Índice Invertido

Luiz Otávio Resende Vasconcelos

Junho 2017

1 Introdução

O presente trabalho visa solucionar o problema o qual descreve um aplicativo de mensagens em que é necessário uma busca por ocorrências em palavras no histórico. Para isso, devemos gerar um arquivo com índices invertidos, ordenado, contendo cada palavra e as respectivas informações sobre ela: conversa em que se encontra, número de vezes que se repete naquela conversa e sua posição na conversa respectivamente.
Porém, o espaço na memória do celular será limitado sendo assim necessário a implementação de uma ordenação externa.
Para isso será alocado o limite de memória em bytes para um array no começo do código, e então, apenas esse array será utilizado para realizar todos os cálculos envolvendo os arquivos.

O presente trabalho foi desenvolvido na linguagem C.
Para executá-lo, basta utilizar os comandos em um bash/unix:
$ make
$ ./inverted_index

Será utilizada a pasta *./tmp* para alocar os arquivos utilizados temporariamente, por isso, é necessário que a pasta exista e as permissões estejam liberadas.
O projeto completo e as etapas do desenvolvimento podem ser encontrados no repositório:
https://github.com/Luiz0tavio/inverted_index

2 Implementação

Como foi informado na descrição que as palavras da conversa não ultrapassariam 20 bytes e que o conjunto (palavra,arquivo,quantidade,posição) não ultrapassaria 32 bytes, escolhemos então trabalhar com blocos de 32 bytes cada, ou seja, assumimos que toda palavra ocupará 20bytes na memória e todo conjunto, 32bytes.

2.1 Gerando os Sufixos

Sufixo se refere ao conjunto (arquivo,quantidade,posição) que se sucede logo após cada palavra no arquivo final.
Como os sufixos são dependentes apenas da conversa em que a palavra se encontra, é mais fácil gerá-los primeiro e então, realizar a ordenação e gerar o arquivo final, pois, as palavras serão mescladas no decorrer da ordenação. Apesar de mais fácil, terá um custo extra pois, a palavra ficará mais extensa e custará mais na memória resultando em mais passadas.

Geramos um único arquivo chamado *suffixed* no diretório *./temp* com todas as palavras de todas as conversas já com seus respectivos sufixos corretos, bastando então apenas ordenar o arquivo para obter o resultado esperado. Para gerarmos tal arquivo, seguimos o seguinte pseudo-código:

```
posicao_palavra = 0;

para (cada conversa)
{
    para (cada palavra na conversa)
    {
        escolha a palavra;
        conta_palavra = 0;

        para (cada palavra na conversa)
        {
            se (palavra escolhida == palavra)
            {
                conta_palavra++;
            }
        }
        escreve no arquivo de saida(
            <palavra,
            indice do loop de conversas,
            conta_palavra,
            posicao_palavra\n>
        );

        posicao_palavra += strlen(palavra) + 1;
    }
}
```

Lembrando que como a memória é limitada, não é possível trazer todas as palavras de uma só vez para a memória principal. É necessário trazer apenas o limite da memória, fazer os cálculos e então sobrescrevê-las.

Neste momento já obtemos o arquivo final com todas as palavras com seu devido

sufixo, com quebra de linha as separando, porém, não ordenadas. Basta então agora utilizar um algóritmo de ordenação externa para obter-mos o resultado final. O algorítmo é descrito na próxima secção.

2.2 Algorítmo de Ordenação Externa

Após os arquivos com os respectivos sufixos serem devidamente gerados utilizamos o algorítmo de **Intercalação Balanceada** para realizar a ordenação.

Cada passada do algorítmo, incrementaremos a letra que será o prefixo do nome dos nossos novos arquivos temporários. Começando em 'a', seguido por 'b' etc.

2.2.1 Arquivos Pré-Intercalação Balanceada

Para executar-mos o algorítmo de Intercalação Balanceada, devemos primeiramente gerar blocos ordenados de tal modo que possamos então intercalálos.

Será então gerado p arquivos, ordenados, com no máximo $M/32$ registros (onde M é o tamanho da memória e p é n/M, onde n é o tamanho total em bytes de todas as conversas somadas), ou seja, p é o número de arquivos necessários tal que cada um armazene no máximo M bytes.
Para gerar esses arquivos, seguimos o seguinte pseudo-código:

```
indice_arquivo = 1;

enquanto (arquivo suffixed != EOF)
{
    selecionar palavras ate M bytes;

    qsort(palavras);

    escreve as palavras no arquivo 'a'+indice_arquivo;

    indice_arquivo++;
}
```

Note que *qsort()* functiona muito bem neste caso utilizando *strcmp()* como função de comparação.
Para cada iteração, *indice_arquivo* é incrementado. A variável *indice_arquivo* será nosso sufixo no nome do arquivo temporário da passada. Como ainda estamos gerando os primeiros arquivos, serão nomeados 'a1', 'a2, ... , 'ap'.

2.2.2 Intercalação Balanceada

Agora que temos os 'a1', 'a2, ... , 'ap' arquivos já ordenados, fazemos a intercalação balanceada entre eles e geramos o arquivo final, seguindo o pseudo-código:

```
pref_file = 'a';

enquanto (houver arquivos com o prefixo pref_file)
{
    abra p* arquivos;

    enquanto (arquivos != EOF)
    {
        escolha a palavra de menor ordem alfabetica utilzando strcmp();

        escreva a palavra no arquivo;

        coloque a proxima palavra do arquivo no lugar dessa;
    }
}

// * p e o numero maximo de palavras que cabem na memoria = M/32
```

Gerando por fim o arquivo final desejado.

3 Estrutura do Projeto

Segue uma breve exposição de como o projeto está particionado.

3.1 main.c

Arquivo principal do projeto. Inicializa as varíaveis, faz a leitura das entradas, faz a chamada das principais funções.

3.2 suffixed.h / suffixed.c

Cria um único arquivo com todas as palavras já com seus respectivos sufixos.
Método descrito na secção 2.1.

3.3 filesBlocksOrdered.h / filesBlocksOrdered.c

Criam os arquivos com os blocos devidamente separados e ordenados necessários para a intercalação balanceada.

3.4 balancedInterleaving.h / balancedInterleaving.c

Executa a Intercalação Balanceada responsável por gerar o arquivo final a ser apresentado.
Método descrito na secção 2.2.

3.5 utils.h / utils.c

Arquivo com funções úteis.
Utilizado pelas funções principais para chamar métodos comuns.

4 Análise Assintótica

4.1 Temporal

Para gerar o arquivo *suffixed* descrito na secção 2.1 temos um loop externo até k, onde k é o número de conversas (primeiro parâmetro da entrada).
Em seguida, para cada palavra do arquivo, lemos todo o arquivo para contarmos quantas iguais estão presentes, ou seja, $p * p$, ou, p^2.
Logo, para gerar o primeiro arquivo temos $\theta(k * p^2)$

Para gerarmos os arquivos com prefixo 'a', ou seja, aqueles que estão separados em blocos ordenados para a intercalação, gastamos $M/32$ iterações, onde M é o tamanho da memória.
Na Intercalação Balanceada, o que nos interessa é o número de passadas, a qual apresenta complexidade $O(log(n/m))$. Um bom algorítmo não fará mais que 10 passadas.
Portanto, nosso programa apresenta uma complexidade $\mathbf{O}(k * p^2 + M/32 + log(n/m))$.

4.2 Espacial

Nosso programa deve ser focado em realizar operações com memória secundária limitado à entrada passada.
Por isso, alocamos um array no início do programa, com o valor passado na entrada, que será utilizado em todo código.
Portanto, podemos afirmar que, a medida que M cresce, nosso algorítmo tende a ocupar Mbytes (onde M é a entrada fornecida em bytes).

5 Avaliação Experimental

Primeiramente, geramos casos de testes que variam apenas a entrada em Bytes, fixando o limite de memória em 640bytes.
Foram gerados 10 arquivos de 100bytes cada. Na *Figura 1* abaixo é mostrado que a medida que aumentamos o tamanho da conversa, o tempo cresce, aproximadamente, em grandeza linear nesse intervalo.

Em um segundo teste, variamos a memória disponível.
Em princípio, um aumento da memória disponível pode parecer vantajoso para a execução do algorítmo. Porém, com o aumento da memória, aumenta-se a quantidade de palavras que podem ir para a memória, e, devido à um grande número de loops de 0 até o limite de palavras, o programa como um todo ficará

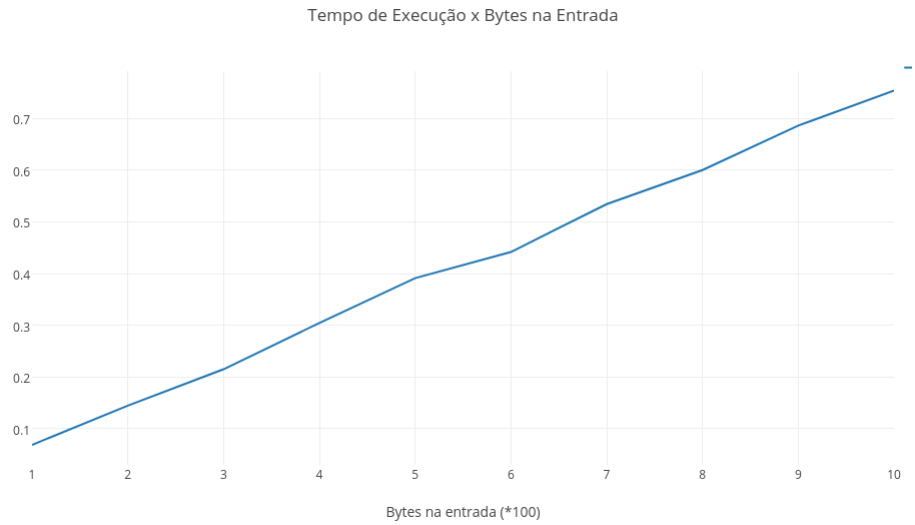

Figura 1: Gráfico Tempo Execução / Bytes Entrada

mais custoso. A *Figura 2* mostra como o tempo de excução reage à um aumento de memória disponível.

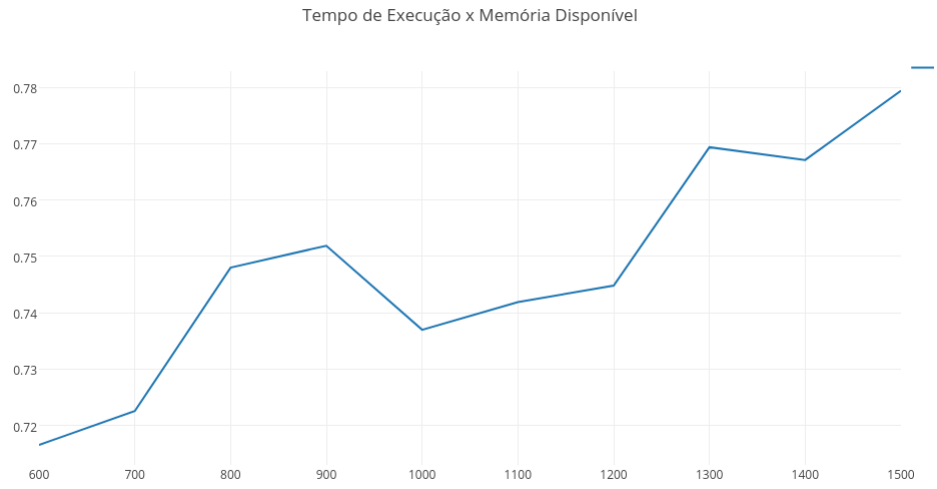

Figura 2: Gráfico Tempo Execução / Memória disponivel

6 Conclusão

Ordenação Externa é um tópico de extrema importância em Ciência da Computação. Tem importante aplicação em Bancos de Dados, Big Data, etc. Devemos ter cuidado com o acesso à memória secundária, pois, é muito mais custoso que à memória primária, por isso, um bom algorítmo não faz mais que 10 passadas em um arquivo, pois, apresenta crescimento $log(n/m)$.