

# Trabalho Prático 2

## Índice Invertido

### Algoritmos e Estruturas de Dados III - 2017/01

Pedro Otávio Machado Ribeiro

14/06/2017

## 1 Introdução

Neste trabalho temos como contexto o rapaz Hetelberto Topperson que gosta muito de conversar no *ZipZop*, porém ele possui uma memória seletiva e consegue lembrar do assunto que conversou com uma pessoa, mas não da pessoa. Hetelberto gostaria que o *ZipZop* permitisse buscas que retornem o trecho de uma conversa com alguém dado uma palavra qualquer. Dado isso, Hetelberto pediu ajuda para construir um índice invertido, já que sua amiga Inês implementara o buscador.

O problema consiste em, dado  $D$  arquivos de conversa de Hetelberto presentes num diretório  $E$  e  $M$  bytes de memória disponível, devemos escrever o índice invertido cada palavra de todos os arquivos em  $M$  no arquivo *index* dentro do diretório  $S$ . A solução do problema se resume a ordenar as palavras e obter os parâmetros que compõem o índice invertido. Para resolver este problema, os seguintes são necessários:

### 1.1 Ordenação Interna

Dado que para realizar uma ordenação externa com um limite de memória primária dado, é necessário saber como ordenar valores armazenados na memória primária. Para isso, escolhi o *Quicksort* para ordenar os dados. Para uma breve descrição sobre este método, veja [5].

### 1.2 Índice Invertido

O índice invertido, conhecido também como índice remissivo, é conhecido popularmente como uma lista no final de livros, artigos, etc, de palavras se-

guidas de números referentes às páginas em que estas podem ser encontradas. No caso deste trabalho, o índice invertido será uma *4-upla* representada por  $I = \langle w, d, f, p \rangle$ , onde  $w$  é a chave, no caso uma palavra,  $d$  o número do arquivo em que ela se encontra,  $f$  sua frequência no arquivo  $d$  e  $p$  sua posição em *bytes* em relação ao início de  $d$ .

## 2 Metodologia

Para solucionar o problema foi necessário utilizar um método de ordenação externa. No caso de minha solução, escolhi o *Mergesort* externo. Seja  $D$  a quantidade de conversas,  $M$  o tamanho da memória primária em *bytes*,  $E$  o diretório de conversas,  $S$  o diretório onde o índice invertido deve ser armazenado,  $T_I = 28$  o tamanho em *bytes* da estrutura que armazena o índice invertido sem a frequência. Para solucionar o problema, os seguintes passos foram tomados:

1. Para cada arquivo de conversa, leia-se  $C = M/T_I$  palavras até chegar ao final do arquivo, ordena-se todas as  $C$  entradas em memória interna utilizando o *Quicksort*, e finalmente escreve o resultado em um arquivo(página) temporário.
  - (a) Toda vez que uma palavra é lida, sua posição em *bytes* a partir do início é obtida e armazenada na estrutura de índice invertido, juntamente com o número do arquivo em que ela se encontra, que de antemão já sabemos qual é.
2. Agora, para cada novo arquivo temporário já ordenado, realizo o *merge* 2 a 2, até que ao final sobre apenas um arquivo, o qual possui todos índices invertidos, ainda sem a frequência, ordenados.
3. Finalmente, no arquivo ordenado com todos índices invertidos parciais, duas passadas são feitas para obter a frequência de cada palavra em seu respectivo arquivo de conversa. Este procedimento é feito de forma que toda vez que uma uma chave diferente é encontrada, outro ponteiro do arquivo parte da primeira ocorrência da chave e vai andando até chegar a última, escrevendo cada uma no arquivo *index* no diretório  $S$  com sua respectiva frequência.

## 3 Complexidade

### 3.1 Temporal

Seja  $C = M/T_I$  a quantidade de palavras definida na seção anterior. As  $C$  palavras são ordenadas utilizando o algoritmo *Quicksort* que, no caso médio, possui complexidade  $O(C \log C)$ . Como nenhum outro processo, ignorando a leitura em arquivos, possui complexidade maior que isso, então a complexidade temporal será

$$O(C \log C) \quad (1)$$

### 3.2 Passadas

Temos  $M$  como tamanho da memória principal. Podemos imaginar os arquivos de conversa como apenas um grande arquivo. Seja  $T_a$  o tamanho do grande arquivo. Seja  $N = T_a/M$  o número de páginas(subdivisões) de tamanho  $M$  do grande arquivo. Como criamos um novo arquivo ordenado a partir de dois ordenados, o número de passadas diminui pela metade a cada iteração. Para cada *merge* de arquivos realizado, lemos e escrevemos. Sabemos também que o *merge* das página é realizado em  $O(N)$ . Dessa forma, o número de passadas  $P$  sobre o arquivo será:

$$P = 2N * (\lceil \log_2 N \rceil + 1) \quad (2)$$

Assintoticamente, a complexidade será dada por:

$$O(N \log N) \quad (3)$$

### 3.3 Espacial

Como neste trabalho temos limite de memória interno dado por  $M$ , a complexidade espacial do programa será limitada superiormente por  $M$ , ou seja, a complexidade seria  $O(M)$ . Entretanto, ao desenvolver o algoritmo, temos que utilizar as funções de manipulação de arquivo da linguagem C, e estas utilizam a memória primária de forma que não temos controle.

## 4 Experimentos

Os testes foram realizados em meu computador pessoal, um notebook que não é mais um notebook, com as seguintes configurações:

- Sistema Operacional: Arch Linux
- Processador: Intel(R) Core(TM) i5-2430M CPU @ 4 × 2.40GHz
- Memória RAM: 6GB

Cada teste foi realizado 30 vezes e os valores apresentados é referente a média destes.

Os casos de testes utilizados foram gerados utilizando um gerador de frases aleatórias [3] juntamente com uma ferramenta de textos [1] que remove letras maiúsculas e pontuações.

Optei por gerar 10 conversas aleatórias, cada uma com aproximadamente 88 bytes de conteúdo, totalizando 888 bytes de conversas. Para cada caso de teste, utilizei sempre as mesmas 10 conversas, variando apenas o tamanho da memória primária disponível, começando de 320 bytes e dobrando até chegar em 163840 bytes. Abaixo segue o gráfico do tempo em função do tamanho da memória primária:

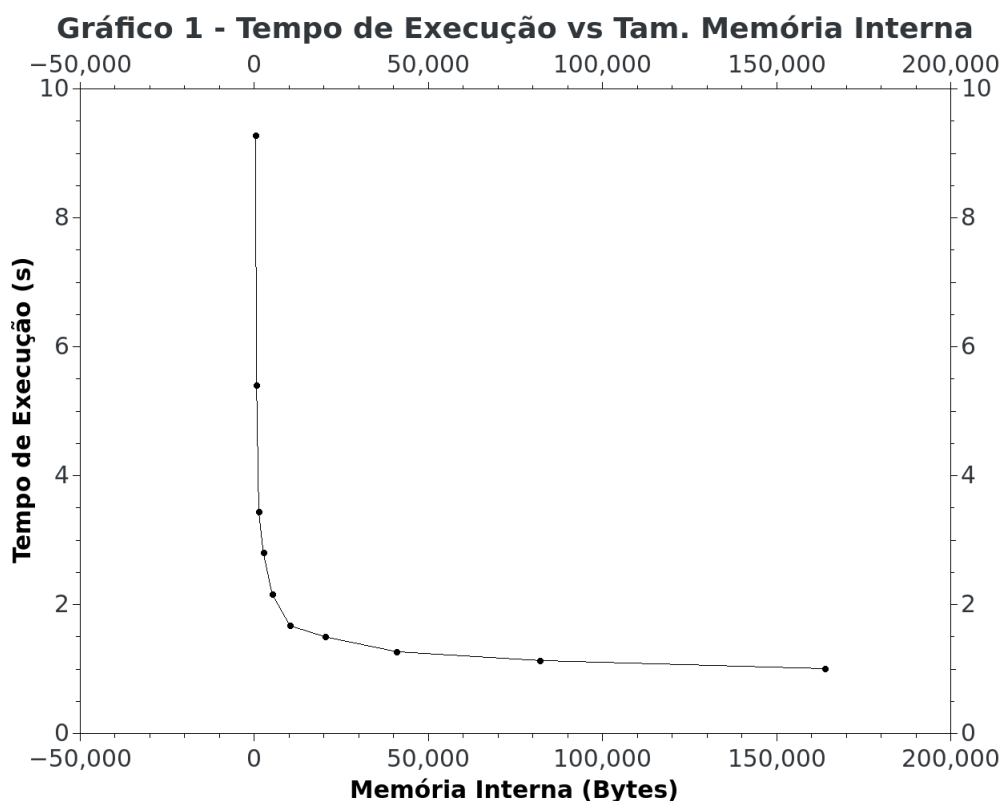


Figura 1: Tempo de Execução em função do tamanho da Memória Primária

## 5 Análise de Resultado

Note que a cada vez que a memória primária disponível é dobrada, o tempo cai aproximadamente pela metade, pois o número de páginas(arquivos) criadas diminui pela metade, corroborando assim com a complexidade do problema. É possível perceber que em alguns intervalos não ocorre a diminuição do tempo pela metade. Isso ocorre possivelmente devido ao tempo de uso de meu notebook(6 anos), além de que outros programas também poderiam estar utilizando o disco ao mesmo tempo.

## 6 Conclusão

O objetivo do trabalho foi alcançado. O programa retorna para Hetelberto Topperson o índice invertido de todas conversas em conjunto devidamente ordenado.

O problema de ordenação externo é um dos problemas clássicos da computação, e é amplamente utilizado em aplicações de *database*, como a máquina de busca do *Google* ou de qualquer outro buscador.

No *Mergesort* externo que implementei, utilizo apenas dois *buffers* para fazer o *merge* das páginas. Uma melhora que poderia ser feita neste programa é utilizar mais de dois *buffers*, diminuindo mais ainda o tempo de execução do programa.

## Referências

- [1] John Anderson. *GoRewrite.com*. URL: <http://www.gorewrite.com/modify.html>.
- [2] Berkeley University of California. *Midterm Review*. URL: <http://db.cs.berkeley.edu/dbcourse/section/mreview2.pdf>.
- [3] Marcus Tullius Ciceros. *Lorem Ipsum*. URL: <http://www.lipsum.com/>.
- [4] Wikipedia. *External Sorting*. URL: [https://en.wikipedia.org/wiki/External\\_sorting](https://en.wikipedia.org/wiki/External_sorting).
- [5] Wikipedia. *Quicksort - Wikipedia*. URL: <https://en.wikipedia.org/wiki/Quicksort>.