Instituto Superior Técnico

Mestrado em Engenharia Eletrotécnica e de Computadores

Algoritmos e Estruturas de Dados

2016/17 – 2º Ano, 1º Semestre

Relatório do Projeto

Word Morph

Grupo nº 48:

Manuel Carvalho:
Nº 84127, email: manuel.carvalho@tecnico.ulisboa.pt
José Silva:
Nº 84109, email: j.ferreira.da.silva@tecnico.ulisboa.pt

Docente: **Carlos Bispo**

Índice

Conteúdo

Descrição do Problema

O problema a ser resolvido neste projeto era, obtendo 2 palavras, uma inicial e outra final, encontrar o caminho de menor custo que levasse a palavra inicial a transformar-se na palavra final.
Partindo da palavra inicial, poderiam alterar-se X caracteres para a transformar numa outra palavra (onde X seria um numero que variava de caso para caso), desde que a palavra gerada existisse no dicionário.
Assim, de transformação em transformação chegar-se-ia à palavra final, formando, portanto, uma sequência de palavras que seria o "caminho" a encontrar.
Contudo, entre 2 palavras poderão haver mais do que 1 caminho possível, pelo que o objetivo deste problema era o de encontrar o caminho com o menor Custo associado.
O Custo associado a um caminho calcula-se ao fazer X^2 (em que X, como em cima, seria o nº de carateres alterados por transformação) a multiplicar pelo numero de transformações (logo, quantas mais fossem utilizadas maior seria o custo).

Abordagem do Problema

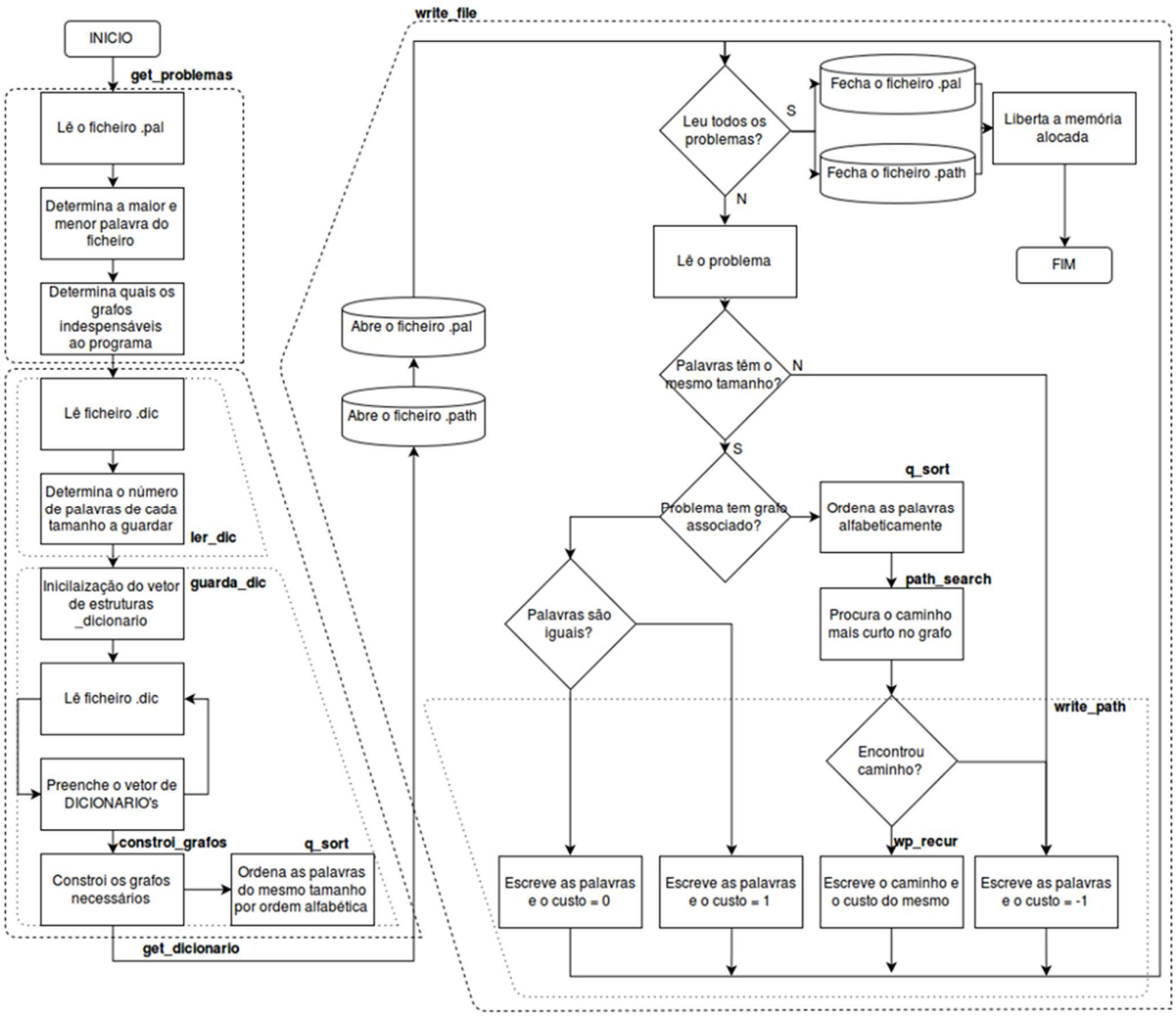
Para abordar e resolver o problema, houve várias etapas extremamente importantes.
Primeiro que tudo, teve de ser definida a maneira como iria ser utilizada a memória, isto é, o que é necessário guardar, o que só precisa de ser guardado temporariamente, o que não precisa de todo ser guardado e naquilo que precisa de ser guardado, como o é feito (vectores, listas, filas, …). Alguns exemplos destas decisões:

- Guardar apenas a parte útil do dicionário fornecido, sobre a forma de um vetor de estruturas;
- Guardar temporariamente quais os grafos a construir, sobre a forma de uma lista simplesmente ligada;
- Guardar as adjacências dos grafos, sobre a forma de listas de adjacências;
- Não guardar os problemas lidos do ficheiro.

Estes tipos de decisões foram feitas de modo a maximizar a eficiência do programa, sem exceder os limites de memória permitidos, ou seja, optou-se sempre que possível por estruturas em que as operações mais frequentes fossem de menor complexidade possível (O(1) sempre que possível). No exemplo das adjacências do grafo, seria mais eficiente fazê-lo na forma de matriz de adjacências, no entanto não existe memória disponível para o fazer e, portanto, optou-se pela forma de listas de adjacências. No caso do dicionário, este é guardado sobre a forma de vetor de estruturas, possibilitando o acesso direito ao mesmo, sendo que as palavras guardadas estarão também sobre a forma de vetor, de forma a que se possa realizar procura no mesmo, em tempo logarítmico de base 2, através da procura binária.
Com isto, obtém-se como os diferentes elementos necessários ao programa irão estar organizados. De seguida, terá de ser decidido o que é que o programa irá fazer no geral. Como ficou decidido anteriormente que não irão ser guardados os problemas fornecidos e que só irão ser guardadas as palavras necessárias para a resolução dos problemas, teremos no inicio do programa de ler o ficheiro *.pal à procura da menor palavra existente neste, juntamente com a maior e retornar estes valores ao programa. Além disso, como a construção de grafos é um processo muito dispendioso em termos de tempo e memória, só irão ser construídos os grafos que sejam absolutamente necessários para os problemas, ou seja, só vale a pena construir 1 grafo por tamanho de palavras, sendo o número de mutações máximas permitidas deste, o maior valor lido dos problemas de um determinado tamanho de palavras, se as palavras do problema diferirem mais do que um carácter entre elas.
Cada tamanho de palavra que justificar a existência de grafo, terá um e um só grafo, sendo que este

estará representado por lista de adjacências em que cada nó da lista terá o 2º vértice da ligação sob a forma de inteiro e o peso da ligação, em que o inteiro representa a posição da palavra no vetor de palavras do mesmo tamanho, simplificando assim as operações referentes a grafos.
De seguida, teremos de decidir como iremos procurar o caminho ou a existência deste no grafo. Para tal, compararam-se vários algoritmos de procura, tais como o algoritmo de Dijkstra, A*, Floyd, Procura em profundidade, etc... e ficou decidido optar pelo algoritmo de Dijkstra pois dos analisados era o que apresentava uma complexidade computacional melhor por complexidade de implementação que os restantes. Para o implementar, foi necessária uma fila prioritária, tendo também várias hipóteses para estas (heaps binárias, de fibonacci, binomiais,...) e acabou por ser usada uma binária, pelas mesmas razões da escolha algoritmo de Dijkstra.
A prioridade na fila é dada pelo custo do caminho do vértice índice, ao vértice origem, presente num vector wt[], sendo que o valor deste é alterado ao longo da execução do algoritmo e, portanto, a posição de um dado elemento também irá mudar. No entanto, para mudar a ordem de um elemento era necessário saber a posição deste elemento no vetor da heap, e para evitar realizar procuras nesta, criou-se um vetor auxiliar com a posição de cada elemento na heap.
Por fim, o algoritmo de Dijkstra retorna um caminho ao contrário, ou seja, retorna o caminho do vértice destino, ao vértice de origem, no entanto o ficheiro de saída requer que este seja escrito do vértice de origem para o vértice destino, e para o fazer optou-se por uma implementação recursiva, sendo que esta percorre o caminho até ao fim, e depois escreve-o de trás para a frente.

Descrição Completa do Programa



Tipos de Dados Utilizados

No nosso programa, foram utilizados diversos tipos de dados:
Foram utilizadas listas simplesmente ligadas (FIFO), implementadas desta forma:

Lista (struct _lista)	Tamanho: Depende do caso
Item * this;	Ponteiro para o conteúdo do presente nó da lista.
Struct _lista *next;	Ponteiro para o próximo elemento da lista.

Caso 1:
Construímos também uma lista de grafos a construir, ou seja, à medida que o programa vai lendo o ficheiro de problemas, vai inserindo numa lista o grafo respetivo ao tamanho de palavra do problema em questão, formando assim uma lista com uma quantidade de grafos igual ao numero total de problemas.

Lista de Grafos a construir (struct _grafo_fila)	Tamanho: nº de grafos necessários para a resolução do problema. (*)
Int size;	Tamanho das palavras nos vértices
Int mut;	Nº de mutações que se podem fazer

Caso 2:
A estrutura _palavra serve para guardar temporariamente uma palavra, juntamente com o seu tamanho, lida do ficheiro .dic , que irá fazer parte de uma das estruturas _dicionario.

Lista de Palavras Guardadas (struct _palavra)	Tamanho: nº de palavras de tamanho menor que n_max e maior que n_min presentes no ficheiro .dic
Int size;	Tamanho da palavra;
char* word;	Palavra em si;

Foram criadas várias estruturas do tipo DICIONARIO, organizadas num vetor, em que cada uma delas corresponde às palavras de um certo tamanho e está organizada da seguinte forma:

Vetor de Dicionários (struct _dicionario)	Tamanho: n_max – n_min
Int size;	Número de palavras desse tamanho;
Char **words	Vetor com todas as strings desse tamanho;
GRAFO * graph	Ponteiro para o grafo correspondente às palavras desse tamanho;

Utilizou-se, também, uma tabela para representar o caminho encontrado no final do programa, gerando assim uma tabela com as várias palavras da sequência.
Utilizou-se uma fila para os vários problemas a resolver, ou seja, assim que se iam lendo os problemas do ficheiro, iam-se inserindo estes numa fila.
Utilizaram-se, também, grafos formados por lista de adjacência, em que os vértices eram todas as palavras de um só tamanho. Estes grafos encontram-se no dicionário referente a esse tamanho de palavra e servem para encontrar o caminho de menor custo.
Esta é a descrição de um vértice e a sua respetiva lista, onde cada elemento guarda a informação de uma das ligações do vértice:

Vértice (struct _vert)	Tamanho: nº de ligações do grafo.
Int w;	2º vértice da aresta
Int peso;	Peso da Aresta
Struct_vert *next;	Ponteiro para a próxima aresta deste vértice

Esta é a estrutura de cada um dos grafos existentes, um por cada problema que for necessário resolver:

Grafo (struct _grafo)	Tamanho: nº de grafos necessários para a resolução do problema. (*)
Int v;	Nº de vértices do grafo (=nº de palavras do respetivo tamanho)
Int mut;	Nº de mutações que se podem fazer
VERT ** adj;	Vetor com as listas de adjacências dos vários vértices

Para encontrar o caminho, criou-se uma fila prioritária da forma de um acervo (heap), necessária para a aplicação do algoritmo de Dijkstra, estruturada desta forma:

Fila Prioritária (struct _fila_p)	Tamanho: 1
Int livre;	Índice da 1ª posição vazia da heap (= nº de elementos na heap)
Int size;	Nº máximo de posições na heap
Int *fila;	Vetor de inteiros com o valor de cada posição da heap
Int * pos;	Vetor de inteiros que indica a posição que cada elemento ocupa na heap. (Ex: fila[pos[A]] = A)

(*)Para palavras iguais e palavras que só diferem 1 caracter uma da outra, não é necessário construir grafo.

Algoritmos Utilizados

Os principais algoritmos utilizados nos cálculos foram os que se encontram nas funções write_path, wp_recur, constroi_grafo, det_peso, insrt_grafo e get_problemas.

Wp_recur :

A escrita do caminho, encontrado pelo algoritmo de Dijkstra, é um algoritmo ao qual demos o nome de wp_recur e cuja apresentação em fluxograma se encontra à direita.
No início de cada ciclo deste algoritmo, é verificado se o caminho já acabou de ser escrito, ao verificar se st[u], ou seja, o valor do caminho respetivo à palavra que se vai escrever, é igual a -1. Não sendo igual a -1, escreve-se a palavra words[u] no ficheiro.
Este ciclo é realizado de forma contínua (até que st[u] seja igual a -1) e num sentido contrário ao do vetor caminho, ou seja, começa-se por escrever a palavra "destino" e vai-se seguindo o caminho a partir daí. Isto foi necessário, pois o caminho que o algoritmo de Dijkstra fornece é um caminho que liga o vértice destino ao vértice origem, e o que se quer neste caso é o contrário. Portanto, inverteu-se o caminho do vetor st, de forma a escrevê-lo no ficheiro pela ordem correta. No fim deste algoritmo, escreve-se a palavra de origem (última a ser escrita pela razão do inverso, já mencionada) e o custo associado a este caminho (path).

Det_Peso :

O algoritmo **det_peso** serve para calcular o peso de 2 palavras do mesmo tamanho, ou seja, o número de caracteres em que uma difere da outra.
Este algoritmo utiliza uma variável i, que é incrementada em todos os ciclos, facilitando, assim, a comparação caracter a caracter, pois, como sabemos, uma string funciona como se fosse um vetor de char's (caracteres). Assim, é comparado o 1º caracter de cada palavra, aumentando a variável dif (diferenças), apenas se estes se verificarem diferentes. No ciclo seguinte, é comparado o 2º caracter de cada palavra (se existir), e assim sucessivamente até ao fim das palavras.
Em todos os ciclos, é verificado se já se chegou ao fim da palavra (i = strlen(pal2)). Se for o caso, o algoritmo termina e **retorna o valor de dif**, que será o número de caracteres pretendido.
É também verificado todos os ciclos se a variável dif é maior que mutmax, ou seja, se o número de caracteres diferentes é maior do que as mutações máximas que podem ser feitas. Se for o caso, mais uma vez o algoritmo termina, com a diferença de que **retorna o valor de -1**.

Constroi_grafo :

A construção de cada um dos grafos associados às estruturas _dicionario é realizada a partir do algoritmo **constroi_grafo**, cujo esquema em fluxograma se encontra em cima.
Este algoritmo irá ser utilizado tantas vezes quantos grafos forem necessários construir para resolver o problema. Os grafos a criar irão ter um vértice por cada palavra diferente, do tamanho de palavra associado à estrutura, como já explicado anteriormente. Assim, cada vértice será uma palavra diferente, mas do mesmo tamanho de todas as outras.

A chave deste algoritmo assenta na função insrt_grafo, que insere um vértice num grafo por lista de adjacências, acrescentando dois vértices a duas listas, à lista do vértice i do grafo e à lista do vertice j do grafo.

Este dois índices, i e j, são referentes às duas palavras que foram testadas para se saber as diferenças entre elas. Visto que, caso duas palavras sejam adjacentes, irão ser inseridas simultaneamente nas listas de adjacências uma da outra, não será necessário comparar um palavra com todas as outras, mas sim uma palavra com todas as outras que não foram testadas já. Assim realiza-se um ciclo que começa com a variável i a 0, e esta variável irá tomar os valores das posição de todas as palavras que pertencem ao vector de palavras com o mesmo tamanho, até chegar à ultima posição do mesmo. Dentro deste ciclo irá ser feito um outro ciclo em que a variável j será inicializada a j=i+1 e a variável tomará tal como i, todos as posições do vector entre j e o a ultima posição do vector. Dentro do ciclo j, será feita a a comparação entre as palavras dos dois índices da tabela, i e j e caso a diferença entre estas seja menor que as mutações máximas permitidas, a adjacencia i:j inserida no grafo.

Desta forma dois pares de palavras são apenas testados 1 fez da seguinte forma:
0:1 , 0:2, 0:3 ,... ; 1:2, 2:2, 2:3,... ; 3:4 , 3:5, 3:6 …. e por ai em diante

Com isto a complexidade desta função, assumindo que a complexidade de det_peso é O(1), a complexidade do constroi_grafo é de O(N!).

Get_problemas :

A função get_problemas é uma função muito importante para o funcionamento correto, e eficiente do programa visto que esta decide a quantidade de memória a ser ocupada pelo dicionário, assim como quais grafos que o programa tem de construir.

Como dito anteriormente apenas são guardadas as palavras necessárias para a resolução dos problemas, para tal existiriam duas opções, guardar todas as palavras de tamanho igual às lidas do ficheiro, ou guardar todas as palavras de tamanho maior ou igual a n_min, e menor ou igual a n_max, sendo que estes dois valores representam os tamanhos da menor e maior palavra lida do ficheiro de problemas. No primeiro caso iria ser usada uma menor quantidade de memória, no entanto arranjar uma fórma de a aceder depois de guardada seria uma tarefa difícil e provavelmente não teria um custo unitário. Na segunda hipótese irão ser guardadas mais palavras mas depois de guardadas, a estrutura onde estas se encontram pode ser acedida directamente, usando um vector de estruturas _dicionario, de tamanho n_max-n_min, em que cada posição do vector contem a estrutura referente a palavras de tamanho índice+n_min. Sendo esta a opção mais viável, esta função determina o menor e o menor tamanho de palavras a ser guardado.

Alem disso, esta função também determina quais os grafos que o programa tem obrigatoriamente de construir para o seu correto funcionamento. Sendo que um caminho entre palavras apenas pode conter palavras do mesmo tamanho, cada grafo será referente a um dado tamanho de palavras e portanto só será necessário construir um para cada tamanho. No entanto cada ligação do grafo tem um peso associado, e quanto maior este for maior será grafo e mais lento e mais pesado será o programa, portanto terá de ser escolhido um peso máximo por ligação no grafo de forma a este calcular correctamente os caminhos, usando o mínimo de recursos possível. Para que o programa funcione correctamente, o peso máximo por ligação será o maior identificador do problema lido do ficheiro, sendo que problemas que tenham este inferior na resolução dos mesmo são ignoradas arestas do grafo que tenham peso superior ao permitido. No entanto não será necessário criar grafos para palavras que diferem apenas um carácter, ou até mesmo nenhum, visto que o caminho pode ser dado directamente. Assim só serão construídos grafos para palavras que tenham tamanho igual aos lidos do ficheiro *.pal com o máximo de mutações permitidas, o maior valor lido do ficheiro se as palavras referentes ao problema difiram em mais que 1 carácter entre elas.

Para saber que grafos é que o programa deve construir, a função guarda num vector com 28 posições(maior palavra existente no dicionário), um inteiro que se refere ao número máximo de mutações entre palavras num grafo, sendo que o tamanho das palavras no grafo irá ser dado pelo índice da posição do inteiro desse vector . Depois a função transforma esse vector numa lista de estruturas _grafo_fila, que contem as propriedades acima descritas.

Descrição dos Subsistemas

O programa implementado possui 4 subsistemas diferentes.
O subsistema **lista** (constituído por lista.c e lista.h), que contém as funções que estão ligadas à manipulação de listas.
O subsistema **grafo** (constituído por grafo.c e grafo.h) tem as funções relacionadas com a manipulação dos grafos, cujas estruturas estão definidas no ficheiro grafo.h.
O subsistema **fila_p** (constituído por fila_p.c e fila_p.h) contém as funções relacionadas com a heap (fila prioritária), que está definida como estrutura no ficheiro fila_p.h.
Por fim, o subsistema **dicionario** (constituído por dicionário.c e dicionário.h), está responsável pelas funções relacionadas com a manipulação das palavras no dicionário.
Existe ainda o ficheiro **const_estru.c** , que contém a definição de algumas funções básicas que vao ser usadas por todo o programa. (ex: less(A, B)) e o ficheiro **main.c**, responsável pela ligação de todos os subsistemas do programa e onde também se encontram funções relativas à leitura do ficheiro de problemas e à escrita do ficheiro de saída, juntamente com a função main, que é usada apenas para chamar as funções necessárias para o funcionamento do programa.

Lista.c :

`LISTA * isrt lista(LISTA * head, Item this)` : insere novo nó no início da lista
`LISTA * isrt ord lista(LISTA ** head, LISTA * tail, Item this)`: insere novo nó no final da lista
`Item getItem lista(LISTA * node)`: retorna item do presente nó da lista
`LISTA * getNext lista(LISTA * node)` : retorna ponteiro para o próximo nó da lista
`void free lista(LISTA * head, void (* freeItem)(Item))`: Liberta toda a memória alocada para a lista

Grafo.c :

`GRAFO * init grafo(int v): Inicializa um grafo com v vértices- aloca a memória necessária, inicializa os vértices e retorna o grafo.`

`VERT * new vert(VERT * head, int w, int peso ): Insere um novo vértice na lista de adjacência do vértice w, com uma ligação de peso "peso".`

`void insrt grafo(GRAFO * G, int v1, int v2, int peso) : insere uma adjacência no grafo G entre o vértice v1 e o vértice v2, em que o inteiro peso é o peso da ligação.`

`int *path search(GRAFO *G, int s, int dest , int mut, int * st, int *path) : Procura o caminho mais curto entre 2 vértices(s e dest), com as mutações possíveis = mut, de um grafo e retorna o vetor de inteiros st com esse caminho. Calcula também o custo do caminho (path), que é passado por referência.`

Fila_p.c:

int **get pos**(FILA_P * fila, int w): Retorna a posição do vértice w na fila prioritária "fila".

FILA_P * **PQinit**(int size) : Aloca a memória necessária e inicializa uma fila prioritária de tamanho size.

void **free fila**(FILA_P * fila) : Liberta a memória alocada para a fila prioritária "fila".

Para as funções deste subsistema que se seguem, a entrada wt é um vetor que contém os custos de todos os caminhos e a partir dele que se calcula as prioridades (prioridade inversa ao valor de wt.

FILA_P * **FixUp**(FILA_P * fila, int Idx, int *wt): Altera a posição do vértice com a posição Idx na fila prioritária "fila" com a do vértice acima de si("pai"), se a prioridade do vértice acima for menor.
FILA_P * **FixDown**(FILA_P * fila, int Idx, int * wt): Altera a posição do vértice com a posição Idx na fila com a do vértice abaixo de si ("filho"), se a prioridade do vértice abaixo for maior.

FILA_P * **Modify**(FILA_P * fila, int Idx, int *wt) : Muda a ordem da fila prioritária consoante as mudanças de prioridades. (só faz fixUp pois as prioridades apenas aumentam, nunca diminuem)

int **PQdelmin**(FILA_P * fila, int*wt): Remove da fila prioritária "fila" o elemento com maior prioridade.

int **PQempty**(FILA_P * fila) : Verifica se a fila prioritária "fila" se encontra vazia, retornando 1 se estiver e 0 se não.

FILA_P * **PQinsert**(FILA_P * fila, int v, int * wt) : Insere o vértice v na fila prioritária "fila", de forma ordenada, tendo em conta a sua prioridade.

Dicionario.c :

Neste subsistema é usado variadas vezes como entrada, um inteiro n_min, que corresponde ao menor tamanho de palavras do vetor de estruturas DICIONARIO. Este é necessário em bastantes funções, pois, no nosso programa, não é guardado todo o dicionário, mas sim apenas a parte deste que é precisa para resolver os problemas, desde a palavra mais pequena até à maior palavra lida do ficheiro de problemas. Portanto, para aceder ao vetor de DICIONARIO, acede-se ao índice tamanho da palavra - n_min, pois este corresponde à primeira posição da parte do vetor que foi guardada. O índice máximo do vetor é n_max.

void **swap_str**(char **arg1, char **arg2) : Troca a ordem de 2 strings num vetor, onde arg1 e arg2 são ponteiros para a posição dessas 2 strings.

int **string_cmp**(const void *p1, const void* p2): Compara 2 string p1 e p2, retornando 1 se p2 for a maior e 0 se não for.

void **FixDown_char**(char** words, int Idx, int size) :

void **Heapify_char**(char ** words, int size) :

void **HeapSort**(char ** words, int size) :

int **procura_bin**(char** a, int ini, int fim, char * pal) : Procura a palavra pal no vetor de palavras "a" pelo método da procura binária, retornando a posição da mesma. Os inteiros ini e fim são os índices do início e fim desse mesmo vetor. Se a função não encontrar a palavra no vetor, retorna -1.

int **get_size**(DICIONARIO * dicionario, char* pal1, char* pal2, int n_min): Retorna o numero de palavras existentes no dicionário com o mesmo tamanho das palavras pal1 e pal2, verificando, também, se estas têm o mesmo tamanho (se não tiverem retorna 0).

int **get_position**(DICIONARIO * dicionario, char * pal, int n_min): Retorna a posição da palavra pal no dicionário.

LISTA * **inserir_pal**(char * pal, LISTA * head) : Insere a palavra pal, lida do ficheiro, e insere-a numa lista que contém as palavras lidas, em que "head" é um ponteiro para o seu inicio.

LISTA * **ler_dic**(char * nome_dic, LISTA *head, int n_max, int n_min, int **ocorrencias) : Lê de um ficheiro um dicionário de nome "nome_dic" e insere-o por completo numa lista simplesmente ligada (da qual "head" é o ponteiro para o inicio), guardando o tamanho da maior palavra (n_max).

DICIONARIO * **aloca_dic**(int n_max, int n_min, int *ocorrencias) : Aloca a memória necessária para guardar um vetor de estruturas do tipo DICIONARIO.

void **free_palavra**(Item this) : Liberta a memoria usada para guardar uma estrutura do tipo Palavra.

DICIONARIO * **guarda_dic**(LISTA *head, int *ocorrencias, int n_max, int n_min) : Transforma a lista de palavras que foram lidas do ficheiro dicionário num vetor de estruturas do tipo DICIONARIO.

void **free_vert**(Item this) : Liberta a memória necessária para guardar uma estrutura do tipo vértice de um grafo.

void **free adj**(VERT * head) : Liberta a memoria necessária para guardar todas as adjacências de um vertice de um grafo.

void **free grafo**(DICIONARIO dicionario) : Liberta a memoria necessária para guardar a estrutura do tipo grafo do dicionário que quisermos.

void **free dic**(DICIONARIO * dicionario, int n_max, int n_min) : Liberta a memoria necessária para guardar todas as palavras lidas do vetor de DICIONARIOs e os grafos correspondentes.

int **det peso**(char * pal1, char * pal2, int mutmax) : Determina o numero de caracteres em que a palavra pal1 difere da palavra pal2 e retorna-o. Se for maior que "mutmax" retorna -1.

GRAFO * **constroi grafo**(int mut, DICIONARIO dicionario) : Constrói um grafo com lista de adjacências no dicionário "dicionario", no qual o numero de diferenças entre 2 quaisquer palavras (vértices) terá de ser menor que "mutmax".

void **free grafo fila**(Item this) : Liberta a memória alocada para guardar as propriedades de um grafo a ser construído.

DICIONARIO * **constroi grafos**(DICIONARIO * dicionario, LISTA * grafo_fila, int n_min) : Depois da análise do ficheiro de problemas, constrói todos os grafos necessários para a resolução destes e guarda-os na respetiva estrutura DICIONARIO.

DICIONARIO * **get dicionario**(char* nome_dic, int n_max, int n_min, LISTA * grafo_fila) : Função geral deste subsistema, que chama todas as outras. Portanto, o que esta função faz é ler o ficheiro .dic, guardar as palavras necessárias aos problemas no vetor de DICIONARIO's e guardar todos os grafos na respetiva estrutura.

Main.c :

void **get problemas**(char* nome_prob, int * n_max, int * n_min, LISTA **grafo_fila) : Lê o ficheiro de problemas .pal de nome "nome_prob" e determina o tamanho da maior e da menor palavra, guardando-os em n_max e n_min, respetivamente, que são passados por referência. Determina também todos os grafos que têm de ser construídos (1 por cada tamanho de palavra), guardando as propriedades de cada um na fila grafo_fila (também passada por referência).

void **wp recur**(char ** words, int*st, int s,int u, FILE * fp, int path) :

Escreve o caminho final no ficheiro fp de vértice em vértice, onde s

é o vértice inicial e u é o vértice que estamos a escrever. Nesta função, o caminho é escrito de trás para a frente pois o algoritmo de Dijkstra retorna o caminho do vértice de destino para o de origem, e nós queremos que este seja escrito no ficheiro da forma contrária.

FILE * **write_path**(char ** words, int * st, int s,int dest, FILE * fp, int path) : Decide se escreve o caminho do vértice s até ao dest de forma recursiva (chamando wp_recur) ou de forma direta (se o custo – path – for igual a 0 ou 1).

void **write_file**(char* file_out, char * nome_prob, DICIONARIO * dicionario, int n_min) : Escreve um ficheiro de nome file_out com todas as soluções (caminhos) dos problemas lidos no ficheiro nome_prob.

int **main**(int argc, char * argv[]) : Função main do programa, que serve para chamar todas as outras funções e fazer correr o programa.

Requisitos do Programa

Sendo que ao implementar o programa teve-se o cuidado de tentar usar ao máximo funções cuja complexidade fosse O(1) poucos são os casos onde esta complexidade não é verificada. Nos casos em que esta complexidade não se verifica, as funções irão depender de vários parâmetros tais como o números de palavras no ficheiro .dic, o número de mutações em cada problema, o número de problemas, número de caracteres em cada palavra, … E como foi feito um grande número de funções, em que muitas delas dependem de outras desenvolvidas, iremos analisar os casos que mais influenciam o desempenho do programa, nomeadamente as operações sobre grafos.

O desempenho da construção de um grafo irá depender do número de palavras que pertencem ao grafo, ao tamanho destas, e ao número de mutações máximas permitidas entre palavras do grafo. Analisando o algoritmo de construção de grafos, facilmente se verificam estas condições, e verifica-se também que a complexidade deste algoritmo é O((Nº de palavras)!)*O(Tamanho das palavras). Sendo que este algoritmo é chamado sempre que é necessário construir um grafo, o programa irá usar no pior caso (Nº de grafos)*((Nº de palavras)!)*O(Tamanho das palavras) tempo para os construir a todos, em que o Nº de grafos irá depender da configuração do ficheiro de problemas, como referido anteriormente.

Para além da construção de grafos, a procura de caminho nestes é também uma operação que ocupa grande percentagem do tempo de execução do programa. Esta operação é feito conjuntamente pelas funções write_file, write_path e path_search sendo que a última realiza a procura em si e as outras duas realizam a escrita do caminho encontrado. Analisando a função path_search, sabendo que esta é referente ao algoritmo de Dijkstra usando uma heap binária como fila prioritaria, sabe-se então que este tem uma complexidade de O(E*log(V)) sendo que E é o número de ligações do grafo que será proporcional do número de palavras, do tamanho destas e das mutações máximas permitidas no grafo. Ora este algoritmo é usado sempre que se pretenda resolver um problema, em que este, para

além de ter um grafo associado, as duas palavras do mesmo terão de diferir em mais que um caracter e de ter o mesmo tamanho, sendo que o número de vezes que este algoritmo é chamado irá depender da configuração do ficheiro de entrada.

Desempenho do Programa e Análise Crítica

No inicio da implementação do programa, notaram-se várias falhas neste, sendo muitas delas graves, nomeadamente que o programa utilizava o dobro da memória máxima indicada pelos docentes da cadeira que cada ficheiro de teste devia utilizar. Para além disso, não era capaz de libertar completamente toda a memória alocada no decorrer do programa e ainda demorava muito mais que o tempo necessário para correr o programa, sendo que em alguns ficheiros de teste o programa corria durante mais de 10 minutos.

Dividindo o programa desenvolvido em duas partes, sendo que a primeira parte era responsável por ler os ficheiros de entrada e guardar os elementos necessários para a resolução dos problemas, e a resolução destes e a escrita dos seus resultados seria a função da segunda parte. Verificando que a primeira parte não levava mais do que 1 segundo a correr e não ocupava grande parte da memória em geral, ficou claro que os nossos problemas estavam na segunda parte da implementação do programa, em concreto na construção dos grafos e na procura do caminho mais curto nestes. Focando a atenção para estes dois elementos do programa, rapidamente se descobriu que, por exemplo, realizar procura linear para encontrar o índice de um certo elemento na fila prioritária não é uma boa ideia em termos de tempo de execução. Substituindo-a por um vetor com o índice de cada elemento resolve a questão. Também se descobriu que utilizar a interface de lista global usada em outras partes do código para representar a lista de adjacências de cada vértice do grafo ocupa mais memória que a necessária, resolvendo este problema ao criar-se uma lista "hardwired" para grafos.

Os dois exemplos acima descritos verificaram ser as principais falhas da implementação inicial do programa. Depois de resolvidas, como no geral, continuou a ter-se sempre em atenção qual o tipo de dados mais eficiente para cada operação, sem comprometer em demasia a memória utilizada. Não foram encontrados muitos mais casos críticos na execução do programa e resolvendo outras pequenas falhas existentes nas execuções seguintes, foram passados 19 testes ao submeter, sendo que dá a indicação de "Limite de tempo excedido" no teste que falta.

Tendo em conta que o código realizado tenta sempre aceder às estruturas criadas de forma direta e a escolha dos algoritmos a usar foi feita por comparação com outros algoritmos existentes, apresentado estes desempenhos semelhantes ou inferiores aos usados- Além disso, foi feito um esforço para se minimizarem as perdas de desempenho em operações simples, através da utilização de macros. Não fomos capazes de localizar com exatidão a origem deste problema, no entanto foi ponderada a hipótese de por exemplo o quicksort estar a falhar no teste em falta, tendo uma complexidade $O(N^2)$ em vez da esperada $O(NlogN)$, não tendo sido implementada por falta de tempo. Fora esta, não temos outra hipótese forte para a razão de falhar um teste, pode se estar a perder eficiência ao aceder às estruturas escondidas ao cliente, pode também ter sido usado um tipo de dados não tão adequado ao problema que nos passou ao lado, mas como referido anteriormente a falta de tempo derivada da resolução de outros problemas não permitiu um diagnóstico assertivo do problema.

De resto, o programa funciona como esperado, indicando os caminhos mais curtos entre palavras, não excede os limites de memória impostos, e não apresenta qualquer erro de compilação ou de execução libertando toda a memória utilizada durante a sua execução.