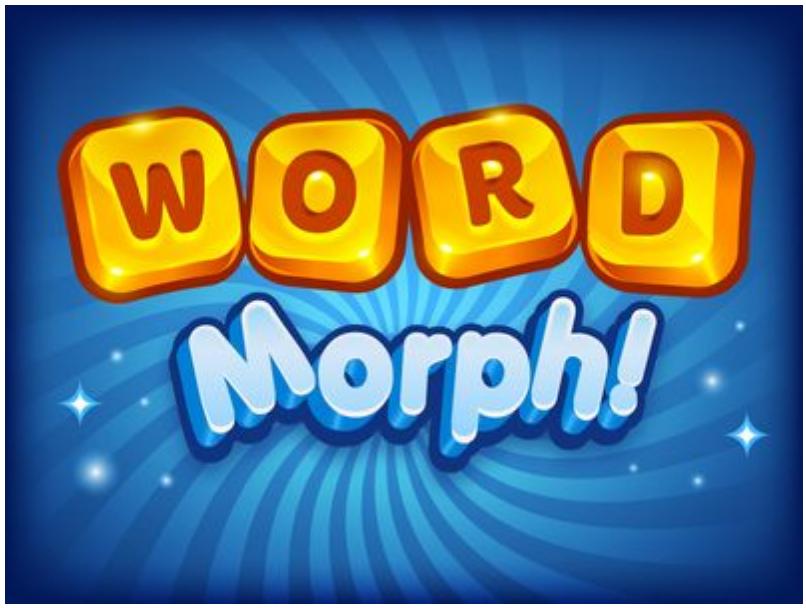
Mestrado Integrado em Engenharia Electrotécnica e de Computadores (MEEC)

Algoritmos e Estruturas de Dados

2016/2017 - 1º semestre

Relatório do projeto



Docente
Carlos Bispo

Grupo nº37
Madalena Muller nº 78708 - madalena.muller95@gmail.com
Mariana Martins nº 80856 - mrrvm@hotmail.com

Índice

Descrição do problema

O problema proposto neste projeto de Algoritmos e Estruturas de Dados é o de criar um caminho de palavras, entre uma palavra de partida e uma de chegada tendo estas o mesmo tamanho. Este percurso é feito através da substituição de letras que dá origem a uma sequência de palavras que pertence a um determinado dicionário.
Numa primeira fase, sendo dado o dicionário e os problemas, pretende-se apenas indicar quantas palavras existem no dicionário que têm o mesmo tamanho da palavras do problema dado ou indicar a sua posição alfabética entre as palavras do dicionário do mesmo tamanho.
A fase final tem como objetivo calcular o caminho com menor custo entre duas palavras dadas, sendo esse custo dado pela soma dos quadrados do nº de mutações (letras que mudam) de palavra para palavra no caminho .

Resolução do problema

Na fase final, a nossa resolução divide-se em três partes, **criação do dicionário** e ciclicamente, **resolução do problema** e **escrita da sua solução** por cada problema a resolver pela ordem que está no ficheiro.

Na **primeira parte**, é feita a leitura dos ficheiros dados e armazenam-se as palavras dadas num vetor de vetores (dicionário). Os índices do vetor-mãe representam o número de letras que as palavras guardadas no elemento com esse índice podem ter, e assim as palavras ficam distribuídas pelos vetores-filho conforme o seu tamanho. A criação do dicionário foi já feita na primeira fase do projecto, sendo a chave para resolver os problemas tanto para a opção 1 (número de palavras com o mesmo tamanho) como para a opção 2 (localizar a posição da palavra).

Já na **resolução do problema**, caso haja um com dado tamanho de palavra, é criado o grafo para esse tamanho a partir do vetor de vetores, se ainda não tiver sido criado. Cada elemento de cada vetor-filho tem uma lista de adjacências, em que os nós da lista representam uma palavra e contêm o seu peso da substituição de letras em relação à palavra que está a "segurar" a lista.
A seguir é executado o algoritmo de Dijkstra para encontrar a solução do problema, com auxílio de um acervo binário. Neste acervo, a primeira posição corresponde à palavra de partida (elemento 'pai'), e são criadas ligações a outras palavras por ordem de peso. O peso neste caso corresponde ao número de letras que mudam entre as palavras ligadas ao quadrado.
Finalmente, determina-se o caminho de menor custo até à palavra de chegada e **escreve-se** o mesmo juntamente com o custo/peso total **para o ficheiro**, recursivamente. Caso não exista um caminho possível o programa escreve apenas a palavra de partida e de chegada sem nenhum caminho possível e -1. Passa-se ao próximo problema até concluir a resolução de todos.
Na primeira fase foram usados os algoritmos *binary search* e *merge sort,* que são depois usados na fase final.

Arquitectura do programa

De um forma generalizada, o funcionamento do programa resume-se ao fluxograma da figura 1. É agora explicado cada um dos processos assinalados.

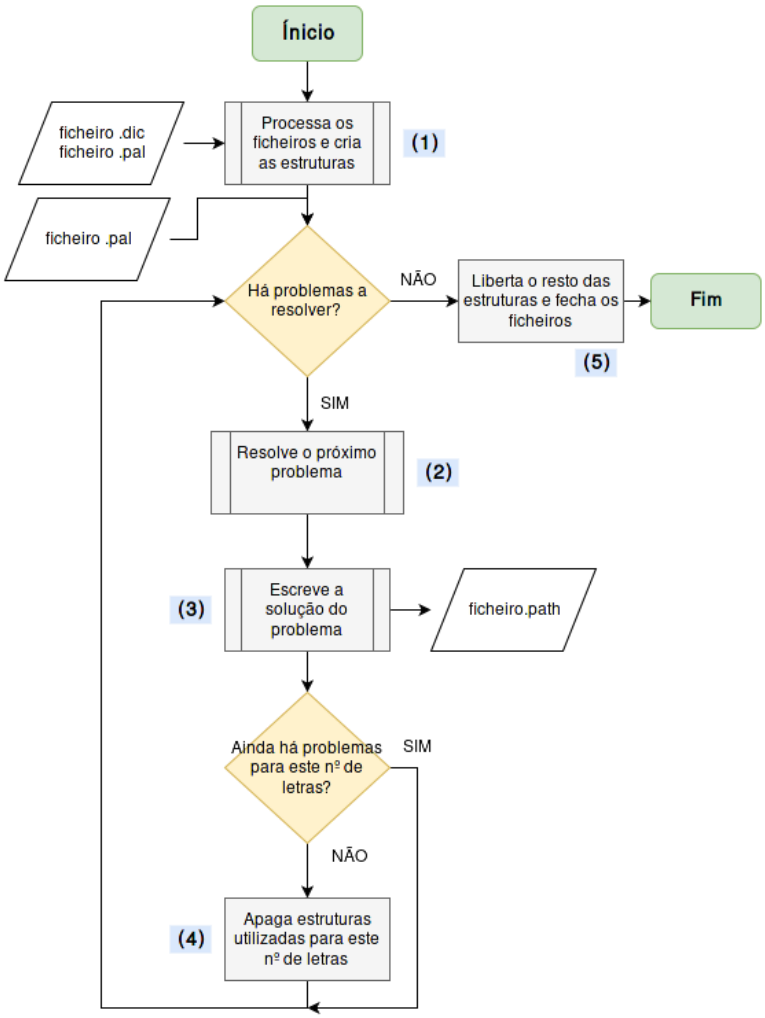


Figura 1 - Fluxograma do funcionamento do programa

Inicialmente **(em 1)**, é processado o ficheiro *.pal*, de forma a saber que tamanhos de palavra é que irão resolver problemas, o nº de problemas a resolver que existem para cada um desses tamanhos e o máximo do nº de mutações máximas dos problemas dados para cada tamanho. Depois é processado o ficheiro *.dic* uma primeira vez, de forma a obter o nº de palavras que existem para cada tamanho que irá potencialmente resolver problemas. E ainda uma outra vez, para inserir as palavras nas estruturas criadas após o primeiro processamento. Esta estrutura é um vetor de vetores, em que os índices do vetor mãe (*indexing_vector*) representam o nº de letras, e cada elemento dos vetores filhos (*word_vector)* contém uma palavra e um ponteiro para uma futura lista de adjacências. O fluxograma para esta primeira parte, encontra-se na figura 2.

NOTA: na determinação do máximo do nº de mutações máximas dos problemas dados, não são contadas as mutações dos problemas cujas as palavras só diferem numa letra.

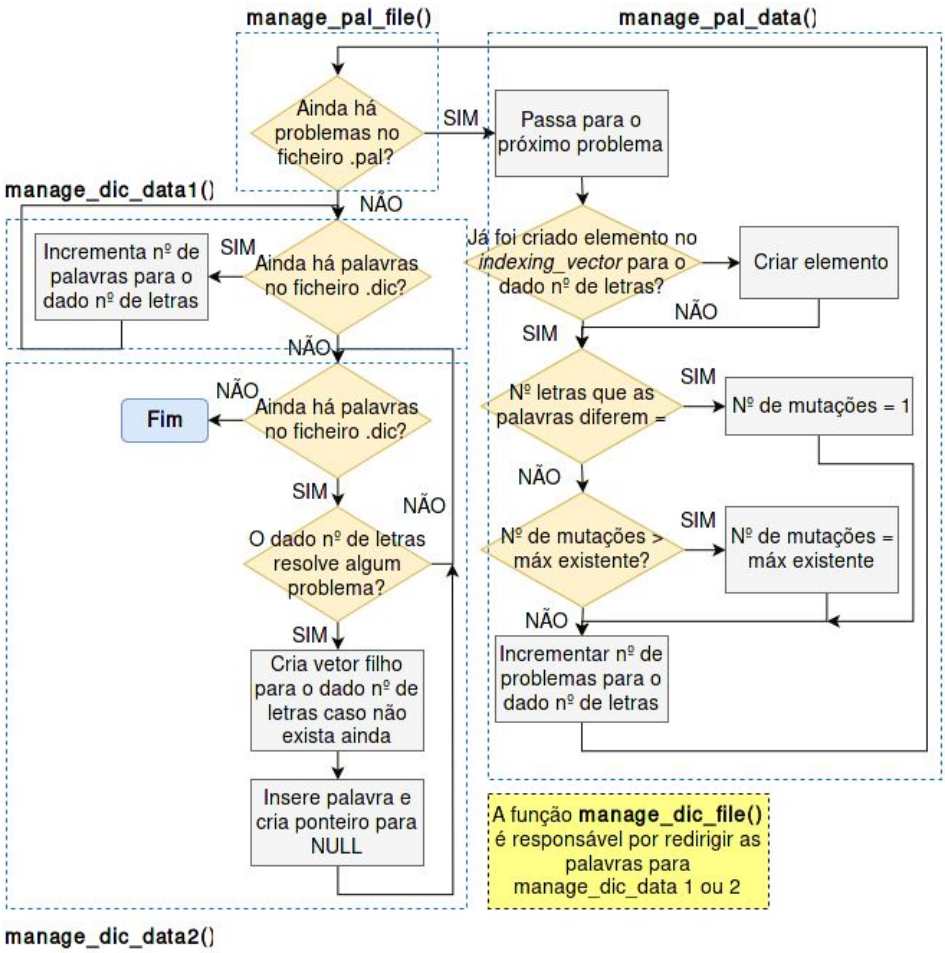


Figura 2 - Fluxograma do processamento dos ficheiros (1)

Agora é lido o ficheiro de problemas, problema a problema através da função *read_pal_file()*. Nesta parte **(2)**, a cada problema, é criado o grafo para o dado tamanho de palavra (caso ainda não exista), executado o Dijkstra e retornando um vetor de solução, *path_vector*, que contém o caminho da palavra fonte para todas as outras palavras do mesmo tamanho e o respectivo peso total. Se as palavras do problema forem iguais, o peso é 0 e não há um caminho, pelo que se passa logo para a escrita do ficheiro de solução. Se as palavras diferem numa letra só, o peso é 1 e o caminho é óbvio, passando-se também logo para a escrita do ficheiro. Na figura 3, está representado o fluxograma desta segunda parte.

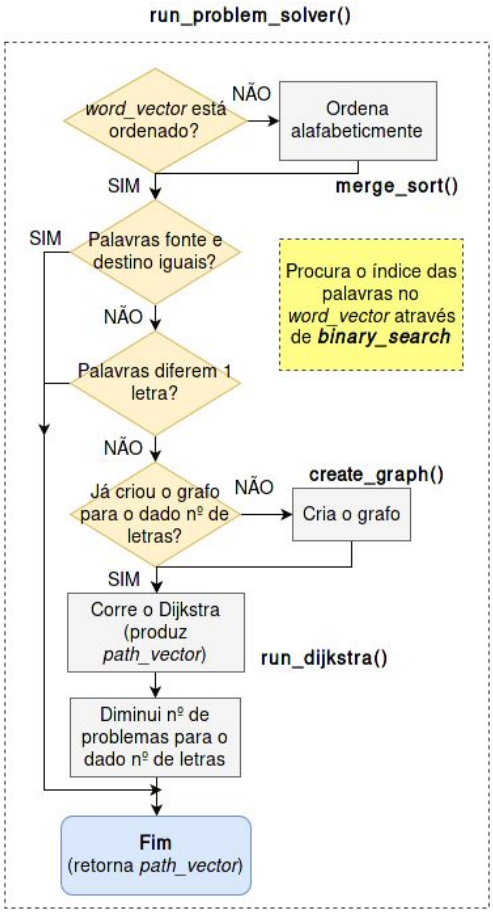


Figura 3 - Fluxograma da resolução do problema (2)

Na parte **(3)**, com o *path_vector* já criado, é escrita a solução do problema para o ficheiro de solução *.path*. De forma a escrever o caminho, é usada uma função recursiva, *print_path*, que percorre o *path_vector* desde palavra destino até à palavra fonte, e escreve as palavras no retorno. Caso o *path_vector* seja igual a NULL, ou seja, a palavra de partida e chegada forem iguais ou diferirem apenas numa letra (e consequentemente, o Dijkstra não tiver sido executado), basta escrever logo a fonte, o destino e 0 ou 1 como peso. Caso não haja caminho (i.e. a palavra destino não tiver pai), basta escrever apenas a fonte, o destino e -1. Representa-se o fluxograma desta parte na figura 4.

Na parte **(4)**, é apagado o *word_vector* (vetor que guarda todas as palavras de um determinado tamanho e os ponteiros para as listas de adjacências dessas palavras) caso já não haja mais problemas desse tamanho para resolver, de forma a evitar picos de memória. Este procedimento é feito dentro da função *write_to_file()*, uma vez que só se pode apagar este vetor após escrever as palavras do caminho do problema para o ficheiro de solução.

Finalmente **(em 5)**, são apagadas as restantes estruturas que ainda não foram apagadas, i.e. o *indexing_vector* e a estrutura que vai guardando cada problema

enquanto ele está a ser resolvido (*new_problem*). E são fechados todos os ficheiros abertos.

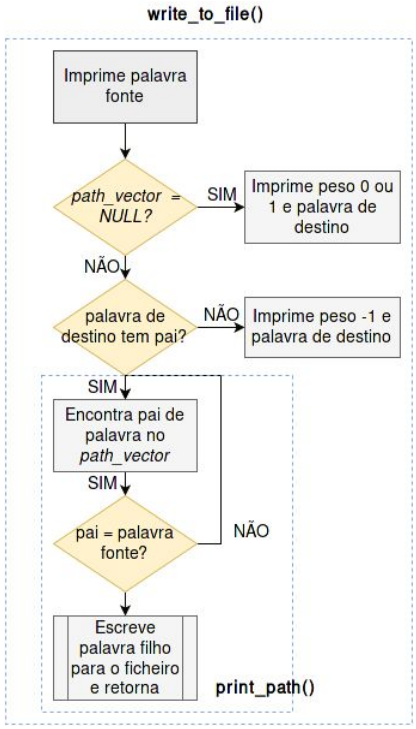


Figura 4 - Fluxograma da escrita do problema para o ficheiro(3)

Estruturas de dados

Todas as estruturas de dados encontram-se nos ficheiros *structures.c*, *storing.c* e *heap.c*, e são respectivamente, estruturas de tipos abstractos, estruturas de tipos não abstractos, e somente estruturas auxiliares à realização do Dijkstra.
As estruturas de dados presentes em *structures.c* foram criadas de forma a poder-se usar com rapidez, simplicidade e sem preocupações listas e vetores. Devido à quantidade enorme estruturas e de acessos a estruturas com que é necessário lidar durante o programa, optou-se pela maioria das estruturas não serem abstratas para diminuir o tempo de execução e a memória gasta, no entanto mantiveram-se as funções para estruturas abstratas que não estão a ser utilizadas (listas) para fins de prototipagem e acessibilidade.

Para **guardar o dicionário**, e mais tarde habilitar o programa de um **grafo** (lista de adjacências), tem-se a seguinte estrutura que pode ser representada intuitivamente pela figura 5, onde está simulada a existência de um dicionário de palavras de tamanho 1. Tendo-se em atenção os nomes das estruturas de dados na figura, explicam-se a partir daqui as mesmas.

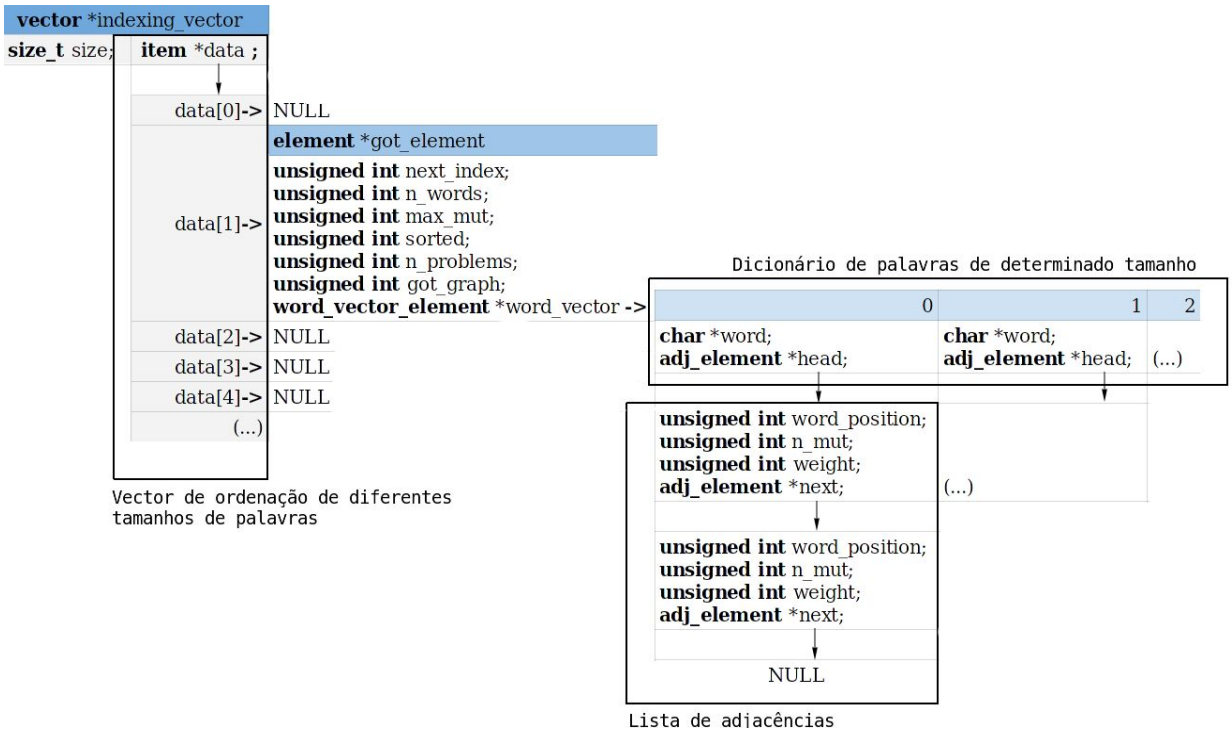


Figura 5 - Representação intuitiva do dicionário

O tipo de estrutura *vector* contém sempre uma variável do tipo *size_t* usada para contar o nº de elementos do vetor e uma outra variável que é responsável por guardar um vetor de ponteiros para *NULL*. Optou-se por usar este tipo para o *indexing_vector* porque inicialmente não se sabe a quantidade de dicionários de cada tamanho de palavra que se espera, logo em termos de memória é melhor guardar 100 ponteiros para *NULL*, aos quais a alguns é atribuída mais tarde uma estrutura de dados para apontar, do que 100 estruturas do tipo *element*.
O tipo de estrutura *element* contém 6 tipos inteiros sempres positivos (*unsigned int*):

next_index	Próximo índice do word_vector livre onde inserir uma nova palavra
n_words	Nº de palavras de determinado tamanho
max_mut	Nº de mutações máximas pedidas nos problemas para um dado tamanho de palavra
sorted	1 - Dicionário ordenado alfabeticamente, 0 - Caso contrário
n_problems	Nº de problemas para um determinado tamanho de palavra
got_graph	1 - Grafo está criado para um dado tamanho de palavra, 0 - Caso contrário

e uma outra variável do tipo *word_vector_element* que guarda um vetor do tamanho *n_words*, que contém em cada elemento uma *string* do tamanho da palavra recebida, e um ponteiro directo para a cabeça da lista de adjacências (do tipo *adj_element*). Este vetor é que contém todas as palavras de um determinado tamanho de palavra,

optou-se por não ser um tipo abstracto devido a serem feitos imensos acessos ao mesmo quando se cria o grafo e se realiza o Dijkstra como se explicou anteriormente.
Cada nó da lista de adjacências (tipo não-abstracto), contém 3 inteiros sempre positivos (*unsigned int*):

word_position	Posição da palavra, que é adjacente, no vetor que contém todas as palavras de um dado tamanho.
n_mut	Nº de mutações entre a palavra adjacente e a palavra que "contém" a lista de adjacências
weight	Nº de mutações ao quadrado (SQUARE(n_mut))

e um ponteiro para o próximo elemento da lista.

Para **guardar os problemas**, é usada uma estrutura do tipo *pal_problem* que vai sendo atualizado cada vez que chega um novo problema. A estrutura contém as seguintes variáveis:

char word1[100]	Primeira palavra do problema - palavra fonte
char word2[100]	Segunda palavra do problema - palavra de destino
unsigned int position1	Posição da palavra fonte no dicionário
unsigned int position2	Posição da palavra de destino no dicionário
int typeof_exe	Nº de mutações máximo que se pode fazer para dado problema

As variáveis *word1* e *word2* são ponteiros para 100 tipos *char*, uma vez que o nº máximo de letras que se pode encontrar numa palavra é 100.

Para **executar o Dijkstra**, é usado um vetor para o acervo, *heap_vector*, um vetor *hash_table* e um outro para os caminhos e pesos, *path_vector*.

O ***heap_vector*** é uma fila prioritária/acervo em que o elemento de menor peso é o pai e cada pai tem até 2 filhos. Formalmente, consiste num vetor com n+1 estruturas do tipo *heap_element*, em que **n é o nº de palavras de um dicionário de dado tamanho de palavra**, uma vez que é necessário o vetor começar em 1 para que seja possível determinar a posição dos elementos pai ou filho no vetor.
O tipo *heap_element* contém 2 inteiros, o *dic_index* que guarda a posição do dicionário onde está determinada palavra, e o *weight* que contém o nº de mutações entre uma determinada palavra e a palavra fonte do problema ao quadrado.
Optou-se por um acervo binário de forma a diminuir a complexidade/tempo de execução do algoritmo de Dijkstra, relativamente à implementação original.

A ***hash_table*** é um simples vetor de inteiros de tamanho n, que faz corresponder a cada índice do dicionário de dado tamanho de palavra, o índice do *heap_vector* no qual se encontra um determinado índice do dicionário (determinada posição de palavra),

para que quando se executar o Dijkstra não seja preciso procurar iterativamente a posição de uma palavra no *heap_vector*.

O ***path_vector*** é um vetor com n estruturas do tipo *path_element*, que contêm o peso total (int *total_weight*) de uma dada palavra à palavra fonte e o pai (int *parent*) dessa dada palavra que a permite chegar até à palavra fonte com um determinado peso total. Este vetor é depois usado para escrever o caminho no ficheiro de solução.

Descrição dos algoritmos

Os algoritmos do programa podem dividir-se em algoritmos relacionados com a manipulação do grafo, algoritmos relacionados com manipulação de acervos, o Dijkstra e o algoritmo para escrever o caminho para o ficheiro de solução.

Criação e Manipulação do grafo

- Criar o grafo

Para criar o grafo é usado um algoritmo que é executado pela função *create_graph()*. Consiste em 2 ciclos iterativos, um dos ciclos percorre todo o *word_vector* (vetor que contém todas as palavras de dado tamanho), e por cada elemento é executado o outro ciclo desde o elemento imediatamente à frente até ao fim do vetor, comparando os dois elementos. Se o nº de mutações entre as palavras dos dois elementos for menor ou igual que o máximo do nº máximo de mutações dos problemas desse tamanho (valor retirado ao ler o ficheiro de problemas), então são palavras adjacentes. Como tal, é criado um nó na lista de adjacências de cada palavra com o índice do *word_vector* da palavra adjacente.

- "Merge Sort"

Este algoritmo está implementado para ordenar alfabeticamente o dicionário de um dado tamanho de palavra. Para fazer esta ordenação, é criado um vetor auxiliar de N *strings,* que contém todas as palavras de um certo tamanho (palavras de um *word_vector*). O vetor é dividido em subgrupos partindo da palavra a meio do vetor de forma recursiva, até se obter N subgrupos cada 1 com uma palavra . Por fim combina-os repetitivamente de forma a produzir um único vetor de palavras alfabeticamente ordenadas. As palavras do *word_vector* são substituídas pelas palavras ordenadas do vetor temporário.

- "Binary Search"

O algoritmo (implementado iterativamente) permite encontrar a posição de uma determinada palavra no dicionário de um dado tamanho de palavra. Tendo as palavras do vetor ordenadas, este algoritmo começa à procura da palavra pretendida no meio do vetor. Se a palavra do meio, for alfabeticamente maior do que a que pretendemos encontrar, continua a procura a partir do meio da metade esquerda do vetor, caso contrário continua a partir do meio da metade direita. Este processo acontece repetidamente até se encontrar o elemento pretendido.

Dijkstra e Manipulação de acervos

- Ordenação de acervos - “Heapify”

Servindo para a ordenação de acervos, este algoritmo torna um “acervo” desordenado num acervo, de cima para baixo. O *heapify* analisa desde os nós-pais de geração mais avançada que ainda têm filhos até à raiz do acervo, i.e. desde metade do vetor até ao ínicio do mesmo. Estando num determinado nó-pai, são analisados os filhos de modo a que o peso do nó-pai seja menor que os dos nós-filhos. Caso seja maior, troca o menor nó-filho com o nó-pai e analisa a condição de acervo do novo nó-filho. Caso contrário, nada acontece e passa a analisar o elemento do vetor que é imediatamente anterior. Este processo repete-se até à raiz.

- Encontrar o caminho - “Dijkstra”

Este é o algoritmo principal do programa, é através dele que se obtém o vetor que contém o caminho entre a palavra fonte e todas as outras palavras que lhe são adjacentes, e de onde é depois retirado o caminho até à palavra de destino. Está a ser executado pela função *run_dijkstra()*, e tendo uma implementação baseada em acervos como filas de prioridade, começa por criar os 3 vetores, *heap_vector*, *hash_table* e *path_vector*, cujo conteúdo foi explicado na secção anterior. Os vetores são atualizados de forma a conterem o elemento de menor peso como pai, i.e. a palavra fonte, cujo o pai é ela própria e o custo é 0, e o resto dos elementos a custo infinito e orfãos (-1).

Seguidamente, inicia-se o ciclo no qual é realmente executado o *Dijkstra*: é retirado o primeiro elemento da fila prioritária (*heap_vector*), e é percorrida a sua lista de adjacências elemento a elemento até ao fim. Caso o nº de mutações entre uma palavra do elemento da lista e a palavra que “segura” a lista for menor ou igual ao nº de mutações máximas do problema, então é testado se existe ou não uma caminho melhor por essa palavra, através da função *find_better_path*. Nesta última função, o programa verifica se o custo total atual da palavra adjacente (até à palavra fonte) é maior que a soma do custo total da palavra atual (palavra que “segura” a lista) com o custo da palavra adjacente até à palavra atual. Caso sim, então foi descoberto um melhor caminho, e os vetores são atualizados de acordo, passando a fila prioritária a ter o novo elemento de menor peso à frente. Após a lista de adjacências ter sido toda percorrida, é retirado novamente o primeiro elemento da fila e repetido o processo, até a fila acabar.

De forma, a evitar a procura da posição dos elementos adjacentes no *heap_vector*, é utilizada a *hash_table*, tornando a complexidade do problema muito menor.
Uma vez que, há momentos em que elementos na fila tem o mesmo peso, e o programa tem de escolher um para analisar os adjacentes, implementações semelhantes produzem resultados diferentes, de acordo com a forma como está construído o grafo e implementado o Dijkstra. Na figura 6, encontra-se o fluxograma deste algoritmo.

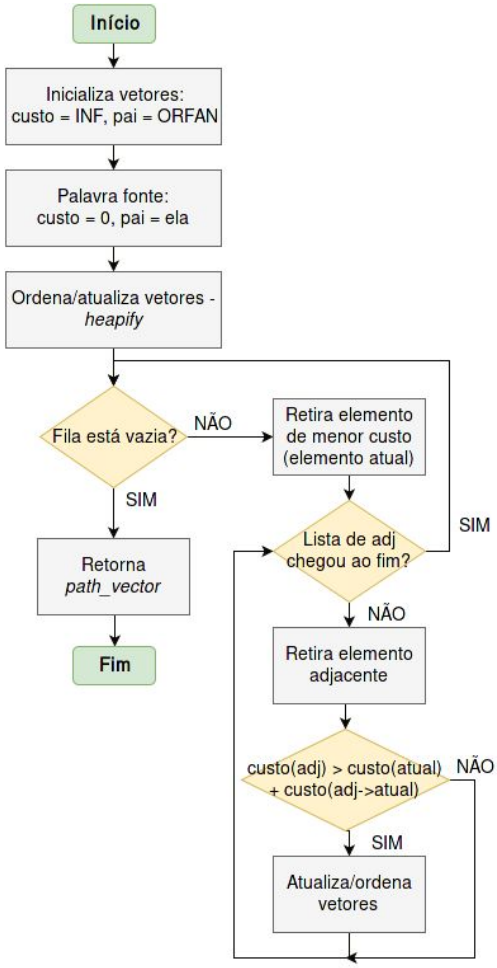


Figura 6 - Fluxograma do algoritmo para encontrar o caminho de menor custo

Subsistemas funcionais do programa

O programa apresenta diversos subsistemas onde estão organizadas um conjunto de funções a fim de resolver ou simplificar uma parte do problema proposto. Estas funções estão divididas em vários ficheiros. No ficheiro *defs.c* estão presentes definições generalistas utilizadas ao longo de todo o programa.

files.c

Este ficheiro contém todas as funções responsáveis pela leitura, criação e tratamento de dados de ficheiros. Contém um *header* - *files.h*.

Funções:

- *void manage_pal_data(char *word, vector *indexing_vector, int typeof_exe)*

Esta função realiza a leitura do ficheiro *.pal*, que tem como objetivo saber o número de problemas que existem no ficheiro para cada tamanho de palavra, definindo os elementos do *indexing_vector* que irão potencialmente resolver problemas, e guardar

o máximo dos números de mutações máximas de cada problema de um dado tamanho de palavra (caso o nº de mutações entre as palavras do problema for maior que 1).

NOTA: O inteiro *typeof_exe* corresponde ao nº de mutações máximas pedidas para o problema a ser analisado.

- *void manage_pal_file(char *file, vector *indexing_vector)*

Esta função lê o ficheiro *.pal* uma vez e envia os problemas para *manage_pal_data.*

- *int read_pal_file(FILE* pal_file, pal_problem *new_problem)*

Esta função lê um problema e envia-o para a função *set_problem_variables()*. *S*e o ficheiro de problemas não estiver terminado a função retorna 1, caso contrário retorna 0.

- *void manage_dic_data1(item got_char, item got_vector)*

Função que lê o ficheiro do dicionário e verifica quantas palavras existem para cada número de letras de forma a saber quanta memória alocar.

- *void manage_dic_data2(item got_char, item got_vector)*

Esta função coloca cada palavra que recebe na memória que foi alocada para ela na função anterior.

- *void manage_dic_file(char *file, void (*manage_dic_data)(item, item), vector *indexing_vector)*

Esta função abre o ficheiro .dic e envia as palavras para uma determinada função que lhe é passada como argumento. Primeiro é o *manage_dic_data1* (para ver a quantidade de memória necessária para colocar as palavras) e depois é o *manage_dic_data2* (para colocar as palavras).

- *char *create_output_filename(char *pal_filename)*

Muda a extensão do ficheiro de entrada para *.path* e retorna o nome do ficheiro de saída.

- *void print_path(word_vector_element *word_vector, FILE *aux_file, int src_index, path_element *path_vector, int i)*

Esta função imprime o caminho das palavras no ficheiro de saída.

- *void write_to_file(vector *indexing_vector, pal_problem *new_problem, FILE *output_file, path_element *path_vector)*

Esta função escreve tudo o que é necessário no ficheiro de saída. Verifica se existe um caminho gerado (*path_vector* diferente de *NULL*) e manda imprimí-lo caso exista, caso contrário imprime apenas a palavra de partida seguida de -1, 0 (no caso das palavras serem iguais) ou 1 (no caso as palavras só diferirem 1 mutação) e da palavra de chegada.

Execution.c

Este ficheiro inclui todas as funções encarregues de manipular as palavras obtidas dos ficheiros *.dic* e *.pal* seja em termos de procura, ordenação ou de encontrar o caminho entre elas através do *Dijkstra*. Contém um *header* - *execution.h*.

Funções:

- *int binary_search(word_vector_element *got_word_vector, int left, int right, char *word)*

Esta função realiza o algoritmo *binary search*, retornando o inteiro pretendido. Esta procura já se realiza com as palavras ordenadas alfabeticamente devido ao algoritmo de ordenação *merge sort.*

- *void merge(word_vector_element *got_word_vector, int left, int mid, int right)*

Função que realiza a ordenação alfabética das palavras, dividindo o vetor que as contém em grupos, ordenando-os separadamente e depois juntando-os (*merge*).

- *int check_number_of_mutations(char *word1, char *word2, int max_mut, int *n_mut)*

Função que verifica o número de mutações entre duas palavras. Caso esse número exceda o máximo possível para o problema a função retorna 0, caso contrário retorna 1.

- *void create_graph(element *got_element)*

Função que cria um grafo entre palavras, criando uma lista de adjacências que inclui para cada palavra, todos índices das palavras desse tamanho cujo nº de mutações em relação à palavra que "segura" a lista é menor ou igual que o máximo do nº de mutações máximas dos problemas.

- *path_element *run_dijkstra(element *got_element, int src_index, int max_mut)*

Função que corre o algoritmo *Dijkstra*. Inicializa todos os elementos do acervo a custo infinito e sem elementos pai (órfãos), excepto a palavra de partida que tem custo 0 e pai ela própria. Executa a função *heapify* para ordenar o acervo de acordo com os seus pesos/custos e executa o *Dijkstra* de forma a encontrar o caminho de menor custo entre a palavra de partida e a palavra de chegada. Retorna o vetor com os menores caminhos da palavra fonte para todas as outras.

- *path_element *run_problem_solver(pal_problem *new_problem, vector *indexing_vector)*

Função que recebe os problemas e que os resolve. Encarrega-se de procurar os índices das palavras de partida e de chegada, verificando se as palavras são iguais ou se diferem apenas numa mutação (em ambos os casos retornando imediatamente o vetor a NULL). Caso nenhumas dessas situações aconteça, constrói o grafo e corre a função

run_dijkstra, retornando o vetor com os menores caminhos da palavra fonte para todas as outras. Faz também a atualização dos problemas que já foram resolvidos.

Estes subsistemas acima descritos utilizam estruturas de dados presentes em outros 3 subsistemas/ficheiros: *structures.c* e *storing.c*, cujas funções são auto-explicativas e *heap.c*, explicado abaixo.

Heap.c

Este ficheiro contém todas as funções auxiliares ao algoritmo *Dijkstra,* e funções relacionadas com os acervos. Contém um *header* - *heap.h*.

Funções:

- *int *create_hash_table (int n_elements)*

Função que cria uma *hash table* que tem como objetivo ligar a posição das palavras do dicionário guardadas no *word_vector* através do *indexing_vector* à posição das palavras no *heap_vector* de forma a evitar procurar onde se encontram determinadas posições de palavras do dicionário no mesmo. Retorna o vetor criado.

- path_element *create_path_vector(int n_elements)

Função que cria o *path_vector* sendo este o vetor com os caminhos e pesos totais entre cada palavra e a palavra fonte. Retorna o vetor criado.

- *heap_element *create_heap_vector(int n_elements)*

Função que cria o *heap_vector* sendo este vetor um acervo que contém a posição das palavras no dicionário e o peso de cada palavra em relação à palavra fonte. Retorna o vetor criado.

- *void initialize_heap(int size, int *hash_table, path_element *path_vector, heap_element *heap_vector)*

Função que inicializa todos os vetores com peso infinito e pai -1 (orfão).

- *int get_first_heap_dic_index(int *hash_table, int size, heap_element *heap_vector)*

Esta função retorna o primeiro elemento da fila prioritária, posicionando-o no fim do vetor *heap_vector*. Atualiza a *hash_table* de acordo e repõe a condição de acervo novamente através da função *heapify()*.

- *void heapify(int i, int size, int *hash_table, heap_element *heap_vector)*

Esta função converte um "acervo" desordenado num acervo, que é uma árvore binária organizada de acordo com os pesos entre as palavras. Nesta estrutura as ligações são por ordem crescente de peso.

- *void find_better_path(path_element *path_vector, int curr_heap_dic_index, adj_element *node, int *hash_table, heap_element *heap_vector)*

Esta função trata de analisar os caminhos e atualizá-los com outros caso sejam mais vantajosos. Actualiza o acervo e a *hash_table* recorrendo à função *swap_heaps()* e *swap_hash_values()*, respetivamente, sempre que for necessário trocar algum elemento.

Análise dos requisitos computacionais

Pré processamento e armazenamento do dicionário

Inicialmente, é feito **pré-processamento dos ficheiros**, isto implica percorrer 1 vez o ficheiro de problemas (que inclui ver o nº de mutações entre as palavras de cada problema) e 2 vezes o ficheiro de dicionário, uma para ver o nº de palavras de cada tamanho e outra para adicioná-las a estrutura criada. **Adicionar à estrutura é um processo com complexidade O(1)**, porque é guardado o próximo lugar livre quando se insere um elemento. A **memória para guardar o dicionário inicial (sem lista de adjacências)** é proporcional ao nº de palavras de cada tamanho de palavra e ao seu tamanho em si (porque se alocam dinamicamente as palavras), O vetor-mãe que guarda 100 ponteiros para os vetores-filhos, na sua maioria nulos (por não existirem problemas desse tamanho), não tem grande influência.

Seguidamente, é lido o ficheiro de problemas de novo, problema a problema. A cada problema:

Merge Sort

Se o vetor-filho para o tamanho de palavra ainda não estiver ordenado alfabeticamente, será ordenado. Este processo independentemente da situação do vetor vai sempre dividir e juntar as palavras, por isso em qualquer caso tem **complexidade de O(Nlog(N))** e **complexidade de espaço O(N)** (em que N é o nº de palavras no vetor). Esta memória ocupada é apenas temporária e ocorre uma vez por cada tamanho de palavra a resolver.

Binary Sort

As posições das palavras de fonte e destino são agora procuradas no vetor-filho. Com *Binary search* a complexidade da procura é da ordem de **O(logN)**, em que N é o tamanho do vetor.

Caso as palavras fonte e destino sejam diferentes ou tenham mais do que uma mutação, então é criado o grafo e executado o Dijkstra.

Construção e manipulação do grafo

Caso o grafo ainda não tenha sido construído para o dado tamanho de palavra, então é construído com complexidade proporcional a $\mathbf{O(N^2)}$ e complexidade espacial proporcional ao nº de adjacentes a cada palavra, que varia de acordo com o máximo de mutações com o qual é criado o grafo.
Neste processo, a comparação dos caracteres das palavras é parada, caso o número de mutações já exceda o máximo, diminuindo em muito o tempo de execução do programa. É também nesta fase que é calculado o quadrado do nº de mutações. Ambos estes cálculos são efectuados com *macros*, o que ajuda a diminuir o tempo.

Durante todo o programa nunca é necessário aceder a um elemento em específico das listas, pelo que a complexidade para aceder ao primeiro ou ao próximo elemento da lista é **O(1)**.

De seguida é executado Dijkstra:

Dijkstra e Manipulação de Acervos

Inicialmente, são criados os 3 vetores (2 deles com V elementos, e outro com V+1, em que V é o nº de palavras do vetor-filho), em termos de complexidade temporal e espacial esta operação é proporcional a **O(V)**. A única memória a utilizar será esta.
A seguir é feita a ordenação do acervo de acordo com as suas prioridades, esta operação tem complexidade temporal **O(V)**. Posteriormente, o custo de ordenação será apenas O(log(V)).
São agora feitas V iterações até chegar ao fim da fila prioritária, a cada iteração é: retirado o primeiro elemento da fila - O(1) -, reposta a condição de acervo - O(log(V)) - e analisada a lista de adjacências com M elementos, cada elemento exposto ao processo de ordenação para cima - O(log(V)).
Este processo teria complexidade proporcional a O(V(log(V) +M)) = O(Vlog(V)+ VMlog(V)), em que VM é o nº total de adjacentes/arestas (E), ficando assim **O(Vlog(V)+ Elog(V))**. No entanto, o tempo de execução varia muito consoante o nº de arestas, que variam consoante o máximo de mutações pedidas para o problema. Isto é, são formados *clusters* aos quais só pertecem determinadas arestas que cumprem o nº máximo de mutações. Quanto menor o *cluster*, melhor o tempo de execução, uma vez a complexidade passa a ser $\mathbf{O(Vlog(V) + E_{\text{fora do cluster}} + E_{\text{no cluster}}log(V))}$.

Escrita do ficheiro de solução e libertação de estruturas

Para a escrita do ficheiro quando há um caminho, é feita uma procura recursiva desde a palavra de destino até à palavra fonte, e impresso o caminho no retorno. Isto tem uma complexidade temporal e espacial proporcional ao nº de palavras no caminho.
Se não houver mais problemas do dado tamanho, então é apagado grafo do mesmo, o que em complexidade temporal é proporcional a E (nº total de arestas).

Análise Crítica

Ao submeter o programa no *mooshak* foram passados 19 testes, sendo o erro do último "Limite de memória excedida", que acabou por não ser investigado. Caso fosse resolvível por diminuir a memória em pequenas doses (que quase de certeza que não era), podia-se por exemplo diminuir os 64 *bits* dos inteiros para 32.
Apesar de não se ter tido problemas em relação ao tempo de execução, apresentam-se a seguir algumas ideias que poderiam ter melhorado a *performance* do programa se tivessem sido implementadas, mas dada a vontade para passar às restantes disciplinas e a falta de tempo, não foram.

De forma a diminuir o tempo de execução de algumas funções que realizam cálculos, foi usado ***macros***, por exemplo, a calcular o quadrado do nº de mutações, somas durante o Dijkstra e comparações das palavras durante a construção do grafo. Se fosse usado em mais sítios, possivelmente o tempo de execução melhoraria.

Em problemas que não tem solução, podia-se determinar imediatamente isso através de um **algoritmo da conectividade**, em vez de correr integralmente o *Dijkstra* para no fim descobrir que não tem caminho. No entanto, há imensa incerteza em relação a se realmente melhoraria ou não.

Em vez de se usar *binary heaps*, poderia-se ter utilizado ***Fibonnaci Heaps***, o que diminuiria a complexidade do *Dijkstra* dado que a complexidade do processo de ordenação para cima no acervo é O(1).
Fazer um **grafo incremental**, ou seja, um grafo que aumenta o nº de elementos nas listas de adjacências consoante o nº de mutações máximas dos problemas que aparecem, melhoraria também o programa porque ao analisar as palavras adjacentes, algumas das vezes não se fariam iterações inúteis nas listas de adjacências a elementos não adjacentes.

Exemplo de aplicação

De forma a demonstrar um exemplo completo e detalhado, é usado o dicionário e problemas a resolver, presentes na figura 7.

.dic	.pal
aio	*aio tal 3*
bata sopa	*aio tal 1*
nao	*sopa sopa 2*
bola	*bola roxo 2*
rola tal	
roxa	
roxo	
sequencia	

Figura 7 - *indexing_vector* após leitura do ficheiro de problemas

1. É criado um vetor de 100 ponteiros nulos (*indexing_vector*).
   indexing_vector = *create_vector(101);*

2. É lido o ficheiro de problemas, do qual se retira que elementos do *indexing_vector* (com um determinado nº de letras) serão usados para resolver os problemas, quantos problemas existem e o nº máximo de mutações entre palavras para esses elementos (caso as mutações entre as palavras do problema sejam maiores que 1), como está representado na figura 8.
   manage_pal_file(argv[2], indexing_vector);

0 ->	NULL
1 ->	NULL
2 ->	NULL
3 ->	n_problems = 2 max_mut = 3
4 ->	n_problems = 2 max_mut = 2
(...) ->	NULL
9 ->	NULL
(...) ->	NULL
100 ->	NULL

Figura 8 - *indexing_vector* após leitura do ficheiro de problemas

3. É lido o ficheiro do dicionário de forma a retirar o nº de palavras a alocar para cada elemento, tal como se mostra na figura 9.
manage_dic_file(argv[1], manage_dic_data1, indexing_vector);

0 ->	NULL
1 ->	NULL
2 ->	NULL
3 ->	n_problems = 2 max_mut = 3 n_words = 3
4 ->	n_problems = 2 max_mut = 2 n_words = 6
(...) ->	NULL
9 ->	NULL
(...) ->	NULL
100 ->	NULL

Figura 9 - *indexing_vector* após leitura do ficheiro de dicionário

4. É lido novamente o ficheiro de dicionário de modo a inserir as palavras, ficando o mesmo como se representa na figura 10.
manage_dic_file(argv[1], manage_dic_data2, indexing_vector);

		0	1	2	3	4	5
0 ->	NULL						
1 ->	NULL						
2 ->	NULL						
3 ->	n_problems = 2 max_mut = 3 n_words = 3 **word_vector ->**	aio	nao	tal			
4 ->	n_problems = 2 max_mut = 2 n_words = 6 **word_vector ->**	bata	sopa	bola	rola	roxa	roxo
(...) ->	NULL						
9 ->	NULL						
(...) ->	NULL						
100 ->	NULL						

Figura 10 - *indexing_vector* após leitura do ficheiro de dicionário pela segunda vez

5. É agora lido novamente o ficheiro de problemas, mas desta vez problema a problema.
while(read_pal_file(pal_file, new_problem))
A cada ciclo é guardado na estrutura "*new_problem*" o respetivo problema para resolver. O primeiro problema é *"aio tal 3"*, como tal a estrutura fica como representado na figura 11.

pal_problem *new_problem
word1 = aio word2 = tal position1 = -1 position2 = -1 typeof_exe = 3

Figura 11 - estrutura *new_problem* para o problema "*aio tal 1*"

6. Para este dado nº de letras (3), é ordenado o dicionário (caso ainda não tenha sido ordenado). Ficando como se mostra na figura 12.

3 ->	n_problems = 2 max_mut = 3 n_words = 3 **word_vector ->**	aio	nao	tal

Figura 12 - *word_vector* para palavras com 3 letras após ser ordenado

7. São procurados os índices do dicionário correspondentes às palavras de fonte (*src_index = 0*) e destino (*dest_index = 2*). E atribuídos, respectivamente, à *position1* e à *position2* da estrutura *new_problem*.

8. Caso a palavra fonte e a palavra de destino fossem iguais, o programa passaria logo para a escrita da solução no ficheiro. Neste caso não o são, como tal é seguidamente criado o grafo (caso este ainda não exista para as palavras com 3 letras), tendo em conta que o nº máximo de mutações entre as palavras é 3. O grafo fica então a estrutura da figura 13.
create_graph(got_element);

		0	1	2
3 ->	**word_vector ->**	aio	nao	tal
		word_position = 2 n_mut = 3 weight = 9 next	word_position = 2 n_mut =2 weight = 4 next	word_position = 1 n_mut =2 weight = 4 next
		word_position = 1 n_mut = 2 weight = 4 next	word_position = 0 n_mut =2 weight = 4 next	word_position = 0 n_mut = 3 weight = 9 next
		NULL	NULL	NULL

Figura 13 - Grafo para palavras com 3 letras

Simbolicamente, está representado na figura 14.

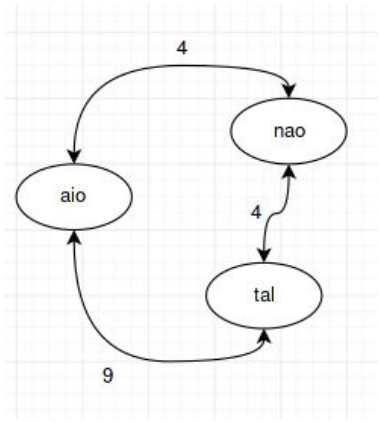


Figura 14 - grafo simbólico para palavras com 3 letras

9. É agora executada a função que constrói o acervo e corre o Dijkstra para se obter um vetor com os caminhos e pesos totais entre cada palavra e a palavra fonte, em que o nº de mutações é menor ou igual ao pedido (neste caso menor ou igual a 3).
path_vector = run_dijkstra(got_element, src_index, max_mut);

9.1. É inicializado o acervo e todas as estruturas auxilares para correr o Dijkstra (*heap_vector*, *hash_table*, *path_vector*), e é seguidamente atribuído à palavra fonte como elemento pai ela própria e peso para chegar a ela própria zero, como se mostra na figura 15.
initialize_heap(n_words, hash_table, path_vector, heap_vector);

heap_vector			
0	1	2	3
dic_index = -1	dic_index = 0	dic_index = 1	dic_index = 2
weight = INF	weight = 0	weight = INF	weight = INF
hash_table			
0	1	2	
1	2	3	
path_vector			
0	1	2	
parent = 0	parent = ORFAN	parent = ORFAN	
total_weight = 0	total_weight = INF	total_weight = INF	

Figura 15 - *heap_vector*, *hash_table* e *path_vector* após inicializados e atribuída a palavra fonte

9.2. É executado o "*heapify*", de modo a fazer do "acervo" um acervo. Por acaso, nesta situação, o primeiro elemento do *heap_vector* é a palavra fonte, que tem peso 0, como tal o acervo já cumpre as condições de acervo, por isso, os vetores são iguais aos da figura x. Simbolicamente, o acervo apresenta-se na figura 16.
heapify(i, n_words, hash_table, heap_vector);

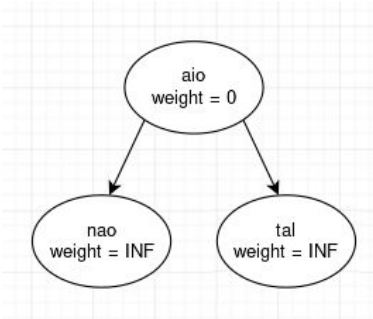


Figura 16 - Acervo simbólico

9.3. Acede-se à lista de adjacências do primeiro elemento do *heap_vector* ("aio"). O primeiro elemento da lista é "tal", como se mostra na figura 13. Agora procura-se um caminho com menos peso de "aio" para "tal" e atualizam-se os vetores, passando o novo elemento com menos peso a ser o primeiro do *heap_vector* e o que era anteriormente passa para o fim do vetor, ficando como está na figura 17.

NOTA: Para não ter de se iterar sobre o *heap_vector* para se encontrar onde está o indíce de cada elemento adjacente, é usada a *hash_table*, que a cada indíce do dicionário faz corresponde o índice do *heap_vector*, onde se encontra esse indíce do dicionário.

heap_vector

0	1	2	3
dic_index = -1	dic_index = 2	dic_index = 1	dic_index = 0
weight = INF	weight = 9	weight = INF	weight = 0

hash_table

0	1	2
3	2	1

path_vector

0	1	2
parent = 0	parent = ORFAN	parent = 0
total_weight = 0	total_weight = INF	total_weight = 9

Figura 17 - *heap_vector, hash_table* e *path_vector* após 1 iteração do Dijkstra

9.4. Ainda na lista de adjacências de "aio", procura-se agora um caminho de menor peso entre "aio" e "nao" (que é o próximo elemento da lista). O peso do caminho encontrado é 4, então "nao" passa a ser o primeiro elemento do *heap_vector*, como se mostra na figura 18.

heap_vector

0	1	2	3
dic_index = -1	dic_index = 1	dic_index = 2	dic_index = 0
weight = INF	weight = 4	weight = 9	weight = 0

hash_table

0	1	2
3	1	2

path_vector

0	1	2
parent = 0	parent = 0	parent = 0
total_weight = 0	total_weight = 4	total_weight = 9

Figura 18 - *heap_vector, hash_table* e *path_vector* após 2 iterações do Dijkstra

9.5. A lista de adjacências de "aio" não tem mais elementos, passa-se para a lista de adjacências do novo primeiro elemento do *heap_vector* -("nao"). Pela figura 13, o primeiro elemento na lista é "tal". É descoberto um caminho com menos peso (8) de "aio" para "tal": "aio->nao->tal". Atualizam-se os vetores, passando este novo elemento para o início do *heap_vector*. Como se mostra na figura 19.

heap_vector

0	1	2	3
dic_index = -1	dic_index = 2	dic_index = 1	dic_index = 0
weight = INF	weight = 8	weight = 4	weight = 0

hash_table

0	1	2
3	2	1

path_vector

0	1	2
parent = 0	parent = 0	parent = 1
total_weight = 0	total_weight = 4	total_weight = 8

Figura 19 - *heap_vector, hash_table* e *path_vector* após 3 iterações do Dijkstra

9.6. Passa-se para o próximo elemento na lista de "nao", esse elemento é "aio", não é encontrado nenhum novo caminho. Passa-se para a lista de adjacências do novo primeiro elemento do *heap_vector* ("tal"), os seus adjacentes são "nao" e "aio", não são encontrados novos caminhos. Os vetores mantém-se como está na figura 19.

9.7. Os vetores *heap_vector* e *hash_table* já não necessários, apagam-se.

10. Acabado o Dijkstra, diminui-se o nº de problemas presentes para as palavras de tamanho 3. O grafo com os caminhos de menor peso da palavra fonte "aio", para todas as outras, apresenta-se simbolicamente na figura 20.
sub_element_n_problems(got_element);

Figura 20 - Grafo com os caminhos de menor peso

11. Agora escreve-se para o ficheiro a palavra fonte, seguida do peso total (8) para chegar da mesma à palavra de destino "tal", esta informação está no *path_vector* como se mostra na figura x. Finalmente, é escrito o caminho recursivamente, andando da palavra destino "tal" até à fonte. Caso não houvesse mais problemas, apagar-se-ia o *word_vector* que contém todas as palavras de tamanho 3.
write_to_file(indexing_vector, new_problem, output_file, path_vector);
A solução deste problema é:

aio 8
nao
tal

12. É apagado o *path_vector*, e inicia-se o próximo problema "aio tal 1". Neste caso **nenhuma palavra**, do dicionário de 3 letras, **se torna noutra palavra com menos de 2 mutações**, como tal todas as palavras do grafo tem peso/distância infinita em relação a "aio", logo não há um caminho com o nº de mutações máximas pedido (1). O *path_vector* fica como se mostra na figura x, é apagado o *word_vector* para este tamanho de letras, já que não existem mais problemas. E a solução é:

aio -1
tal

13. O próximo problema é "sopa sopa 2". Neste caso, a palavra fonte e de destino são iguais, como tal nem sequer é criado o grafo, e passa-se imediatamente para a escrita do ficheiro de solução, que fica:

sopa 0
sopa

14. Passa-se para o último - "bola roxo 2". Este problema não tem nenhuma peculiaridade, a resolução é semelhante à do "aio tal 3", serve apenas para demonstrar como ficam as estruturas de dados iniciais, pelo que não é aqui feita uma resolução intensiva. Não havendo mais problemas a resolver é apagado o dicionário de palavras de tamanho 4 e é apagado o *indexing_vector*. O ficheiro *.path* está indicado na figura 21.

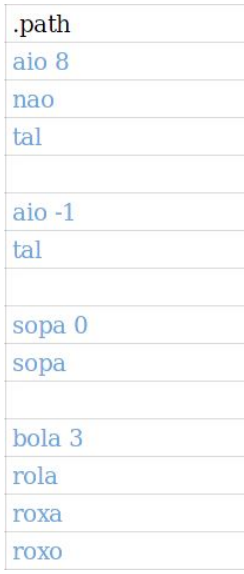


Figura 21 - Ficheiro de solução