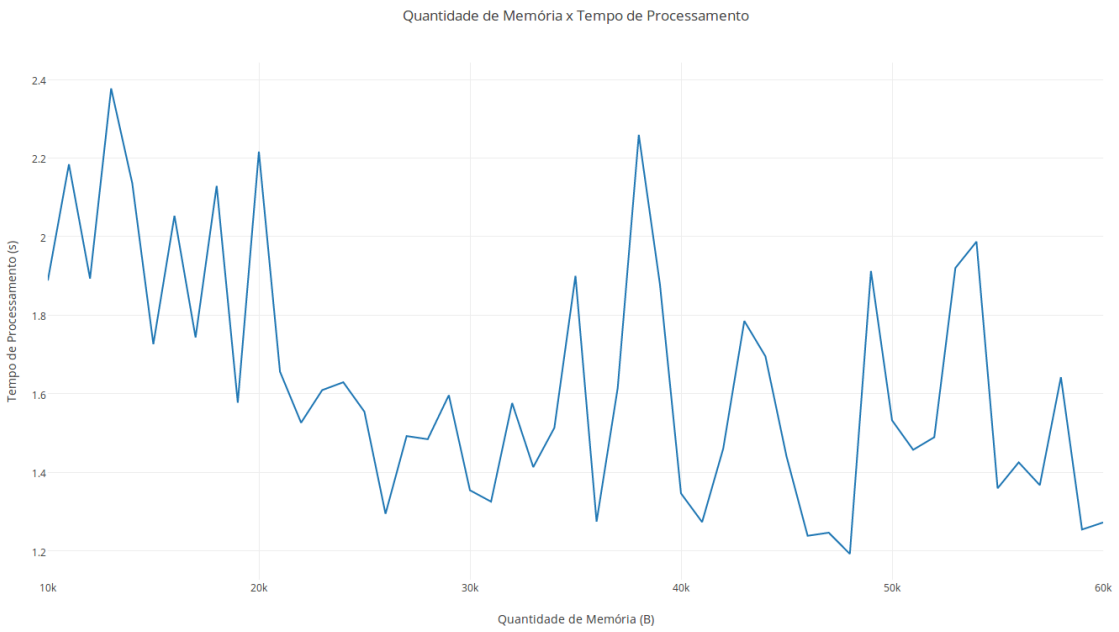

Figura 5 – Experimento com tamanho de conversas fixado e variação na memória disponível

.

5 Conclusão

Este trabalho apresentou uma solução para o problema de geração de um índice invertido dada uma coleção de vários documentos. Para tanto, foi utilizado o algoritmo *QuickSort Externo*, o qual se mostrou bastante eficiente na prática, dados os experimentos realizados. Além disso, a estrutura de arquivo binário foi de suma importância para a implementação deste trabalho, uma vez que permitiu um acesso facilitado aos blocos necessários tanto para a ordenação quanto para a geração do índice invertido em si.

6 Referências Bibliográficas

Cormen, Thomas H., and Thomas H. Cormen. Introduction to Algorithms. Cambridge, Mass: MIT Press, 2001.

ZIVIANI, Nivio. Projeto de Algoritmos com implementações em PASCAL e C, 3 ed. Cengage Learning, 2011.