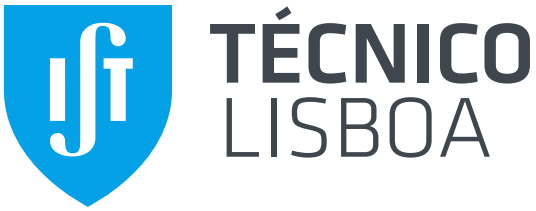


# ALGORITMOS E ESTRUTURAS DE DADOS

*Mestrado integrado em Engenharia Eletrotécnica e de computadores*

*2016/2017 - 2º Ano, 1º Semestre*

## Relatório do Projeto

## WORDMORPH


**Grupo Nº 93:**

**Diogo Emanuel Graça Rodrigues**

Nº 84030

E-mail: diogoegrodrigues@tecnico.ulisboa.pt

**Leandro José Pereira de Almeida**

Nº 84112

E-mail: leandro.pereira@tecnico.ulisboa.pt

Docente: **Carlos Bispo, Luís Silveira, Margarida Silveira e Paulo Flores**

# ÍNDICE

## DESCRIÇÃO DO PROBLEMA

O objetivo deste projeto é desenvolver um programa que seja capaz de produzir "caminhos" entre palavras. Um caminho entre duas palavras, do mesmo tamanho, dadas como ponto de partida e de chegada, é uma sequência de palavras de igual tamanho, em que cada palavra se obtém a partir da sua antecessora por substituição de um ou mais caracteres por outro(s). O número de caracteres máximo que se pode substituir entre 2 palavras é lido do ficheiro de texto de problemas, onde se lê também a palavra de partida e de chegada.

Para a resolução deste problema é necessária a existência de um dicionário (que é lido de um ficheiro de texto), pois todas as palavras pertencentes ao caminho têm de estar no mesmo.

Existe um custo associado à transformação de uma palavra noutra, por substituição de um ou mais dos seus carateres, para que se possa distinguir os vários caminhos possíveis desde da palavra de partida até à de chegada (caso haja caminho). No projeto em questão, assume-se que o custo é quadrático relativamente número de carateres substituídos de uma palavra para a outra. Ao se trocar x carateres de uma palavra para outra, tem-se um custo associado de $x^2$.

Assim, o objetivo do programa é que seja capaz de calcular o caminho de menor custo entre duas palavras e escreva um ficheiro de saída com os resultados de cada problema (o caminho entre as 2 palavras, caso exista, ou a indicação de que não há caminho para esse número máximo de caracteres permitido).

## ABORDAGEM AO PROBLEMA

Antes de se resolver cada problema, começou-se por guardar numa tabela todas as palavras do dicionário referentes aos tamanhos (comprimentos das palavras) dos problemas existentes e, ordená-las alfabeticamente. De seguida criamos um número de grafos igual ao número de tamanhos diferentes, em que cada grafo vai ser utilizado para resolver todos os problemas referentes a esse tamanho. Os grafos criados através de listas de adjacências são não direcionados, pois se a palavra A está ligada à palavra B, a palavra B também está ligada a palavra A.

Para percorrer o grafo de forma a achar o caminho mais curto entre a palavra de partida e de chegada, utilizou-se o algoritmo de Dijkstra, onde através da operação de relaxação do caminho na árvore de procura se verifica se existe um caminho mais curto do que algum dos conhecidos.

A pormenorização deste algoritmo, bem como de todos outros utilizados ao longo do programa, encontra-se especificada mais a frente.

## ARQUITETURA DO PROGRAMA

O Fluxograma que se segue mostra a arquitetura do programa Wordmorph e contém a informação principal acerca do seu funcionamento. Um estudo mais pormenorizado das estruturas de dados utilizadas, funções e funcionamento dos algoritmos é feito mais à frente no relatório.

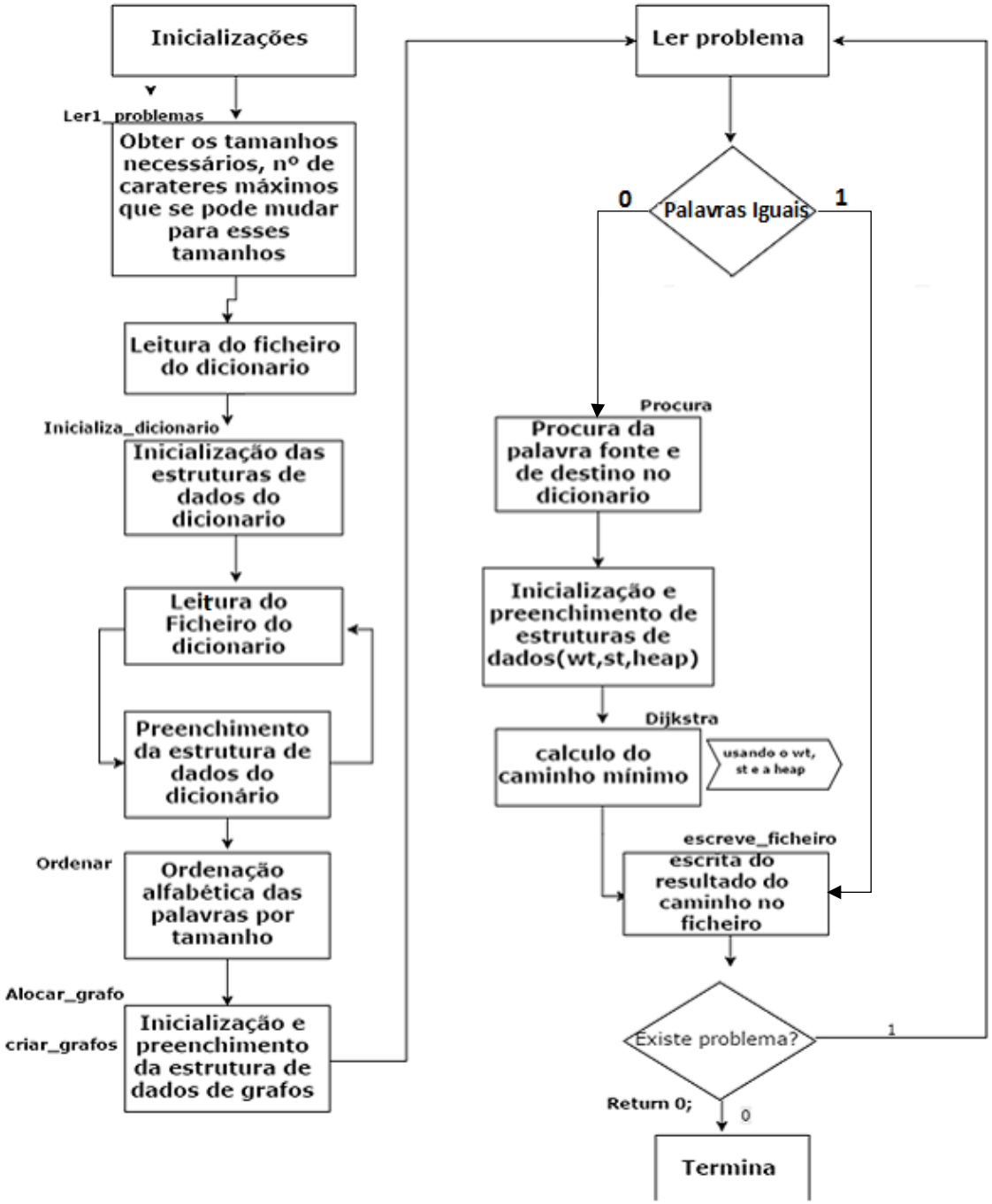



***Figura 1*** **–** Fluxograma do programa

## DESCRIÇÃO DAS ESTRUTURAS DE DADOS

Para guardar todas as informações relativas ao dicionário optou-se por usar uma estrutura denominada ***dicionario***. Constituída por:

| Campos | Descrição |
|---|---|
| Vetor de Inteiros: ***ocorrencias*** | Este vetor irá ter a dimensão do problema de maior tamanho existente, sendo que, será apenas incrementado para os problemas que nos interessarem levando a que desta forma contenha o número de palavras de cada tamanho. |
| Vetor de Inteiros: ***posicao_livre*** | Este vetor auxiliar irá também ter o tamanho do problema de maior tamanho existente, sendo que, este irá conter a primeira posição livre de cada tamanho que será utilizado para preencher a tabela tridimensional. |
| Tabela Tridimensional de carateres: ***palavras*** | Esta tabela tridimensional irá ser AxBxC, sendo que, A é da dimensão do problema de maior tamanho existente, B é da dimensão do número de palavras de cada tamanho e C é da dimensão do tamanho de cada palavra correspondente. Assim, esta tabela vai conter as palavras do dicionário que nos interessarem. |

Para guardar a informação referente ao máximo de problemas de cada tamanho existente criamos uma tabela denominada ***max_mutacoes***.

Tabela Tridimensional de carateres: ***grafos***

No que toca ao grafo criado, utilizamos uma tabela tridimensional com as mesmas dimensões da tabela tridimensional presente na estrutura do dicionário sendo que esta será como a sua reprodução em que no índice correspondente de cada palavra teremos a sua lista de adjacências.

Relativamente à lista de adjacências optamos por usar uma estrutura denominada ***lista_adj***, que será uma lista simplesmente ligada genérica tal como ao longo desta unidade curricular foi diversas vezes implementado.

Para o grafo, mais concretamente para o elemento genérico da lista de adjacências, optou-se por uma estrutura denominada ***aresta*** para que pudéssemos guardar todas as informações relativas às palavras que são adjacentes. Constituída por:

| Campos | Descrição |
|---|---|
| Inteiro: **indice_adjacente** | Este inteiro irá conter o índice adjacente. |
| Inteiro: **peso** | Este inteiro irá conter o peso da ligação. |

Relativamente à fila de prioridades utilizada no algoritmo de Dijkstra, optou-se por usar um acervo. De forma a permitir a sua implementação foi criada uma estrutura denominada ***heap***, constituída por:

| Campos | Descrição |
|---|---|
| Função: **(*maior) (Item, Item)** | Esta função associada à estrutura (definida pelo Cliente) irá definir a condição de acervo. |
| Inteiro: **num_elementos** | Este inteiro irá conter o número de elementos do acervo. |
| Inteiro: **tamanho** | Este inteiro irá conter o tamanho do acervo. |
| Vetor de inteiros: **posicao** | Este vetor de inteiros irá conter a posição na fila de prioridades de cada elemento, que no nosso caso é um vértice. |
| Vetor genérico: **fila_p** | Este vetor genérico irá definir a fila de prioridades. |

Para a implementação do Dijkstra foram ainda utilizadas duas tabelas de dados auxiliares denominada ***wt*** e ***st*** que foram utilizadas tal como está enunciado nos Acetatos da Unidade Curricular, sendo que, o ***wt*** foi utilizado para guardar as distâncias ao índice fonte e o ***st*** foi utilizado para guardar o vértice antecessor de cada um dos vértices.

# DESCRIÇÃO DOS ALGORITMOS

## Algoritmos Estudados

Os principais algoritmos utilizados que foram lecionados na Unidade Curricular foram o ***Quicksort***, ***Procura Binária*** e o ***Dijkstra*****.**

***Ordenar/Quicksort:***

Para a ordenação alfabética de todas as palavras do dicionário presentes na tabela, foi utilizado o algoritmo de ordenação ***Quicksor***t, na sua representação recursiva, onde se efetua a partição dos dados (palavras) em subtabelas e se ordena cada parte independentemente.

Para a aplicação deste algoritmo, percorre-se todos os tamanhos de palavras existentes no dicionário, e caso haja algum problema referente a esse tamanho (e seja condição para criar grafo), aplica-se o algoritmo para esse tamanho. Estando as palavras desse tamanho todas ordenadas, a procura de uma certa palavra no dicionário torna-se muito mais eficiente.

***Procura binária:***

Para a aplicação do algoritmo de Dijkstra, de modo a localizar a palavra de partida e a palavra de chegada no dicionário, necessitamos de realizar uma procura ao dicionário já ordenado. Para isso utilizámos o algoritmo de ***Procura Binária***, em que, por cada comparação de duas palavras, se excluem metade das possíveis posições da palavra no dicionário (pois o valor retornado pela função strcmp na comparação permite optar por uma metade ou outra).

***Dijkstra:***

Para se calcular o caminho com menos custo entre duas palavras utilizou-se o algoritmo de ***Dijkstra***. Para a aplicação deste algoritmo foi necessária a criação de uma fila de prioridades. Para tal, utilizamos a fila de prioridades através de um acervo devido à eficiência na sua manipulação, que é essencialmente logarítmica. Após as inicializações do vetor com as distâncias ao índice fonte, wt (inicialmente todos os vértices têm um peso 10001 menos o da fonte que se encontra a 0), do vetor com os vértices antecessores de cada um dos vértices, st (todos a -1 exceto o da fonte que se encontra com o valor dele próprio) e a inclusão de todos vértices precisos para o problema na fila de prioridades, o algoritmo de ***Dijkstra*** funciona como representado no fluxograma que se segue:

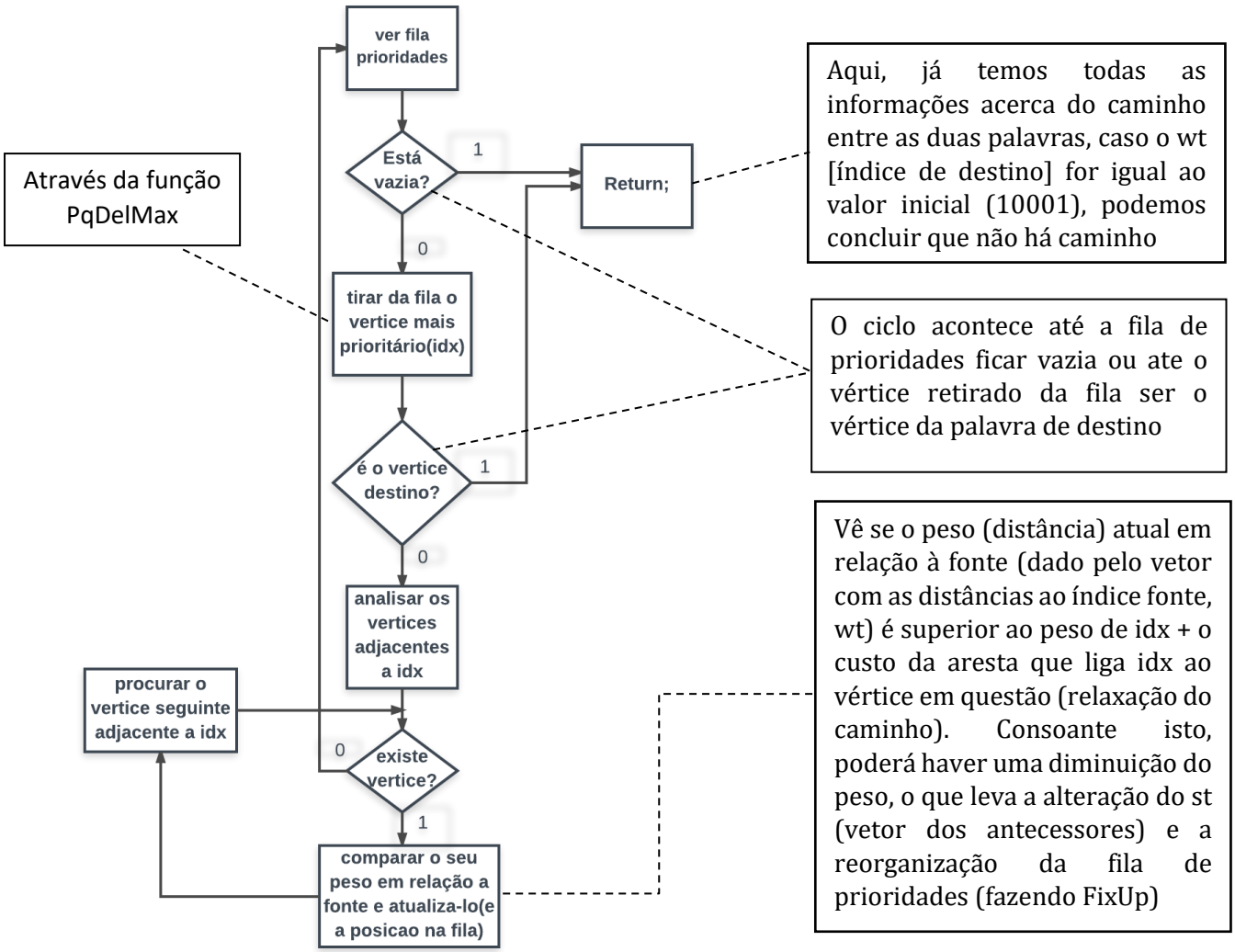



***Figura 2 –*** *Fluxograma do algoritmo de* ***Dijkstra***

## Algoritmos Criados

A par dos algoritmos estudados nas aulas teóricas, também consideramos de extrema importância as duas seguintes funções que têm um papel fulcral na memória e eficiência do programa, sendo, portanto, determinantes no resultado final:

***Ler1_problemas:***

Na primeira vez que lemos o ficheiro de problemas, retiramos informações bastante importantes para a alocação e respetivo preenchimento do dicionário e dos grafos: ficamos a saber os tamanhos de palavras necessários à resolução dos problemas, e o máximo de mutações existentes para cada tamanho (de modo a construir o grafo desse tamanho em função desse valor). Por este raciocínio, caso tivéssemos um ficheiro de entrada .pal com os seguintes problemas:

joao pata 1

foca fita 2

rato pato 3

quando fossemos a criar o grafo para os problemas de tamanho 4, iriamos estar a fazer todas as ligações possíveis de palavras de 4 letras que difiram entre si até 3 caracteres, inclusive. No entanto, como se pode observar, o problema que define esse número máximo de mutações é o 3º (rato, pato), e estas duas palavras diferem entre si apenas por 1 caracter, pelo que o problema é resolvido apenas com uma mutação. Assim, se seguíssemos este raciocínio iriamos estar a alocar memória desnecessária e iria tornar o programa menos eficiente. Neste exemplo concreto, o grafo por nós criado iria ser criado tomando em conta todas as ligações possíveis de palavras de 4 letras que difiram entre si até 2 caracteres. Para contornar este obstáculo, a forma como vemos qual é o máximo de mutações para cada tamanho é ilustrada no fluxograma que se segue na página seguinte, referente a uma parte da função ***Ler1_problemas***:

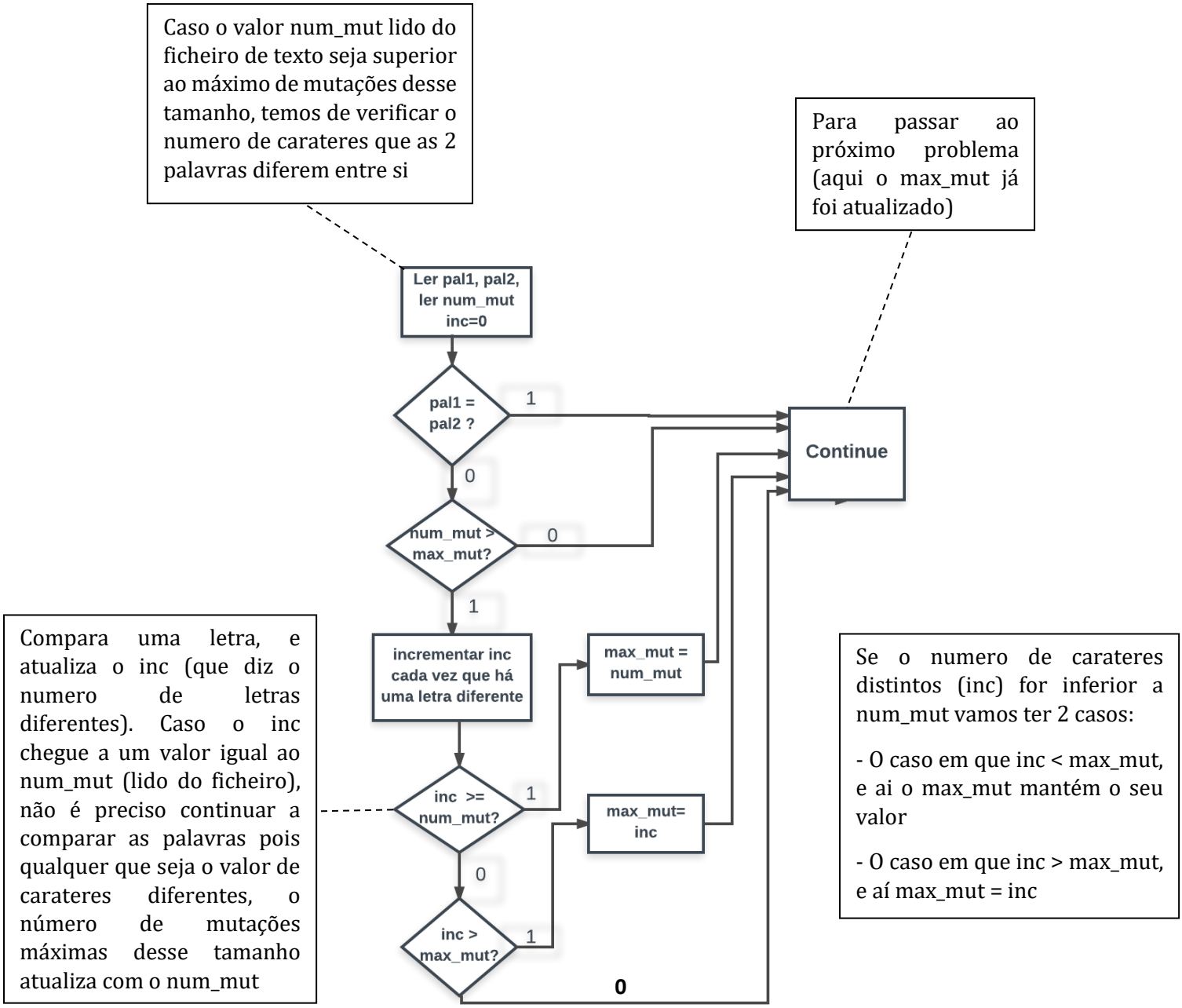



***Figura 3 -*** *Fluxograma do* ***ler1_problemas***

***Criar_grafos:***

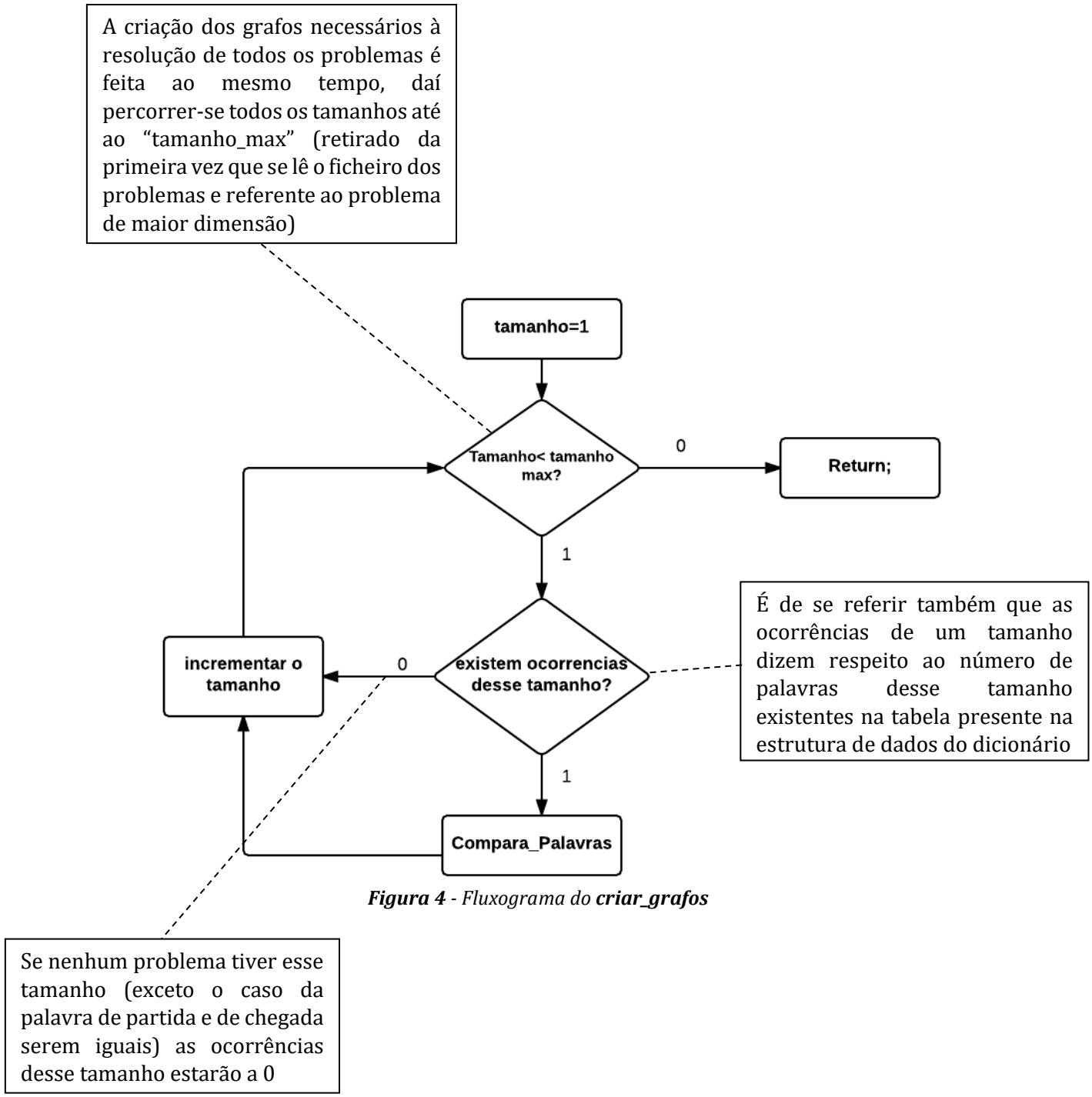
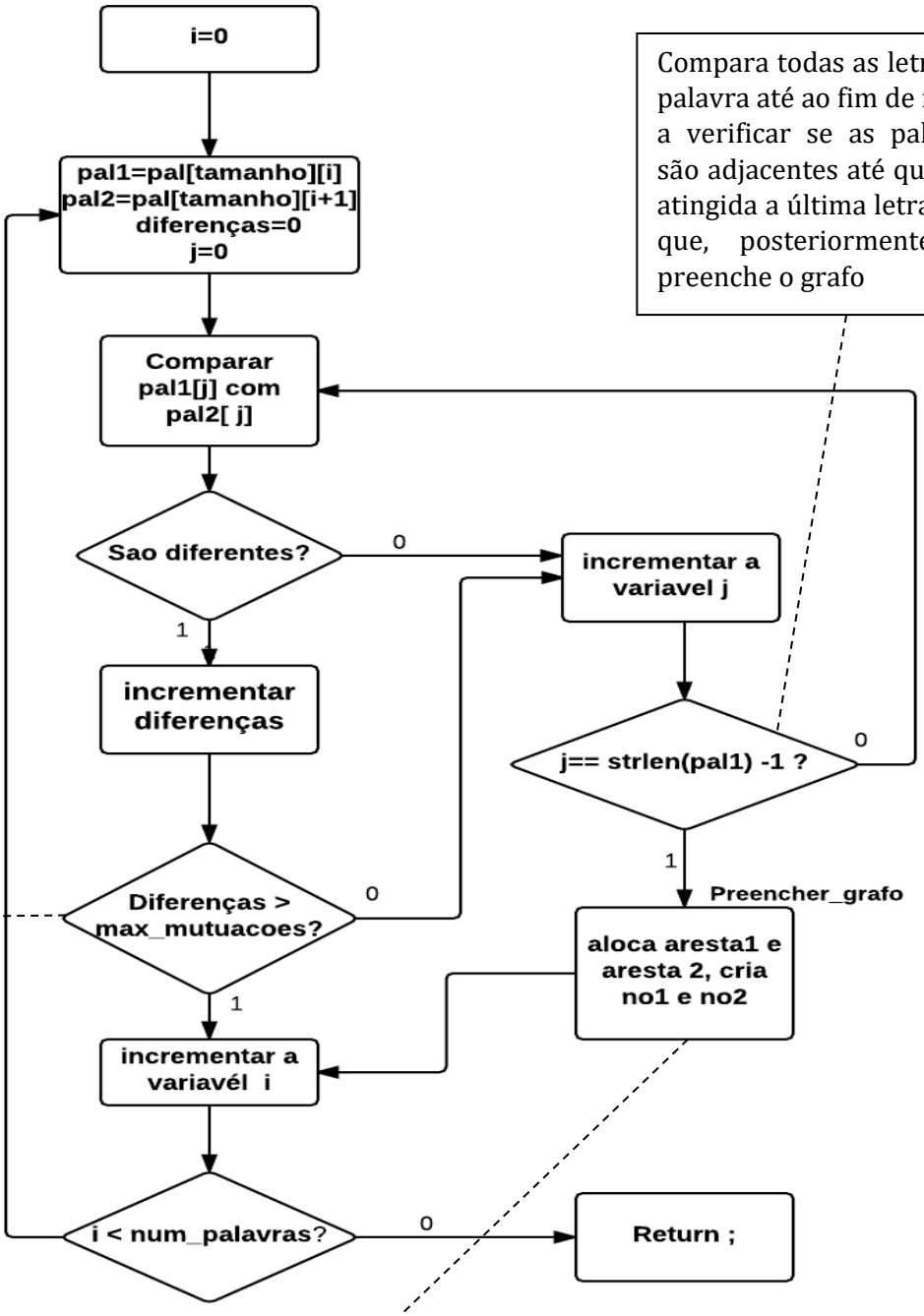



***Figura 4*** *- Fluxograma do* ***criar_grafos***

Para construir um grafo temos de comparar cada palavra desse tamanho com todas as outras, letra a letra, e caso o numero de mutações entre elas for menor que o max_mutações, ligá-las. Para isso, temos a função ***compara_palavras***, cujo fluxograma se apresenta em seguida:




Compara todas as letras da palavra até ao fim de forma a verificar se as palavras são adjacentes até que seja atingida a última letra para que, posteriormente, se preenche o grafo

Nesta subrotina, repetida imensas vezes ao longo do programa, quando se comparam 2 palavras e se deteta que o nº de mutações (diferenças) entre elas já excedeu o número máximo de mutações que poderiam ter, passa-se logo à próxima comparação entre as próximas 2 palavras (pois estas já não se podem ligar).

***Figura 5** - Fluxograma da subrotina **compara_palavras***

Quando 2 palavras se ligam, como o grafo em questão não é direcionado, cria-se 2 nós: um para meter a pal2 na lista de adjacências da pal1, e outro para fazer o recíproco

## DESCRIÇÃO DOS SUBSISTEMAS

O programa por nós implementado contém quatro subsistemas fundamentais: ***Dicionário*** (onde as estruturas associadas estão definidas no dicionario.h e as suas funções respetivas no dicionario.c); ***Grafos*** (onde as estruturas associadas estão definidas na lista_adjacencias.h e grafo.h e as suas funções respetivas no lista_adjacencias.c e grafo.c); ***Acervo*** (onde as estruturas associadas estão definidas no acervo.h e as suas funções respetivas no acervo.c); ***Ficheiros*** (onde as estruturas associadas estão definidas no ficheiros.h e as suas funções respetivas no ficheiros.c).

### *Dicionário*

O objetivo deste subsistema foi o de podermos guardar e aceder a toda a informação útil relativamente ao ficheiro de texto que contenha o dicionário, fundamentalmente, o número de palavras dos tamanhos que nos interessarem e as próprias palavras. Quando se refere "tamanhos que nos interessarem" estamo-nos a referir aquelas palavras para qual os problemas existentes requerem-nos a guardar a sua informação.

- Funções de Criação e Inicialização

  dicionario * **inicializa_dicionario** (int _max_prob);

  int * **inicializar_vetor** (int _max_prob);

Estas funções criam e inicializam toda a estrutura de dados referente ao dicionário, nomeadamente, a estrutura em si e os dois vetores de inteiros a ela associados (vetor das ocorrências de cada tamanho de palavras e vetor das posições livres de cada tamanho).

- Função de Recolha da Informação

  void **ler1_dicionario** (FILE * fp, dicionario * _dic, int _max_prob, int _max_mutacoes[]);

Esta função lê o ficheiro de texto do dicionário e recolhe toda a informação necessária para alocação da tabela tridimensional que irá conter as palavras com os tamanhos uteis para a resolução dos problemas existentes.

- Função de Criação da Tabela Tridimensional

  dicionario * **alocar_dicionario** (dicionario * _dic, int _max_prob);

Esta função aloca a tabela tridimensional de acordo a informação recolhida anteriormente na leitura de ambos os ficheiros de entrada.

- Função de Preenchimento do dicionário

  void **ler2_dicionario** (FILE * fp, dicionario * _dic, int *_posicao_livre, int _max_prob, int _max_mutacoes[]);

  void **escreve_palavra** (dicionario * _dic, int i , int j, char * pal1);

Esta função lê novamente o ficheiro de texto do dicionário, mas guarda apenas as palavras com os tamanhos que nos são uteis.

- Função de Ordenação

  void **ordenar** (dicionario * dic,int _max_prob);

  void **quicksort** (char ** _palavras, int l, int r);

  int **partition** (char ** _palavras, int l, int r);

  void **troca_strings** (char ** _palavras, int i, int j);

Este conjunto de funções ordena alfabeticamente as palavras de cada tamanho.

- Funções de manipulação e leitura de dados

  int * **retorna_ocorrencias** (dicionario * _dic);

  int * **retorna_posicao_livre** (dicionario * _dic);

  char *** **retorna_palavras_dic** (dicionario * _dic);

Estas funções auxiliares dão nos acessos a estrutura de dados associada a este subsistema e à manipulação dos mesmos.

- Funções de libertação de memória

  void **libertar_dicionario** (dicionario * _dic, int _max_prob);

  void **libertar_vetor** (int * _vetor);

Estas funções libertam toda a memória associada a este subsistema.

***Grafos***

O objetivo deste subsistema foi o de podermos guardar, manipular e aceder a toda a informação relativamente aos grafos criados, nomeadamente, os índices adjacentes e os pesos de cada aresta que componha o nó das diversas listas de adjacências existentes.

- Funções de Criação e Inicialização

  lista_adj *** **criar_grafos** (int _max_prob, dicionario * _dic, int max_mutacoes[]);

  lista_adj *** **aloca_grafos** (int * _ocorrencias, int _max_prob);

  aresta * **aloca_aresta** (int _indice, int peso);

  lista_adj * **cria_no** (lista_adj * next, Item this);

Estas funções criam e inicializam toda a estrutura de dados referente ao grafo, nomeadamente, a tabela tridimensional que tem as mesmas dimensões que a tabela tridimensional do subsistema do dicionário. Para além disso, há também as funções que criam e inicializam a estrutura da aresta e o nó da lista que iram formar as várias listas de adjacências existentes.

- Funções de Preenchimento do Grafo

  void **compara_palavras** (lista_adj *** _grafo, int _max_mutacoes[], int tamanho, char * pal1, char * pal2, int indice_p1, int indice_p2);

  void **preencher_grafo** (lista_adj *** _grafo, int num_mutacoes, int _indice_p1, int _indice_p2, int tamanho);

Neste conjunto de funções refere-se aquela que compara todas as palavras de cada tamanho de forma a verificar quais as que são adjacentes entre si. Caso se verifique a sua adjacência cria-se as arestas e os nó das listas de adjacências correspondentes de forma a preencher o grafo no tamanho e índices corretos.

- Funções de manipulação e leitura de dados

  int **retorna_indice_adj** (aresta *a);

  int **retorna_peso_vertice** (aresta *a);

  Item **obter_item** (lista_adj * no);

  lista_adj * **obter_next** (lista_adj * no);

Estas funções auxiliares dão nos acesso à estrutura de dados associada a este subsistema e à manipulação dos mesmos.

- Funções de libertação de memória

  void **liberta_memoria** (lista_adj *** _grafo, dicionario * _dic, int _max_prob);

  void **liberta_item** (Item this);

  void **free_lista** (lista_adj * head , void (* free_item)(Item));

Estas funções libertam toda a memória associada a este subsistema, ou seja, liberta cada aresta, cada nó de todas as listas de adjacências e, por fim, a tabela tridimensional que forma o grafo.

***Acervo***

O objetivo deste subsistema foi o da criação de uma fila de prioridades para auxílio na implementação do algoritmo de Dijkstra. Desta forma, foi utilizado um acervo genérico de modo a aumentar a eficiência na criação e manipulação destes dados (vértices do tamanho de palavras em questão).

- Função de Criação e Inicialização

heap * **nova_heap** (int _tamanho, int (*maior) (Item, Item));

Esta função cria e inicializa toda a estrutura de dados referente ao acervo, nomeadamente, a estrutura em si, a função associada a estrutura, o tamanho do acervo, o número de elementos, o vetor auxiliar que contém a posição de cada elemento na fila de prioridades e, ainda, o acervo propriamente dito.

- Funções de manipulação e leitura de dados

void **FixUp** (heap * h, int k);

void **FixDown** (heap * h, int k);

void **PQinsert** (heap * h,Item I);

Item **PQdelmax** (heap * h);

int **PQempty** (heap * _h);

int **retorna_posicao** (heap *h,int i);

Estas funções dão nos acesso a estrutura de dados associada a este subsistema e à manipulação dos mesmos, nomeadamente, no que se refere a manipulação da fila de prioridades, mantendo sempre a condição de acervo. É de se referir que a última função mencionada retorna a posição de um vértice na fila de prioridades. Esta posição é alterada nas funções FixUp, FixDown, PQinsert e PQdelmax.

- Funções de libertação de memória

void **liberta_heap** (heap * h);

Estas funções libertam toda a memória associada a este subsistema.

***Ficheiros***

Este será de certa forma o subsistema mais importante para o bom funcionamento do programa sendo que os outros são como a base deste. Por conseguinte, o seu objetivo é o tratar toda a informação relativa aos ficheiros de texto, ou seja, primeiro, verifica a correção das extensões dos ficheiros de entrada, posteriormente, recolhe toda a informação necessária do ficheiro de texto dos problemas, resolve cada um dos problemas e, por fim, escreve o ficheiro de texto de saída.

- Função de Verificação dos Ficheiros de Entrada

char * **verificar_ficheiros** (char * _ficheiro_in_dic, char * _ficheiro_in_prob, char * _ficheiro_out);

Esta função verifica se a extensão de cada um dos ficheiros de entrada é a correta, ou seja, o primeiro ficheiro de entrada (que será o que contém o dicionário) terá que ter a extensão .dic e o segundo ficheiro de entrada (que será o que contém os problemas) terá que ter a extensão .pal.

- Função de Recolha da Informação

void **ler1_problemas** (FILE * _fp, int * max_prob,int max_mutacoes[]);

Esta função lê o ficheiro de texto dos problemas pela primeira vez e recolhe daí toda a informação que é imprescindível para o sucesso de todo o programa, isto é, o tamanho máximo dos problemas e se há ou não necessidade da criação do grafo e aplicação do algoritmo de Dijkstra para cada tamanho. Esta função foi já explicada detalhadamente em cima.

- Funções de Criação e Inicialização

  void **ler2_resolve_problemas** (FILE* _fp_prob, FILE* fp_out, dicionario *_dic, lista_adj ***grafo);

Estas funções estão associadas a função que lê pela segunda vez os problemas e resolve cada um deles, pois aí é necessário criar e inicializar algumas variáveis para permitir o seu bom funcionamento. São essas variáveis os dois vetores auxiliares wt e st que irão conter as distâncias ao índice fonte e o vértice antecessor de cada um dos vértices, respetivamente, e, ainda, o ponteiro para inteiros, num, que irá ser utilizado como auxiliar para colocar cada um dos vértices no acervo.

- Funções que implementam algoritmos

  int **procura** (char ** _palavras, char * pal, int peq, int maior);

  void **dijkstra** (lista_adj *** grafo,int tam_palavra, heap * _fila_p,int _num_vertices,int _indice_fonte, int _indice_destino, int _peso_max);

Aqui temos duas funções que implementam algoritmos conhecidos, como a procura binária e o Dijkstra, que já foram explicados acima ao detalhe e que foram implementados de forma idêntica a estudada nas aulas com os seus devidos ajustes ao nosso programa.

- Funções de escrita do ficheiro de saída

  void **escrever_ficheiro** (FILE* fp,char ** _palavras,int idx, int custo);

  void **escever_ficheiro_sem_caminho** (FILE * _fp_out, char ** _palavras, int indice_fonte, int indice_destino);

Existem duas funções de escrita do ficheiro de saída, uma muito simples para os problemas sem caminho onde apenas é necessário escrever a palavra inicial (fonte) e a palavra de destino seguida de -1 e outra que foi implementada de forma recursiva onde utilizando o vetor auxiliar st tomávamos como ponto de partida o índice de destino e recuávamos nos seus antecessores até ser atingido o critério de paragem, isto é, até chegarmos ao índice inicial (fonte). Desta forma, conseguíamos imprimir no ficheiro de saída o caminho entre as duas palavras.

- Funções de libertação de memória

  void **ler2_resolve_problemas** (FILE* _fp_prob, FILE* fp_out, dicionario *_dic, lista_adj ***grafo);

Estas funções estão associadas a função que lê pela segunda vez os problemas e resolve cada um deles, pois aí é necessário estar constantemente a libertar o acervo criado para cada problema e, por vezes, libertar os vetores auxiliares wt e st quando surge um problema de tamanho diferente do anterior.

## ANÁLISE DOS REQUISITOS COMPUTACIONAIS

Todas as escolhas tomadas ao longo do projeto, quer no tipo de estruturas de dados utilizadas, quer nos algoritmos usados, tiveram em atenção a eficiência, a memória e a complexidade do programa.

**Dicionário -** Na criação da tabela das palavras, contida na estrutura de dados do dicionário, necessitamos de aceder regularmente a valores. Para que esse acesso fosse feito diretamente (com complexidade O(1)) utilizaram-se 2 vetores, em que a memória associada a cada um deles é igual ao tamanho do maior problema (lido de início no ficheiro de problemas). A memória utilizada para a alocação destes vetores é muito pequena em comparação com a memória total utilizada. A memória gasta associada ao dicionário é proporcional ao número de tamanhos necessários para resolver todos os problemas e, dentro desses tamanhos, ao número de palavras existentes.

**Grafos –** Na criação de grafos comparámos cada palavra na tabela ordenada com todas as outras que se encontram em posições superiores. Assim, sendo N o número de palavras do tamanho associado ao grafo a criar, o número de comparações de palavras é igual a N*(N-1)/2, pelo que a complexidade associada a este processo é $O(N^2)$. E é aqui, na criação de todos os grafos, que se consome a maior parte do tempo e memória do programa. Sendo que para cada ligação entre 2 vértices alocamos 2 arestas (estrutura que tem o peso e o vértice adjacente), a memória alocada associada ao preenchimento de cada grafo é proporcional ao número de arestas criadas (número total de nós na lista de adjacências). Sendo M o número de ligações entre palavras, como o número de arestas é igual a 2*M, a complexidade de memória associada é de O(M). Após a formação de cada nó, a sua inserção na lista de adjacências é O(1), pois inserimos sempre no início da lista. Aquando da implementação do algoritmo de Dijkstra, temos a necessidade de percorrer a lista de adjacências do vértice que foi retirado da fila de prioridades, sendo que a sua complexidade é O(V) em que V é o número de vértices adjacentes a palavra correspondente.

**Estrutura heap -** A implementação da fila de prioridades através de uma heap permite o melhor desempenho na procura, troca de posições e remoção dos elementos de maior prioridade, a utilizar no algoritmo de Dijkstra. Após a remoção do elemento mais prioritário da fila (feito diretamente, O(1)), tem de se verificar que a fila mantém a condição de acervo. Assim, sendo N o número de elementos existentes na fila, as funções a si associadas, que fazem com que a fila verifique a condição de acervo (FixDown e FixUp) têm uma complexidade de O(Log(N)).

**Vetor posições, vetor pesos (wt) e vetor adjacentes (st) -** Na invocação das funções associadas à heap, é necessário saber a posição na heap do vértice a atuar. A criação do vetor de posições permite-nos saber a posição que um determinado vértice ocupa na heap com complexidade O(1) pois acedemos diretamente a esse valor. A memória associada é igual

ao número de elementos na heap, que é o número de vértices existentes no grafo. O mesmo acontece para os vetores wt e st que têm uma memória associada idêntica.

**Ordenar:** O algoritmo de ordenação utilizado, Quicksort, implementado de forma recursiva, ordena um tamanho (com N palavras distintas) de cada vez, onde tem complexidade média NlogN, e, no pior dos casos, quadrática $O(N^2)$.

**Procura:** Sendo que a procura por nós utilizada, como já se referiu, foi a procura binária a uma tabela já ordenada, em que M é o número de palavras existentes na tabela ordenada do tamanho da palavra que queremos encontrar e a complexidade associada a este algoritmo é O(log(M)), pois por cada iteração se reduz a metade o número de elementos a ter em conta.

**Dijkstra:** Sendo que a forma como implementámos o Algoritmo de Dijkstra foi com um acervo, sempre que se retira um vértice da fila prioritária, esta vai ter de continuar a cumprir com a condição de acervo imposta (neste caso é o peso). Como vimos, sendo N o número de vértices, a complexidade associada a restabelecimento desta condição quando se retira um elemento da fila é O(logN). Sabendo que, no pior caso, cada aresta é visitada no máximo uma e uma só vez, sendo M o número de arestas, o pior caso que se pode obter para este algoritmo é O(M*logN).

## EXEMPLO DO FUNCIONAMENTO DO PROGRAMA

Imaginemos que temos um dicionário com as seguintes palavras: gato, fifa, pato, fala, fila, fifi. Ordenando o dicionário, ficamos assim com:

| **Índice** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| **Palavra** | Fala | Fifi | Fifa | Fila | Gala | Gato | pata | pato |

(conteúdo da tabela tridimensional pertencente a estrutura de dados do dicionário)

O primeiro ficheiro de texto tem os seguintes problemas:

***fala fifi 1***

***fala gato 1***

Desta forma o vetor inserido na estrutura do dicionário que contém as ocorrências estará com todos os valores a 0 a exceção do índice 3 que corresponde as palavras de tamanho 4 que está com o valor de 8, pois são o número de palavras que existe no dicionário com tamanho 4.

Quanto a estrutura de dados do grafo criado, as listas de adjacências correspondentes seriam:

(0) – (4) -> (3) -> NULL
(1) – (2) -> NULL
(2) – (3) -> (1) -> NULL
(3) – (2) -> (0) -> NULL
(4) – (0) -> NULL
(5) – NULL
(6) – (7) -> NULL
(7) – (6) -> NULL

em que o primeiro valor de cada linha representa o índice do vértice a avaliar e os outros valores os índices dos seus adjacentes.

Como só se pode ter no máximo uma mutação e o vértice fonte é o 0 (fala), a SPT associada ao problema é:

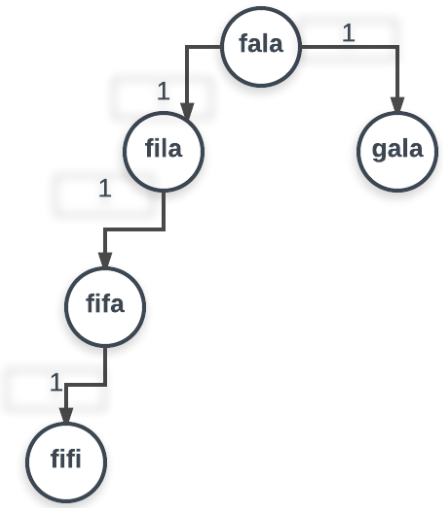



***Figura 6 -*** *SPT associada ao primeiro problema*

Facilmente se verifica que o programa irá criar um ficheiro de saída com os seguintes resultados:

***Fala 3***

***Fila***

***Fifa***

***Fifi***

***Fala -1***

***Gato***

Consideremos, para o mesmo dicionário, um outro ficheiro de texto que tem os seguintes problemas:

***pato gato 3***

***pato pato 2***

Desta forma o vetor inserido na estrutura do dicionário que contém as ocorrências estará com todos os valores iguais ao do problema anterior. Quanto a estrutura de dados do grafo criado, as listas de adjacências correspondentes manter-se-iam também iguais, uma vez que estamos a lidar sempre com o mesmo dicionário e o número de mutações máxima dos problemas em questão mantém-se igual (uma mutação).

O facto a realçar neste exemplo é que como se pode observar, o primeiro problema diz que o máximo de mutações entre as 2 palavras é de 3 carateres. No entanto, estas 2 palavras diferem apenas por um caracter, levando a que seja apenas necessário ter em conta as ligações do grafo com peso igual ou inferior a 1. No segundo problema, observa-se que as 2 palavras são iguais, pelo que nem é necessária a realização do algoritmo de Dijkstra para este problema.

Portanto, o grafo a criar pode ter ligações apenas de peso 1.

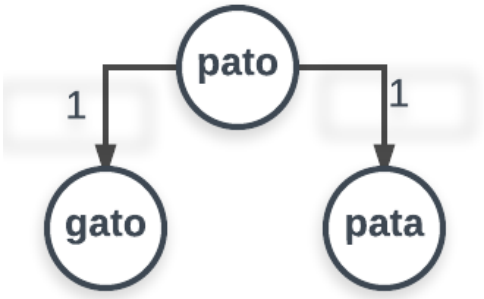



***Figura 7 -*** *SPT associada ao segundo problema*

Os resultados obtidos no ficheiro de saída são os seguintes:

***pato 1***

***gato***

***pato 0***

***pato***

# ANÁLISE CRÍTICA

As primeiras seis submissões do projeto não corresponderam às expetativas visto que não passamos em nenhum dos testes devido a um erro de compilação. Esta situação, inicialmente, ocorreu devido a incorreta escrita do ficheiro Makefile. Após este ter sido corrigido, numa das submissões comprimimos a pasta com os ficheiros de forma errada e, por fim, faltava-nos inicializar uma variável o que se traduzia num warning no site de submissões. Estes percalços foram claramente a parte que correu menos bem no projeto final.

Depois disso, finalmente, conseguimos ter uma submissão bem conseguida e onde conseguimos logo passar nos 20 testes fornecidos pelo corpo docente. Posteriormente, fizemos mais uma submissão onde completámos os comentários ao código e organizámo-lo melhor.

Ainda assim, pensamos que o nosso projeto não está com o máximo de eficiência possível quer seja em termos de tempo como de memória. Isto aconteceu, pois, como já tínhamos concluído com sucesso os 20 testes fornecidos pelo corpo docente não alterámos o código com receio de poder diminuir o número de testes bem-sucedidos. Por exemplo, no caso de as palavras fornecidas no ficheiro de problemas diferirem apenas 1 mutação entre elas não seria necessário aplicar o algoritmo de Dijkstra. Por outro lado, caso o número máximo de mutações desse tamanho fosse igualmente de 1, nem seria preciso criar o grafo. Outro exemplo, tem haver com o facto de não ser necessário estarmos a alocar constantemente uma nova estrutura para o acervo e para a fila de prioridades para cada problema, pois poderíamos ter alocado a estrutura do acervo tendo em conta o maior número de ocorrências dos tamanhos existentes o que evitava as constantes alocações e libertações de memória, o que não implicaria diferenças no pico de memória mas tornaria o programa mais rápido. O nosso programa não tem em conta estes dois casos, mas temos consciência que poderíamos ter feito estas melhorias.

Ao longo do programa, tentámos utilizar sempre o mais alto nível de abstração de forma a tornar o nosso programa o mais genérico possível, mantendo desta forma a relação fornecedor-cliente bem estabelecida, em que neste caso particular representamos ambos.

Em suma, pensamos que o projeto foi bem conseguido e que as decisões feitas ao longo do projeto, quer seja na escolha das estruturas de dados, algoritmos e estratégias foram tomadas de modo a tornar o projeto o mais eficiente possível conciliando, ao mesmo tempo, com gestão de memória e reduzida complexidade.

# BIBLIOGRAFIA

1. Acetato 04 – "EstDados": elemento genérico
2. Acetato 05 – "Análise": algoritmo de procura binária
3. Acetato 07 – "SortB": algoritmo do Quick Sort
4. Acetato 10 – "GrafosC": algoritmo de Dijkstra 4
5. Laboratório 4 – "LinkedList.c"
6. Laboratório 6 – "heap.c"