Laboratórios de Informática III

Trabalho Prático nº 1

Relatório de desenvolvimento

José Resende	Miguel Lobo	Patrícia Barreira
(a77486)	(a78225)	(a79007)

1 de Maio de 2017

Conteúdo

Capítulo 1

Introdução

Este trabalho prático tem como objetivo construir um sistema que permita analisar os artigos presentes em *backups* da *Wikipedia*, realizados em diferentes meses, e extrair informação útil desse período de tempo, como por exemplo, o número de revisões ou ainda o número de novos artigos. Os problemas foram repartidos em diferentes tarefas, tarefas estas que propunham uma forma eficiente da resolução dado o volume de informação a tratar. Todo o código desenvolvido nas diferentes tarefas era submetido através do repositório *GIT* e avaliado na plataforma criada pelos docentes. Este relatório pretende sumarizar todos os parâmetros estudados e desenvolvidos durante este projeto, bem como os passos que foram seguidos e tomados.

Capítulo 2

Concepção do problema

Pela análise do problema, foi possível dividir as *queries* em 3 grupos distintos:

- Artigos
- Contribuidores
- Texto

Nestes três grupos, chegou-se a conclusão que os id's é que iriam gerir a estrutura de dados.

Para as *queries* de artigos são necessários os respetivos id's, logo torna-se imperativo desenvolver uma estrutura de dados onde a procura de um artigo seja o mais rápido possível. De notar que cada artigo podia ter mais do que uma revisão, assim, é necessário implementar uma estrutura de dados associada à estrutura de cada artigo capaz de armazenar as suas revisões únicas.

No caso das *queries* de contribuidores precisa-se de uma estrutura que guarde os nomes de contribuidores , ids e número de contribuições dos mesmos.

Já para os textos, as queries focam-se no número de palavras e tamanho do texto. Por isso, será necessário guardar numa estrutura, o número de palavras , número de carateres e por fim, id e título de artigos.

Capítulo 3

Concepção da Solução

Decidiu-se implementar como estrutura principal uma estrutura composta por uma *Hash Table*, em que cada posição da *Hash Table* é constituída por uma *Árvore Binária Balanceada*, sendo que cada nodo desta, contém uma *Lista Ligada*.

A escolha desta estrutura assenta nos seguinte fatores:

- A forma como os artigos são organizados para a mesma posição da *Hash Table* é através de uma *Árvore Binária Balanceada* com fator de comparação o id. A *Hash Table* foi criada com intuito de reduzir o tempo de procura na *Árvore Binária* e, consequentemente, aumentar o número de *Árvores Binárias* existentes. Este aumento conduz a uma redução das suas alturas o que implica uma procura mais eficiente do artigo. Este tempo de procura em cada árvore é `O(logN)` (em que N é o numero de nós da árvore, ou seja, número de artigos).
- Em cada nodo da Árvore de artigos (referida no ponto acima) será guardada informação referente às suas revisões sob a forma de lista ligada.
- Era necessário implementar uma estrutura eficiente com travessias, com intuito de preencher estruturas auxiliares e ao mesmo tempo conseguir armazenar toda a informação necessária.

Para o caso do grupo de *queries* referentes aos artigos apenas se utilizou a estrutura principal, visto que, através de uma simples travessia, é possível obter a informação requerida.

No que toca ao grupo de *queries* associadas aos contribuidores, foi criada uma *Árvore Binária Balanceada*, onde cada nodo contém uma estrutura `CONT` com imformação relativa aos nomes de contribuidores, ids e número de contribuições dos mesmos. Foi ainda implementada uma *MaxHeap*

com intuito de resolver a *query* referente aos Top contribuidores, uma vez que se trata de uma estrutura de dados bastante eficiente para este tipo de interrogação.

Por fim, para o grupo das *queries* de texto, utilizou-se a estrutura principal e, para o caso em que as interrogações eram do tipo `Top`, foi implementada uma *MaxHeap*.

Capítulo 4

Estruturas de Dados

Foi utilizada a biblioteca *Glib* para a implementação das estruturas utilizadas neste projeto. Em baixo, encontra-se representada a estrutura que contém todas as estruturas implementadas ao longo do projeto.

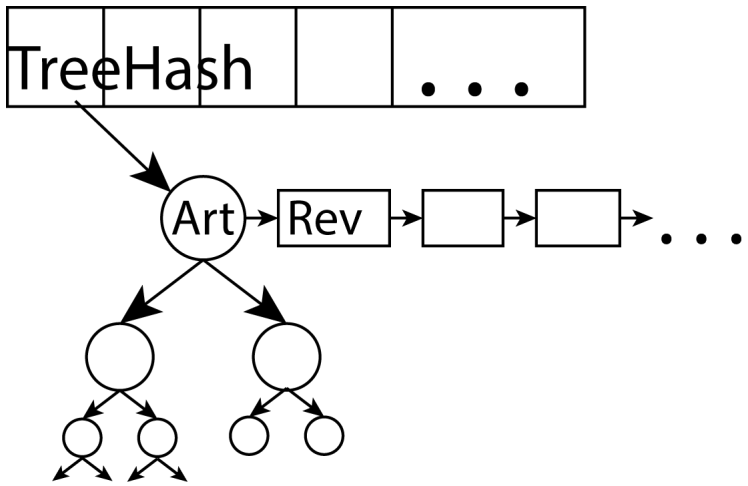
```
typedef struct TCD_istruct {
    TreeHash treeTable;
    xmlArray xmlInfo;
    CTree ContriTree;
    Heap TopN;
    Heap Top20;
    long topCont[10];
    long Atop20[20];
    long *AtopN;
    char *pref[1];
    long numeroArtigos;
    long numeroRevisoes;
    long numeroTotal;
    long numeroContributor;
}*TAD_istruct;
```

Nas secções seguintes será detalhada cada uma das estruturas apresentadas em cima.

4.1 TreeHash

Para a implementação da *Hash Table* utilizou-se a estrutura da *Glib Pointer arrays*. Para *Árvores Binárias Balanceadas* foi utilizada a estrutura *Balanced Binary Trees* e, por fim, *Singly-Linked Lists* para listas ligadas.

Posto isto, a estrutura apresenta a seguinte forma:



- *TreeHash*

```
typedef GPtrArray *TreeHash;
```

- Em cada índice desta *TreeHash*

```
typedef GTree *ArtTree;
```

- Em cada nodo da árvore tem-se:

```
typedef struct artigo {
    char *id;
    char *titulo;
    RevList revisao;
}*ART;


typedef GSList *RevList;
```

- Em cada `RevList` a estrutura das revisões é implementada da seguinte forma:

```
typedef struct revisao {
    char *id;
```

```
        char *timestamp;
        char *content;
    }*REVISION;
```

4.2 xmlArray

Esta estrutura é usada com vista a guardar as árvores resultantes do *parsing* dos *snapshots*.

```
typedef GPtrArray *xmlArray;
```

4.3 CTree

Estrutura que contém as informações dos contribuidores.

```
typedef GTree *CTree;

typedef struct contribuidor {
    char *id;
    int contN; //Numero de contribuicoes
    char *nome;
}*CONT;
```

4.4 Heap

Para as *queries* cuja finalidade é a obtenção dos `top's` decidiu-se que a melhor implementação seria uma `max-heap`. Nesta, bastaria extrair a raíz, reorganizar com vista a ficar o segundo maior na raiz e repetir este processo até se extrair o número de elementos pretendidos. A `struct` elemento foi criada com o intuito de reutilizar o código da `heap` para as três *queries* de top's.

```
typedef struct elemento{
    char *id;
    long count;
}elem;
```

```
typedef struct heap{
    int   size; /* Tamanho alocado para a Heap.*/
    int   used; /* Número de elementos da Heap. */
    elem *array;
}*Heap;
```

Capítulo 5

Desenvolvimento das Queries

Para as *queries* `all_articles`, `unique_articles` e `all_revision`, decidiu-se que estas deviam ser resolvidas aquando do *load* uma vez que à medida que os artigos e as revisões são inseridos na *HashTable*, facilmente se incrementava uma variável de contagem. Assim, houve a necessidade de criar 3 variáveis *numeroTotal*, *numeroArtigos* e *numeroRevisões* na estrutura global *TADistruct* que podem ser atualizadas no *load* e representam, respetivamente, o resultado das 3 *queries*. Assim o tempo de resposta a estas 3 *queries* passa a ser `O(1)`.

No caso das *queries* `article_title` e `article_timestamp`, basta calcular o `hashcode` do id recebido como argumento e, percorrer a árvore até ao nodo em questão. A segunda *query* requer ainda que se percorra a lista ligada até à revisão pretendida. Nestes casos ajudou bastante a criação da *HashTable* uma vez que a árvore tem uma altura menor.

No que toca à *query* `contributor_name`, a respetiva estrutura *Árvore Binária Balanceada*, será adicionada à estrutura global `TAD_istruct` de modo que quando é feito o *load*, esta árvore seja carregada à medida que se percorre a árvore resultante do *parsing*. A pesquisa para encontrar o nome do contribuidor através do id será então `O(logN)` em que N é o numero de contribuidores(nodos).

Relativamente às *queries* `top_10_contributors`, `top_20_largest_articles` e `top_N_articles_with_more_words`, no que refere à primeira, basta percorrer a *árvore dos contribuidores* guardada na estrutura `TAD_istruct` e, inserir na `heap` os respetivos valores. Estes passos são acompanhados pela preservação do invariante em que o pai é sempre maior que os respetivos filhos. O tamanho da `heap` é dada pela variável *numeroContributor*, pertencente à estrutura `TAD_istruct`. Esta variável é incrementada durante o *load* à medida que são inseridos os

contribuidores na *árvore de Contribuidores*. O resultado é então guardado na estrutura principal `TAD_istruct` como *topCont*, evitando assim percorrer a árvore de contribuidores e inserir na `heap`, sempre que a *query* `top_10_contributor` é invocada. Esta metodologia adotada permite ainda que o espaço alocado para a `heap` possa ser libertado depois da obtenção da informação requerida. Para as restantes duas *queries* será aplicado o mesmo critério, contudo, para estes casos, é a *Hash Table* que é percorrida. Em cada revisão do artigo, é contado o tamanho e o número de palavras do texto (de uma só assentada). Estes 2 resultados são guardados num `array` de 2 posições e em seguida são criadas 2 *max-heaps* (*Top20* e *TopN*), uma para cada *query*. Evitando assim, mais uma vez, pesquisas excessivas na estrutura *Hash Table*.

Por fim, na *query* `titles_with_prefix` é aplicada a função `prefix_titles_art` a cada um dos artigos, recorrendo à instrução `foreach`. Esta função, para ter compatibilidade com a *Glib*, recebe o mesmo artigo nos seus dois primeiros argumentos, o `key` e o `value`, e no último argumento, uma `struct Data`. Esta estrutura foi criada com o intuito de transportar:

- `[r]` Apontador para lista de títulos que se encontra na `TAD_istruct`, que será a resposta da *query*.
- `[used]` Número de títulos que se encontram na lista incluindo, no final, `NULL`.
- `[size]` Espaço disponível para a inserção de títulos na lista (excluíndo o `NULL`).
- `[prefix]` Prefixo a ser procurado.

A função `prefix_titles_art` verifica se cada artigo contém o prefixo no título e, em caso positivo, adiciona-o a `r`, através da instrução `append`. Numa tentativa de otimizar a *query*, experimentou-se inserir os títulos na lista, tendo esta sido previamente ordenada recorrendo à função `insertString`. No entanto, não foi possível obter um nível de eficiência superior. A melhor solução passou então por ordenar a lista no final do algoritmo através da função `qsort` da `stdlib`.

Capítulo 6

Otimização

Para esta secção, as escolhas mais óbvias foram a utilização da biblioteca *Glib* para a implementação das estruturas de dados utilizadas. E também a utilização da API *OpenMP* para a introdução de paralelismo. Outro aspeto considerado importante foi a utilização de uma *Max-Heap* como estrutura auxiliar das *queries* que envolviam o prefixo `top's`.

6.1 Dificuldades e Superações

A utilização da *Glib* levantou algumas dificuldades. De realçar a composição entre as estruturas, nomeadamente entre *Hash Table* onde em cada posição desta temos uma *Árvore Binária Balanceada* e em cada nodo existe ainda uma `Lista Ligada`. Isto aliado ao facto do grupo de trabalho não estar habituado a implementar e utilizar funções genéricas, introduziu uma certa complexidade à resolução do problema.

Já em relação ao *OpenMP* a principal dificuldade residiu em encontrar partes do código que não sejam interdependentes. Assim, decidimos unicamente aplicar a diretiva `OpenMP` às árvores criadas no *parsing* dos inúmeros ficheiros (*snapshots*) através da `xmllib`, onde é criada uma árvore para cada *snapshot*. De notar que tudo isto só foi possível graças à compatibilidade da `xmllib` com paralelismo.

```
xmlArray parseMult(xmlArray array, int N, char *filename[N]){
    int i;
    if(array != NULL){
        #pragma omp parallel for
```

```
        for(i = 0 ; i < N ; i++)
            g_ptr_array_add(array, parse(filename[i]));
    g_ptr_array_set_size (array, N);
    }
    return array;
}
```

Assim consegui-se um *real time* de aproximadamente 6segundos e um *user time* de aproximadamente 10segundos. Houve uma melhoria do *real time* e um aumento no *user time* porque o processo está ligado ao CPU e aproveita a execução paralela em vários núcleos/CPUs, enquanto que sem openMP o real time era aproximadamente igual ao user time porque o processo está ligado ao CPU e não tem qualquer vantagem de execução em paralelo.

Por último em relação à *Max-Heap*, esta foi escolhida com intuito de evitar criar um *Array* com todos os id's e o número que dita a ordem (e.g. número de contribuições dos contribuidores) e consequentemente ordena-lo para retornar os n maiores. A *Max-Heap* é então uma melhor alternativa.

Capítulo 7

Modularidade

Atendendo à dimensão do projeto, foi necessária uma organização cuidada do código para que este seja controlável e legível, em todas as suas fases (desenvolvimento, teste e manutenção). O projeto foi dividido em 3 partes essenciais: *load*, *init* e *parse*. No *load* os dados são carregados para a estrutura, no *init* as estruturas são inicializadas e, por fim, no *parse* é feito o *parsing* dos *snapshots* através da biblioteca `xmlib`.

As queries foram organizadas nos seus 3 grandes tipos: artigos, contribuidores e textos. As estruturas auxiliares surgem num novo módulo bem como o *clean*.

Capítulo 8

Conclusão

As *queries* implementadas respondem aos requisitos impostos pelo enunciado, pelo que este grupo de trabalho se considera sobremaneira satisfeito. É no entanto, evidente, que muitas outras otimizações poderiam ser acrescentadas, tanto na função que conta as palavras e tamanho de um texto como uma maior aposta no paralelismo. Seria portanto desejável, como exercício futuro, implementar a função *count*, integrando, por exemplo, a diretiva *openMP*, etc.Existem portanto diversas formas de continuar o trabalho que aqui foi produzido, todas elas revestidas de utilidade prática.