Relatório do Projeto em *C*

António Sérgio (a78296) Hugo Brandão (a78582)
Tiago Alves (a78218)

Abril 2017

Universidade do Minho
Departamento de Informática
Laboratórios de Informática 3

Mestrado Integrado em Engenharia Informática

Conteúdo

1 Introdução

Este relatório aborda a resolução do projeto prático em C de LI3. O projeto consiste, resumidamente, em construir um sistema que permita analisar os artigos presentes em backups da Wikipedia, realizados em diferentes meses, e extrair informação útil para esse período de tempo como, por exemplo, o número de revisões, o número de novos artigos, entre outros.

Para ajudar nesta tarefa, foi necessário fazer o *parse* dos ficheiros e também criámos quatro módulos. Estes módulos são: o *parse* dos ficheiros dados, as estruturas de dados, código auxiliar para executar corretamente as queries fornecidas pelos professores, e um *program* para teste.

Depois dos ficheiros serem carregados, somos capazes de executar uma lista de queries disponibilizadas pela equipa docente. Para responder às diferentes queries utiliza-se as funções definidas nas API dos diferentes módulos já referidos.

Ao longo deste relatório aborda-se assim as decisões tomadas na implementação do projeto, nomeadamente quais as estruturas utilizadas para criar cada um dos módulos, o porquê das escolhas feitas e as suas APIs.

A secção final destina-se a anexos, isto é, excertos de código fundamental mas de demasiada dimensão, mas que apenas ajudam na melhor compreensão das soluções encontradas. O relatório em si, sem os anexos, possui toda a informação relevante, clara e independente da secção *Anexos*.

2 Módulos

Nesta secção apresentam-se excertos do código comentados, bem como explicações das soluções usadas e os 3 grupos em que dividimos o projecto.

2.1 Estrutura de Dados (*struct*)

Para guardar toda a informação necessária lida a partir dos *snapshots*, resolvemos usar duas *hashtables*, umas para os artigos e outra para os contribuidores. Cada artigo tem a sua *linked list* para as revisions. Decidimos usar estes métodos pois, apesar de necessitar de mais instruções, a procura é muito rápida. A justificação para a *linked list* é que tem uso e implementação simples, e também porque temos duas hash tables, logo não havia necessidade de outra implementação que fosse mais complicada de usar.

2.1.1 TCD_istruct

A nossa TCD_istruct tem quatro parâmetros, que são o hashsize, uma struct para os artigos, uma outra para os contribuidores e um inteiro *occupied_articles*.

O inteiro *hashsize* quase fala por si só, possui o tamanho das hashtables, já que este é calculado em função do número de artigos totais nos *snapshots*.

O inteiro *occupied_articles* possui o número de artigos únicos, ou o número de artigos ocupados, na hashtable dos *articles*.

```
1 typedef struct TCD_istruct{
2   int hashsize; //size of the hash tables (-1 if the hash tables
        have not been allocated)
3   Article *articles; //declaring a hash table for the articles
        called articles
4   Contributor *authors; //declaring a hash table for the
        contributors called author
5   int occupied_articles; //number of elements inside the articles's
         hash table
6 }TCD_istruct;
```

2.1.2 Article

A *articles struct* é uma hashtable criada para os artigos. Nesta, nós temos o número de contribuições de cada artigo, o seu id, uma lista ligada para as *revisions*, e um inteiro "boolean"para referir se está ou não ocupado.

```
1 typedef struct Article{
2   int nrevisions; //number of revisions of the article
3   struct RevisionList *revisions;
4   int isOccupied; // 0 is not occupied and 1 is occupied
5   int id; //the value of the id that was hashed -> -1 if it's empty
6 }Article; //hence, an Article
```

2.1.3 Revision

Cada Revision possui um id próprio, um título, data, o id do seu autor, o número de palavras e caracteres do texto e um apontador para a próxima revisão da *linked list* do artigo.

```
typedef struct RevisionList{
  int id; //id of the revision
  char *title; //title of the revision
  char *date; //date of the creation of the revisions
  int idcontributor; //id of the contributor that wrote the
  int nwords; //number of words of the revision's article
  int nchars; //number of characters of the revison's article
  struct RevisionList *next; //pointer to the next element of the
      revision list
}RevList; //pretty much connected lists (exactly that)
```

2.1.4 Contribuitor

Por outro lado, a estrutura dos contribuidores segue o mesmo princípio que a dos artigos, mudando apenas a informação guardada. Cada contribuidor possui um id, um nome e a quantidade de contribuições por esse autor.

```
typedef struct Contributor{
  int id; //id of the contributor
  char name[256]; //name of the contributor
  int ncontributions; //number of contributions of this author
  int isOccupied; //0 is not occupied and 1 is occupied
}Contributor; //hence, a block called Contributor;
```

2.2 Parse

Esta secção descreve como foi feita uma das partes fundamentais do projeto, o parse. Para isto, usamos funções já fornecidas pela biblioteca *libxml2*.

É também nesta parte que a estrutura é iniciada em função do número total de artigos e os dados são carregados para a estrutura, logo é compreensível que seja uma query mais demorada.

Visto que lidamos com uma grande quantidade de informação foi necessário utilizar várias funções já incluidas na biblioteca *libxml2* para anular a memory leak.

2.2.1 ParseDoc

É nesta função que verificamos se o ficheiro fornecido é de *xml* e se está vazio antes de fazer o processamento da informação. Caso não seja de xml ou esteja vazio, a função termina e retorna um erro, caso contrário, um *xmlNodePtr* é enviado para a *ParseDocID* onde iremos processar a informação.

```
if (doc == NULL ) {
    fprintf(stderr,"Document␣not␣parsed␣successfully.␣\n");
    return 0;
  }

  cur = xmlDocGetRootElement(doc);

  if (cur == NULL) {
    fprintf(stderr,"empty␣document\n");
    return 0;
  }
```

2.2.2 ParseDocID

ParseDocID é a maior função do nosso projeto e é onde toda a informação é processada e guardada na nossa struct.

Nesta função recorremos a *while*s para procurar os nodos que queremos obter para, seguidamente, encontrar a informação necessária.

```
while(aux!=NULL){
        if(!(xmlStrcmp(aux->name, (const xmlChar *) "title"))){
          con = (char*) xmlNodeGetContent(aux);
        }
```

Como se pode ver no pedaço de código acima apresentado, percorremos um *while* até encontrar um nodo com um certo nome (naquele caso é "title") e quando esse for encontrado guardamos a informação nele contida numa variável, após acabar uma página inserimos toda a informação sobre essa na struct com a função insertID.

```
      insertID(id, revisionid, usernameid, con, username, t, text,
          qs);
      xmlFree(text);
```

Nesta função foi necessário fazer vários *frees*, caso contrário obtinhamos grandes quantidades de memory leak.

2.2.3 ParseCount

Como é nesta secção que a struct é iniciada foi necessário criar uma função que contasse o número de pages, ou seja, o número total de artigos para que podessemos alocar uma certa quantidade de memória para a nossa struct. Essa função é a *ParseCount*.

Esta função basicamente é uma junção da *ParseDoc* com a *ParseDocID*, ou seja, verifica se o ficheiro fornecido é de *xml* e se está ou não vazio, caso esteja tudo bem esta conta o número total de page nodes.

```
1 while (aux != NULL ){
2     if(!(xmlStrcmp(aux->name, (const xmlChar *) "page"))) p++;//
          everytime there is a page node the p variable is
          incremented
3     aux = aux->next;
4   }
```

2.3 Queries

Nesta secção fazemos menção ao modo como decidimos executar as queries fornecidas e de que maneira chegamos ao resultado esperado pelo grupo e pela equipa docente. Por ocuparem demasiado espaço, o código referente a cada query está na sua completude na secção referente aos anexos.

2.3.1 Init

A query init não faz, na prática, nada útil. Decidimos alocar memória para a estrutura em função do número total de artigos, para não ter de realocar a hastable no caso de haver mais elementos para inserir do que espaços na hashtable. A verdadeira inicialização é feita no *load* onde existe uma função *init2*. Ver Init em 4.1.

2.3.2 Load

Esta query possui a maior carga de trabalho do programa. É chamada a função parse e consequentemente o carregamento de dados para a estrutura. Foi decisão do grupo amortizar o máximo possível em load, já que não nos foi fornecido o número de queries que seriam executadas ao correr o programa. Amortizando em load, dependemos apenas do número de snapshots passados como argumento, já que as queries são quase instantâneas. Numa análise assimptótica, o programa corre em $\theta(nsnaps, nqueries) = \theta(N, 1)$. Isto permite que um grande número de queries seja executada em tempo constante. Ver Load em 4.2.

2.3.3 All Articles

Dadas as nossas estrututas de dados, para calcular o número total de artigos basta devolver o tamanho da nossa hashtable(*qs-¿hashsize*. (Retornamos metade da hashsize, porque no parse aumentamos o tamanho da hashtable para o dobro, por segurança.) Ver All_Articles em 4.3

2.3.4 Unique Articles

Para os artigos únicos, nós temos um inteiro na nossa estrutura de dados que nos devolve o número de posições da hashtable dos artigos ocupadas, ou seja, o número de artigos únicos. Ver Unique_Articles em 4.4.

2.3.5 All Revisions

Na estrutura dos artigos, temos um inteiro para cada aritgo que nos diz o número de revisiões que cada um tem. Para calcular o número total de revisões, percorremos todos os artigos e somamos o *nrevisions* de cada artigo. Ver All_Revisions em 4.5.

2.3.6 Top 10 Contributors

Para executar esta query, criámos um array(além do final), de 10 posições, para guardar as dez maiores contribuições. Este array é-nos muito útil, porque à medida que vamos percorrendo a hashtable dos contribuidores, comparamos o *ncontributions* de cada contribuidor com o número de contribuições presentes em cada posição no array, inserindo na posição que satisfaça a condição de ordenação e inserção, aplicada pela função auxiliar *inserts*.

Fazendo isto para toda a estrutura dos contribuidores, executámos com sucesso a query pretendida. Ver Top_10_Contributors em 4.6.

2.3.7 Contributor Name

Fazendo o hash do id dado, isto é, calculando o resto da divisão do id pelo *hashsize*, o resultado é a posição da hashtable dos contribuidores em que pode estar o id pretendido. Caso não encontre, procuramos na próxima e assim sucessivamente até encontrar o id. Uma vez encontrado, guardamos o nome do contribuidor na variável *name*(encapsulamento), e retornamos esta mesma. Ver Contributor_Name em 4.7.

2.3.8 Top 20 Largest Articles

Nesta query criámos um array auxiliar (além do final com os id's) com 20 posições, para guardar o número de carateres. O passo seguinte foi percorrer toda a hashtable, e guardar o número de carateres de cada artigo, mas na sua versão mais recente, isto é, na *revision* mais recente. Uma vez guardado o número de carateres atualizado, fomos comparando com os números presentes, e quando o nosso guardado *nc*(número de carateres) fosse maior do que aquele no

array, ou se o nc fosse igual mas o correspondente id fosse menor do que aquele no array final, inseriámos, invocando a nossa função auxiliar *insertArticleId*, o id na posição em que a comparação deu true. Depois disto, inserimos também o nc na mesma posição mas no array dos números de carateres.

O nosso método de inserção e ordenação consistia em começar na última posição do array, e ir colocando o valor anterior na posição em que estamos, por exemplo, arr[9] = arr[8], arr[8] = arr[7], e assim sucessivamente, até chegarmos à posição em que o id tinha que ser inserido. Quando atingida a posição, inseríamo-lo e terminava a função.

Realizando isto para todos os artigos, obtivemos os 20 maiores.

Ver Top_20_Largest_Articles em 4.8.

2.3.9 Article Title

Nesta query, dado o id do artigo e as nossas estruturas de dados, o trabalho ficou facilitado. Fazendo o hash desse id, obtemos a posição da hashtable dos artigos, a partir da qual começámos a procurar esse id. Quando encontrado, o passo seguinte foi aceder à revisão mais recente, para consequentemente, obter o título mais recente. Para isto, e para garantir encapsulamento, criámos uma variável *timestamp*, onde guardamos a data de uma revisão, e outra chamada *title*, para guardar o título dessa mesma revisão. Através de um ciclo, percorremos todas as revisões do artigo e guardámos a data e o título mais atuais, chegando assim ao resultado esperado. Ver Article_Title em 4.9.

2.3.10 Top N Articles With More Words

Assim como realizámos as queries top_10_contributors e top_20_largest_articles, do mesmo modo realizamos esta. Comparando com esta segunda, as diferenças estão apenas no tamanho dos arrays *top_N* e *numberOfWords*, que são de N posições os dois; a variável para guardar o número de palavras de cada artigo chama-se *nw*, em vez de nc. O percorrer da hashtable, a atualização do nw, a inserção e ordenação dos arrays é feita exatamente da mesma forma que as duas queries acima mencionadas. Ver Top_N_Articles_With_More_Words em 4.10.

2.3.11 Titles With Prefix

Na resolução desta query, como não sabemos quantos títulos têm o prefixo, criámos o array *final* com tamanho 1, e sempre que fossemos encontrando títulos para inserir, incrementávamos o tamanho(*size*). O passo seguinte foi, logicamente, percorrer a nossa hashtable e verificar para cada artigo, se o correspondente título possuía o dado prefixo, considerando a atualização das revisões. Já com os dados atualizados, verificámos se o *prefix* era mesmo um prefixo do título, e se fosse, incrementávamos o tamanho do array, inseríamos o título, e incrementávamos também o *inserted*(esta última variável dá-nos o número de títulos inseridos). A razão de primeiramente incrementar o tamanho e só depois inserir o título é porque, desta forma, temos sempre a última posição do array em branco, que servirá para inserir o *NULL*.

Por último, e executando tudo o descrito acima para todos os artigos, faltava apenas ordenar o array. Para isto, usámos a função já pré-definida *qsort*, devolvendo o o array final bem sucedido. Ver Titles_With_Prefix em 4.11.

2.3.12 Article Timestamp

O primeiro passo a realizar foi calcular o *hash* do id do artigo passado como argumento. O resultado deste cálculo dá-nos a posição da hashtable a partir do qual começámos a procurar o id, podendo não encontrar logo à primeira, devido às colisões. Seguidamente, encontrado o id, acedemos à nossa lista ligada das revisions, e percorremos a lista até encontrar o id pedido. Quando encontrado, temos uma variável *timestamp* que guarda a data dessa revision, e retornamo-la, garantindo encapsulamento.Ver Article_Timestamp em 4.12.

2.3.13 Clean

A gestão de memória foi sempre uma preocupação do grupo, causando com que procurássemos uma ferramenta para a verificação de memory leaks e se efetivamente havia perda de memória ao finalizar a execução do programa. A ferramenta que mais objetivamente nos fornecia a informação que precisávamos é o *valgrind*. Esta ferramenta com o comando *"valgrind –leak-check=yes ./program args"* fornece o sumário da heap à saída do programa.

A *clean* em si percorre ambas *hashtables* oa mesmo tempo, já que possuem o mesmo tamanho. Para cada artigo, se está ocupado, libertam-se as revisões uma a uma. No fim libertam-se ambas *hashtables* e a TCD_istruct em si. Ver Clean. 4.13.

3 Conclusão

Em suma, é opinião do grupo que concluimos o projeto com sucesso. Através do parse dos ficheiros xml conseguimos extrair toda a informação relevante para o projeto, de uma maneira otimizada, passando esta para uma estrutura de dados pensada e feita por nós na totalidade. Tínhamos como objetivo criar uma estrutura simples mas eficaz, e observando os resultados, cremos que atingimos esse objetivo.

4 Anexos

```
void parse(int nsnaps, char **snaps_paths, TAD_istruct qs) {
  int i, size = 0;
  if (nsnaps <= 0) {
    printf("Usage: ./program docname\n");
    return;
  }

  xmlDocPtr doc[nsnaps];
#pragma omp parallel for num_threads(8) ordered schedule(dynamic)
  for(i=0; i<nsnaps; i++){
    doc[i] = xmlParseFile(snaps_paths[i]);
    size+=parseCount(doc[i]);
  }


  size = (size *2);
  qs->hashsize = size;
  qs->articles = malloc(size*sizeof(Article));
  qs->authors = malloc(size*sizeof(Contributor));
  qs = init2(qs);

  for(i=0; i<nsnaps; i++){
    parseDoc(doc[i], qs);
    xmlFreeDoc(doc[i]);
  }

  xmlCleanupParser();
}
```

4.1 Init

```
TAD_istruct init(){
  TAD_istruct qs = (TAD_istruct) malloc(sizeof(TCD_istruct));
  qs->occupied_articles = 0;
  qs->hashsize = -1;
  return qs;
}
```

4.2 Load

```
TAD_istruct load(TAD_istruct qs, int nsnaps, char* snaps_paths[]){
  parse(nsnaps, snaps_paths, qs);
  return qs;
}
```

4.3 All_Articles

```
long all_articles(TAD_istruct qs){
  return (long) (qs->hashsize)/2;
}
```

4.4 Unique_Articles

```
long unique_articles(TAD_istruct qs){
  return (long) qs->occupied_articles;//occupied_articles is the
      number of unique id's inside the hash table.
}
```

4.5 All_Revisions

```
long all_revisions(TAD_istruct qs){
  int i;
  long l=0;
  //runs through all the revisions inside the hash table and as
      long as doesn't finish, increase the l variable.
  for(i=0; i<qs->hashsize; i++) l += qs->articles[i].nrevisions;
  return l;
}
```

4.6 Top_10_Contributors

```
long* top_10_contributors(TAD_istruct qs){
  int i, flag;
  int contributions[10] = {-1};//creates an array with the top ten
      contributions.
  long *top_ten = malloc(10*sizeof(long));//creates the final array
      .
  for(i=0; i<qs->hashsize; i++){
    flag = 0;
    if(qs->authors[i].isOccupied){
      inserts(qs->authors[i].id, qs->authors[i].ncontributions,
          contributions, top_ten, 10, flag);
    }
  }
  return top_ten;
}
```

4.7 Contributor_Name

```
char* contributor_name(long contributor_id, TAD_istruct qs){
  int i = hash(contributor_id, qs);
  int found = 0;
  char *name;//final pointer to the contributor name.
  while(qs->authors[i].isOccupied && !found){//runs through all the
       author's block inside the hash table.
      if(qs->authors[i].id == contributor_id) {//check if the given
           id equals to the present idcontributor.
        found = 1;
        name = strdup(qs->authors[i].name);//saves the author's
            name.
      }
      i++;
  }
  if(!found) return NULL;
  return name;
}
```

4.8 Top_20_Largest_Articles

```
long* top_20_largest_articles(TAD_istruct qs){
  long *top_twenty = malloc(20*sizeof(long));
  int i, flag, nc;
  int numberOfChars[20] = {-1};
  RevList* aux;
  for(i=0; i<qs->hashsize; i++){
    flag = 0;
    if(qs->articles[i].revisions){
      nc = qs->articles[i].revisions->nchars;
      aux = qs->articles[i].revisions->next;
      while(aux){
        if(nc < aux->nchars) nc = aux->nchars;
        aux = aux->next;
      }
      inserts(qs->articles[i].id, nc, numberOfChars, top_twenty,
          20, flag);
    }
  }
  return top_twenty;
}
```

4.9 Article_Title

```
char* article_title(long article_id, TAD_istruct qs){
  char *title, *timestamp;
  int i = hash(article_id, qs), flag = 0;
  RevList *aux;
  while (qs->articles[i].isOccupied && !flag){//runs through all
      the author's block inside the hash table.
    if(qs->articles[i].id == article_id){//checks if the given id
        equals to the present article's id.
      flag = 1;
      timestamp = qs->articles[i].revisions->date;//saves the date.
      aux = qs->articles[i].revisions;
      title = strdup(mostRecentTitle(aux, timestamp, title));
    }
    i++;
  }
  return title;
}
```

4.10 Top_N_Articles_With_More_Words

```
long* top_N_articles_with_more_words(int n, TAD_istruct qs){
  long* top_N = malloc(n*sizeof(long));
  int i, l, flag, nw;
  int numberOfWords[n];
  RevList *aux;
  for(l=0; l<n; l++) numberOfWords[l] = -1;
  for(l=0; l<n; l++) top_N[l] = 9999999;
  for(i=0; i<qs->hashsize; i++){
    flag=0;
    if(qs->articles[i].revisions){
      nw = qs->articles[i].revisions->nwords;
      aux = qs->articles[i].revisions->next;
      while(aux){
        if(nw < aux->nwords) nw = aux->nwords;
        aux = aux->next;
      }
      inserts(qs->articles[i].id, nw, numberOfWords, top_N, n, flag
          );
  }
}
  return top_N;
}
```

4.11 Titles_With_Prefix

```
char** titles_with_prefix(char* prefix, TAD_istruct qs){
  int i, inserted=0, size = 1, length = strlen(prefix);
  char** final = (char**) malloc(size*sizeof(char*));//the final
      pointer that the function will return.
  char *timestamp, *title;//timestamp will point to the date, title
       to the article's title.
  RevList* aux;
  for(i=0; i<qs->hashsize; i++){
    if(qs->articles[i].revisions){//it only checks the title and
        inserts it if there are revisions.
      timestamp = qs->articles[i].revisions->date;
      aux = qs->articles[i].revisions;
      title = mostRecentTitle(aux, timestamp, title);//'title'
          saves the most recent title;
      if(!strncmp(prefix, title, length)){
        final = realloc(final, size*sizeof(char*)+sizeof(char*));//
            increases the final's size.
        size = size + 1;
        final[inserted++] = strdup(title);
      }
    }
  }
  final[inserted] = NULL;
  for(i=0;final[i];i++);
  qsort(final, i, sizeof(char*), cstring_cmp);
  return final;
}
```

4.12 Article_Timestamp

```
char* article_timestamp(long article_id, long revision_id,
    TAD_istruct qs){
  int i, flag=0;//'i' variable is to run through the hash table.
      The flag is to get out of the for cycle without a break.
  char *timestamp;
  RevList *aux;
  i = hash(article_id, qs);
  while(qs->articles[i].isOccupied && !flag){
    if(article_id == qs->articles[i].id){//first we search for the
        article's id.
      aux = qs->articles[i].revisions;
      for(; aux && !flag; aux = aux->next){
        if(revision_id == aux->id){//searches for the revision's id
            . If it finds it, then we point the timestamp to the
            date.
          timestamp = strdup(aux->date);
          flag = 1;
        }
      }
    }
    i++;
  }
  return timestamp;
}
```

4.13 Clean

```
TAD_istruct clean(TAD_istruct qs){
  int i;
  for(i=0; i< qs->hashsize; i++){
    if(qs->articles[i].revisions)
      freeREV(qs->articles[i].revisions);
  }
  free(qs->articles);
  free(qs->authors);
  free(qs);
  return qs;
}
```