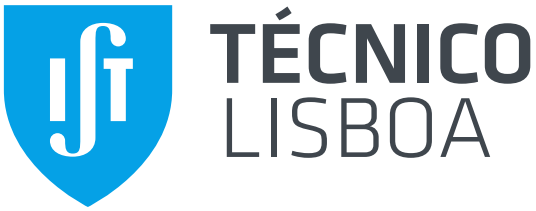

MESTRADO EM ENG. ELETROTÉCNICA E DE COMPUTADORES

ALGORITMOS E ESTRUTURAS DE DADOS

2016/2017 – 2º Ano, 1º Semestre

RELATÓRIO

WORDMORPH

Corpo docente: Carlos Bispo, Luís Silveira,

Margarida Silveira, Paulo Flores

Identificação dos Alunos:

Grupo: 007

Nome: David Ribeiro Número: 84027

Email: david.a.c.ribeiro@tecnico.ulisboa.pt

Nome: Rafael Forte Número: 84174

Email: rafael.forte@tecnico.ulisboa.pt

ÍNDICE

INTRODUÇÃO

O programa desenvolvido no âmbito deste projeto tem por objetivo perante duas palavras dadas, e com base num dicionário fornecido, calcular o caminho mais curto entre estas a partir da alteração de caracteres. O programa para além das duas palavras, recebe um inteiro com o número máximo de letras que se podem alterar de cada vez, e com base nisso apresenta um caminho ao longo de varias palavras e um custo total de todas as alterações, sendo o custo por passo quadrático ao número de letras alteradas.

Especificações do problema:

- O programa devera ler dois ficheiros, um que contem o dicionário (ficheiro .dic) e outro com os problemas a resolver (ficheiro .pal), e devera criar um ficheiro com as soluções (ficheiro .path) com o mesmo nome do ficheiro de problemas;

- Os problemas de palavras a resolver são sempre entre palavras do mesmo tamanho;

- Para a resolução dos problemas é necessário indicar o dicionário sobre o qual se quer trabalhar. As palavras indicadas no problema tem de estar no dicionário e o caminho apenas se baseara em palavras desse dicionário;

- O ficheiro que contém o dicionário apenas pode ter palavras compostas por caracteres latinos e não pode conter palavras acentuadas nem hifenadas;

- O preço das mutações é quadrático, ou seja o custo de uma mutação é igual ao quadrado do número de caracteres alterados;

- O número máximo de caracteres alterados permitido é o que vem referenciado posteriormente às duas palavras dadas para o problema;

- O ficheiro criado devera apresentar as soluções aos problemas pela mesma ordem que estes estão no ficheiro que contem os problemas, independentemente da sua ordem alfabética, tamanho, etc;

- As soluções são apresentadas da seguinte forma:

- Caso exista caminho, devera ser escrita a primeira palavra do problema, seguida do custo total do caminho obtido, posteriormente escrevem-se todas as palavras que compõem o caminho até à segunda palavra do problema, todas pela ordem segundo a qual surgem no caminho;

- Caso não exista caminho, devera ser escrita a primeira palavra do problema, seguida de "-1", e por baixo a segunda palavra do problema;

ABORDAGEM AO PROBLEMA

Para o desenvolvimento do projeto verificou-se que o caminho mais logico de se seguir seria a implementação de um grafo segundo o qual seria aplicado um algoritmo que determine os caminhos mais curtos de um certo ponto a outro especificado (fonte-destino).

Para tal efeito, para cada problema dado, calculamos uma SPT, a partir do algoritmo de Dijkstra, onde a palavra de origem é o vértice da raiz e a partir dai determinar o caminho até ao destino.

Este processo encontra-se num intervalo entre duas complexidades, pois depende do tamanho das palavras em questão e do número de mutações permitidas, podendo assim ser um grafo esparso ou denso (podendo mesmo ser completo), sendo que o algoritmo de Dijkstra tem complexidade linear para grafos densos e complexidade E*lg (V) para grafos esparsos, sendo E o número de arestas e V o número de vértices.

Em termos de memoria este projeto pode-se tornar muito dispendioso, pois em caso de grande existência de palavras em que o grafo seja denso ocupa-se muita memória. Para tentar minimizar este problema apenas se permite que exista um grafo alocado de cada vez e resolvem-se todos os problemas que necessitem deste.

ARQUITETURA DO PROGRAMA

A arquitetura do programa divide-se nas seguintes secções:

- Inicialização da estrutura dicionário, a qual guarda todas as palavras segundo as quais o programa vai trabalhar. O programa analisa o ficheiro e guarda todas as palavras, e inicializa todos os campos da estrutura.

- Alocação da lista com os problemas e registo dos mesmos na lista, neste processo é também registado o número máximo de caracteres que podem ser alterados por mutação para cada tamanho de palavras. Para além destes é também alocada a tabela de soluções.

- Ciclo de resoluções, onde se constrói o grafo para um tamanho específico e se determinam todas as soluções para os problemas desse tamanho e guardam na respetiva posição da lista de soluções.

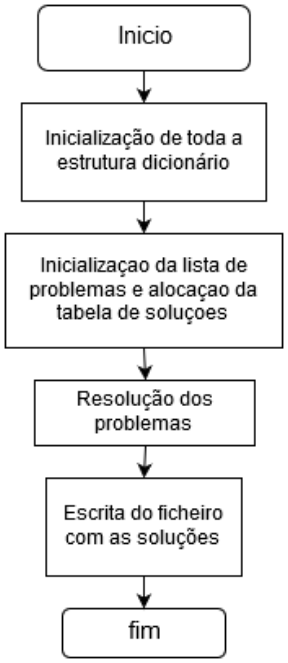

FIGURA 1 - FLUXOGRAMA DO PROGRAMA

Inicialização da estrutura dicionário

Durante este processo são feitas duas leituras ao ficheiro que contem o dicionário. Uma primeira que indica o número de palavras de cada tamanho e guarda essa informação numa tabela pertencente à estrutura dicionário. Acabada essa leitura o programa irá

alocar a memória necessária para alocar todas as palavras, num formato de matriz de palavra.

Na segunda leitura do ficheiro registam-se as palavras na memória previamente alocada. Ainda neste processo é alocada a matriz de posições. Esta matriz guarda os índices das palavras do dicionário e é inicializado com os índices da matriz de palavras iguais aos da matriz de posições.

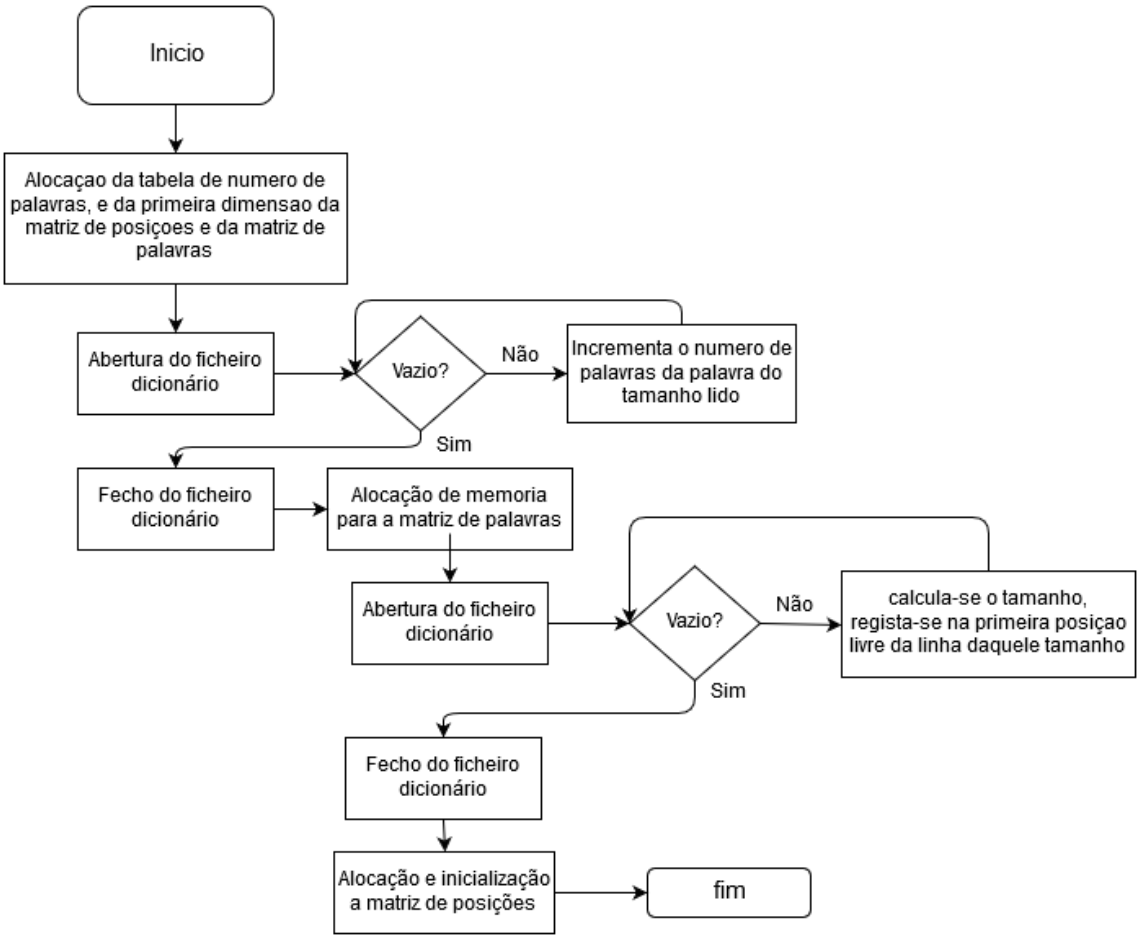

FIGURA 2 - FLUXOGRAMA DA INICIALIZAÇAO DA ESTRUTURA DICIONARIO

Inicialização dos problemas e soluções

Neste bloco é aberto o ficheiro de problemas, e sempre que se verifique a existência de um problema cria-se um no e inicializa-se o mesmo, adiciona-se ao final da lista e verifica-se se para o tamanho de palavras lido se este é o numero máximo de alterações de caracteres permitido. Esta informação estará disponível numa tabela para mais tarde se utilizar, na construção dos grafos.

Posteriormente é criada a tabela de soluções com base no número de problemas lidos.

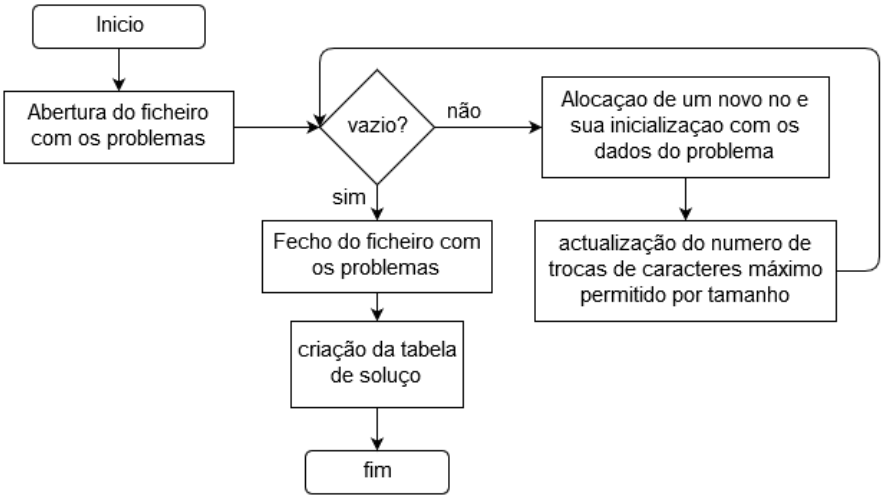

FIGURA 3 - FLUXOGRAMA DA INICIALIZAÇAO DE PROBLEMAS E SOLUÇOES

Resolução de problemas

Durante este processo verifica-se para cada tamanho de 0 a um limite imposto se existem problemas a resolver. Caso não exista o processo é simples apenas incrementando o tamanho visto e volta-se a verificar se existem problemas a resolver. Caso exista o programa ira:

- Organizar alfabeticamente todas as palavras do tamanho em causa do dicionário;

- Construir as listas de adjacências para todas as palavras referidas.

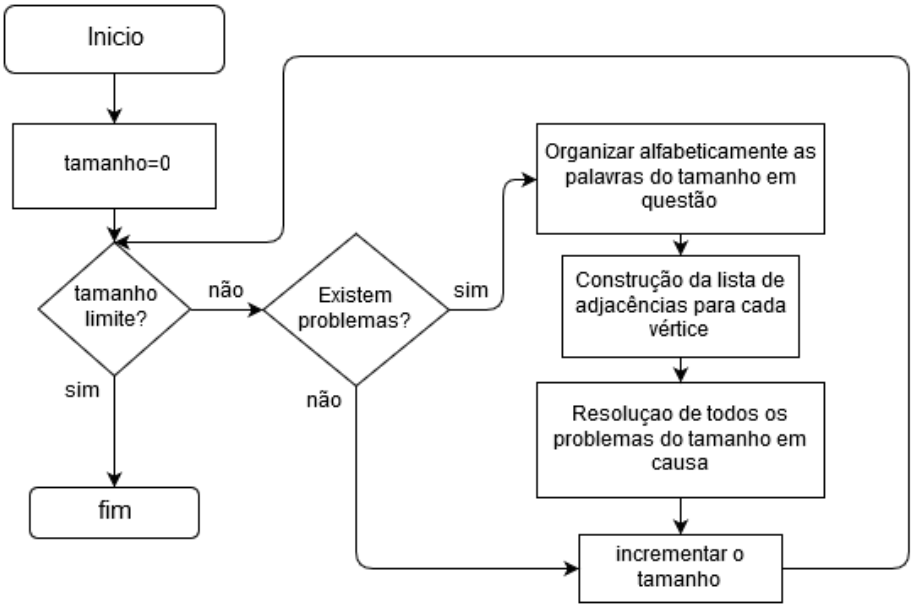

FIGURA 4 - FLUXOGRAMA PRINCIPAL DA RESOLUÇAO DE PROBLEMAS

Concluídos estes passos, será percorrida a lista com os problemas e sempre que se verificar que o problema em questão tiver o mesmo numero de letras nas suas palavras que as do grafo em questão serão percorridos os seguintes passo:

- Procura da palavra de destino do problema na matriz de palavras;

- Construção de uma árvore de caminhos mais curtos fonte-destino;

- Procura da palavra de origem do problema na matriz de palavras;

- Registo do caminho percorrido e do custo total na tabela de soluções.

Depois de percorridos todos estes passos, e depois de ter corrido toda a lista de problemas, incrementa-se o valor do número do tamanho das palavras a analisar e repete-se o processo ate que se atinja o valor previamente estabelecido como limite.

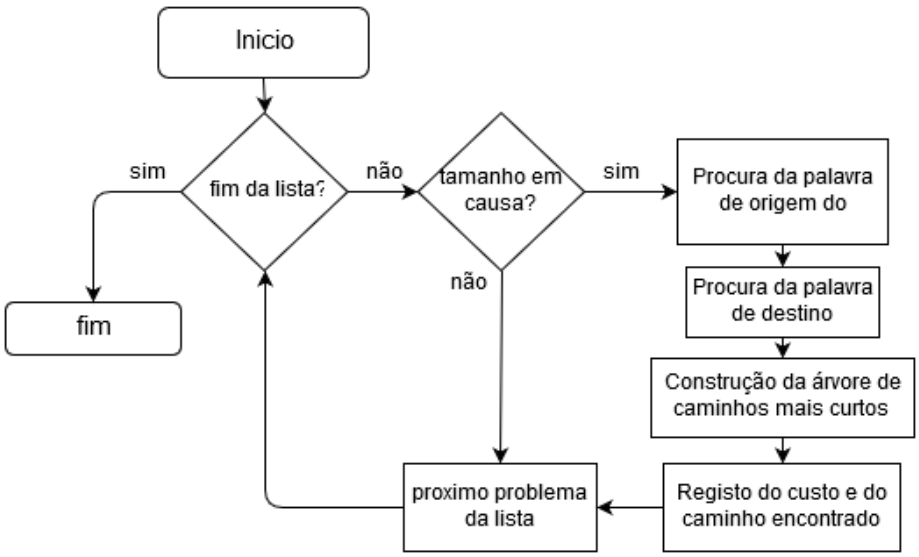

FIGURA 5 - FLUXOGRAMA DA RESOLUÇAO PARA UM DADO TAMANHO

DESCRIÇÃO DOS ALGORITMOS

Neste projeto utilizamos 4 algoritmos importantes para os cálculos e tratamento dos dados, o SelectionSort, o QuickSort, o algoritmo de Procura Binária, e o algoritmo de Dijkstra, que se encontram respetivamente nas funções: organizar_tabelas, find_position e achar_caminho.

organizar_tabelas:

De modo a ser mais rápido encontrar a palavra pretendida no dicionário (usando procura binária) era necessário ter a matriz posição (das palavras do dicionário) organizada alfabeticamente.

O QuickSort está implementado de forma recursiva, sendo chamado para ordenar as duas tabelas mais pequenas, resultantes de um QuickSort que dividiu a tabela em duas, na tabela da esquerda encontram-se apenas palavras menores que o pivô, e na tabela da direita encontram-se apenas palavras maiores que o pivô.

O algoritmo pega na última palavra da tabela fornecida (nosso pivô) e vai comparando com as palavras no início da tabela (incrementando uma variável (I) que inicialmente tem a posição mínima), quando encontrar uma palavra maior que o pivô (PalavraA) começa a comparar do fim da tabela para o início (fazendo uso de outra variável (J)) até encontrar uma palavra menor que o pivô (PalavraB), ou ter atingido o mínimo da tabela, nessa altura troca a posição da PalavraA com a posição da PalavraB.

Depois, volta a comparar palavras com o pivô, partindo do índice I, continuando a incrementar a variável I, repetindo assim o processo até I ultrapassar J. Quando I tiver o mesmo valor que J, troca-se a posição do pivô com a palavra maior que achou, e retorna o valor de I que vai ser utilizado como índice máximo da tabela à esquerda de I (max=I-1), e como mínimo da tabela à direita de I (min = I+1).

Ao executar este processo sucessivamente em tabelas mais pequenas (no caso ótimo tabelas de metade tamanho da inicial) chegaremos ao caso de tabelas de dimensão 2, que ficarão ordenadas. Como o implementamos recursivamente, quando o QuickSort começa a retornar vai ordenando as tabelas consecutivamente maiores.

No fim temos a tabela de posições das palavras ordenada (por ordem alfabética), sendo que no pior caso (caso a tabela já se encontra-se ordenada) o programa executa N^2 comparações, e no melhor caso (tabela é sempre dividida a meio) N*lg(N) comparações.

Contudo para tabelas pequenas optou-se por utilizar o SelectionSort uma vez este ser mais eficaz para esses casos. Este analisa as N palavras, e coloca a menor na primeira posição, de seguida analisa as restantes N-1 palavras e coloca a menor na posição seguinte (2), e assim sucessivamente N vezes.

Escolheu-se o QuickSort (para tabelas grandes) uma vez que em média realiza 2N*lg(N) operações, sendo N o numero de elementos a organizar.

find_position:

Após se ler o problema é necessário procurar as posições das palavras de partida e destino no dicionário e como esta operação se repete múltiplas vezes torna-se impensável usar o método da procura linear, mesmo que para tal se tenha de organizar as tabelas (o que tem um custo superior à procura linear).

Este algoritmo consiste em comparar a palavra do problema com a palavra que se encontra a meio da tabela. Caso seja igual, já temos a posição onde ela se encontra, se for menor que à palavra a meio da tabela, como esta está ordenada, sabemos que encontra-se na parte inferior da tabela, se for maior encontra-se na parte superior da tabela . De seguida é repetir o processo apenas para a parte onde se encontra a palavra (parte superior ou inferior).

Este processo acaba quando a palavra a meio da "nova tabela" for igual à palavra do problema, retornando a sua posição, ou quando já não for possível reduzir mais o tamanho da "nova tabela" (índice que seria o máximo desta tabela é inferior ao que seria o mínimo) retornando "Palavra inexistente".

Este algoritmo executa no pior dos casos lg(N) comparações, sendo N o número de elementos da tabela, uma vez que a tabela resultante tem sempre metade do tamanho da anterior.

achar_caminho:

Tendo já as listas de adjacências das palavras de tamanho X, para se achar o caminho mais curto (menor custo/peso) entre duas palavras precisamos de uma arvore de caminhos mínimos fonte-destino, para tal usamos o algoritmo de Dijkstra, o qual tem complexidade E*lg(V), em que V corresponde ao numero de vértices e E ao numero de arestas.

Para tal, primeiro criamos 2 vetores, uma fila prioritária PQ (um acervo, cuja condição de acervo a analisar é o peso), e o vetor de posições da fila prioritária Pos, com tamanho N (numero palavras de tamanho X).

De seguida criamos outros 2 vetores, vetor peso e vetor caminho, de tamanho N, e inicializamo-los. O vetor peso tem o valor INFINITO em todas as suas posições, e o vetor caminho tem o valor NO_WAY em todas as suas posições. Assim sabemos que se numa certa posição o vetor caminho tiver valor NO_WAY não há caminho conhecido entre a palavra dada e a de origem.

O passo seguinte é colocar na posição da palavra de partida 0, no vetor peso[partida] e a própria posição da palavra no vetor caminho[Partida]. Deste modo a palavra de partida fica a apontar para ela própria, e com peso 0. Insere-se a posição de partida na fila prioritária PQ, como a ordem na fila crescente, segundo os pesos, ao inserir-mos a posição de partida que tem agora peso 0, ela vai para o topo do acervo (neste momento todos os outros índices do vetor peso estão a INFINITO).

Quando se faz Insert na PQ, existe uma variável (N, inicializada a 0) que é incrementada, para mais tarde sabermos a que posição (palavra) vamos buscar os vértices adjacentes (palavras adjacentes/com caminho direto), realiza-se também um fixUp da posição N para ter a certeza que a condição de acervo é mantida depois de inserido um novo elemento.

Sempre que alteramos o valor de um peso, tem que se fazer PQDec, esta função faz fixUp com o vetor Pos, de modo a corrigir as posições do acervo depois de alterar um peso.

Nesta altura entrasse num ciclo. Este ciclo só pára quando a variável N atingir o valor 0 (não houver mais palavras a analisar no grafo) ou quando se atinge a palavra de chegada.

No ciclo, retira-se do acervo o elemento que está no topo (posição 0) e troca-se com o elemento no fim da tabela (posição N-1), e faz-se fixDown desde para atualizar o grafo (com o N já decrementado, deixando o elemento retirado de fora).

Se o peso desse elemento for diferente de INFINITO (o primeiro elemento retirado é o ponto de partida que tem peso 0) entramos num ciclo for que vai buscar ao grafo as palavras adjacentes ao elemento retirado (apenas as que são adjacente por um numero de mutações válido). Se o valor no vetor peso dessas palavras adjacentes (válidas) for INFINITO, significa que ainda não se encontravam na PQ, insere-se as palavras na PQ, fazendo o PQinsert (incrementando o "tamanho" da PQ (N++) e fazendo fixUp para garantir que a palavra fica na posição certa no acervo).

Mas caso a palavra já existisse na PQ (caminho diferente de INFINITO), se o caminho da palavra for superior ao caminho até à palavra que foi retirada mais da palavra retirada a esta (peso da aresta), o caminho da palavra é atualizado ficando com um caminho mais curto (volta-se a fazer PQDec para garantir a condição de acervo se manter depois do peso ser alterado).

Isto faz-se para todas as palavras adjacentes à retirada. Quando já não houver mais, voltasse a retirar o elemento que está no topo do acervo (elemento com menor peso).

Desta forma garante-se que estamos sempre a trabalhar com o caminho mais curto, e que quando chegarmos à posição da palavra de chegada, esse é obrigatoriamente o caminho mais curto.

DESCRIÇÃO DAS ESTRUTURAS DE DADOS

Para a estrutura dicionário optou-se por usar uma matriz de palavras pois facilita no acesso às palavras apesar de obrigar a duas leituras do ficheiro dicionário. Esta estrutura é composta por:

- um campo "matriz" que guarda todas as palavras que compõem o dicionário;

- um campo "num_pal" que contem o numero de palavras para cada tamanho;

- um campo "position" que indica o índice de uma palavra do dicionário na "matriz".

Para a guardar os problemas tem-se uma lista de estruturas "problema". Optou-se por usar uma lista pois assim lemos apenas uma vez o ficheiro e a memória fica à conta pois à medida que se lê um problema aloca-se o espaço necessário para o mesmo. A estrutura "problema" é composta por:

- um campo com a palavra de origem;

- um campo com a palavra de destino;

- um campo "mutações" com o numero limite de alterações de caracteres por passo permitida para o problema.

Para a estrutura com as soluções optou-se por uma tabela, pois o tamanho é conhecido e o acesso direto. Esta é composta por estruturas do tipo "solução", que contem a informação do custo do caminho e um ponteiro para uma pilha de palavras que compõem o caminho a percorrer. Optou-se usar este formato pois à medida vão-se inserindo as palavras no topo da fila e quando for para escrever o ficheiro leem se da palavra do topo até à da base.

Na construção do grafo utilizou-se uma tabela de listas de adjacências, pois a matriz iria apesar de poder ser de acesso direto iria ocupar imensa memoria, e como na maioria das vezes o programa trabalha com grafos esparsos não existe essa necessidade de gasto. As listas são constituídas pela estrutura "aresta" que contem:

- um campo com o peso dessa mutação;

- um campo com o índice da palavra em questão;

Por fim para a fila de prioridades utiliza-se um acervo, pois otimiza o tempo gasto nas operações a efetuar sobre esta, visto terem de se mudar posições (tabela é pior), e fazer acessos (listas são piores).

DESCRIÇÃO DOS SUBSISTEMAS FUNCIONAIS

O programa implementado tem 6 subsistemas implementados, matriz.c e matriz.h que são responsáveis por criação e tratamento de dados sobre a forma de matrizes, lista.c e lista.h que realizam as várias operações de interesse a uma estrutura de dados sob a forma de lista. Tem-se ordenação.c e ordenação.h para os processos de ordenamento e de procura. O subsistema ficheiros.c e ficheiros.h é responsável por tudo o que envolva escrita/leitura de ficheiros, o heap.c e heap.h que são responsáveis por todas as operações que envolvam filas prioritárias e por fim o functional.c e o functional.h, este é responsável pela criação e inicialização do dicionário, da lista de problemas e soluções, e contem todas as funções que são necessárias para a construção e manipulação das estruturas de dados necessárias para o grafo. Tem-se ainda o wordmorph.h com as várias estruturas do programa, constantes e macros auxiliares.

matriz.c

void **criar_matriz**(int**, int*);

Alocação de espaço para uma tabela de inteiros de tamanho variável e inicializa a tabela.

void **criar_matriz_2**(char**, int, int);

Aloca memória para uma tabela de palavras de tamanho único.

void **criar_matriz_3**(char***, int*);

Aloca memória para várias tabelas de palavras.

void **prencher_matriz**(char***, char*);

Preenche a matriz de palavras com as palavras contidas no dicionário.

void **free_matriz_3**(char***, int*);

Liberta a memoria alocada por uma matriz.

void **free_matriz_2**(void**, int);

Liberta a memoria ocupada por uma tabela.

lista.c

Item **criaNovoNo** (void);

Aloca um nó.

Item **adicionar_elemento**(Item, Item);

Adiciona um no previamente criada ao fim de uma lista já existente.

Item **criar_lista**(Item, int);

Cria uma lista com o número recebido como parâmetro de nós.

int **cont_ele**(Item prob);

Conta os elementos de uma lista.

Item **get_next**(Item);

Recebe um elemento de uma lista como argumento e retorna o elemento seguinte.

Item **get_est**(Item);

Recebe um elemento de uma lista como argumento e retorna a estrutura com os dados desse nó.

void **insert_est**(Item, Item);

Liga uma estrutura ao respetivo nó.

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

Liberta a memoria alocada para a lista e respetivos campos constituintes.

ordenacao.c

void **organizar_tabelas**(int, int*, char**);

Selecionar qual o melhor algoritmo para organizar uma tabela, e executa-lo.

void **quick_sort**(int*, char**, int, int);

Organiza uma tabela.

void **selection_sort**(int, int*, char**);

Organiza uma tabela.

int **partition**(int*, char**, int, int);

Encontra os índices a trocar de posição com o pivô.

int **find_position**(char*, int*, char**, int);

Procura um elemento numa tabela organizada.

ficheiros.c

void **verifica_inputs**(int , char **);

Verifica se os ficheiros de input são do formato pretendido.

void **sele_str**(char **, int);

Função que seleciona qual a extensão pretendida.

void **numero_pal**(FILE *, int*);

Contagem de palavras por tamanhos.

FILE* **abre_ficheiro**(char*, char*);

Abre um ficheiro.

FILE* **cria_ficheiro**(char*);

Criação de um ficheiro com extensão ".path".

void **escreve_ficheiro**(solucao*, int, char*);

Escreve as respostas aos problemas num ficheiro de saída.

heap.c

void **PQinit**(int);

Alocação de espaço para o acervo e o seu mapa.

int **PQempty**(void);

Verifica se a fila prioritária está vazia.

void **PQinsert**(int, int*);

Insere um elemento na fila de prioridade consoante a sua precedência.

int **PQdelmin**(int*);

Elemina o elemento de menor prioridade.

void **PQdec**(int, int*);

Corrige as posições no acervo aquando uma alteração de peso.

void **fixUp**(int*, int);

Comparação de um membro com o seu antecessor no acervo e em caso de a posição estar trocado proceder com sua correção.

void **fixDown**(int*, int, int);

Compara e faz descer pelo acervo uma posição que seja menos prioritária do que outras para trás.

void **PQfree**();

Liberta a memoria alocada pelo acervo e o respetivo mapa.

void **troca**(int, int);

Troca o valor de duas posições no acervo e corrige o mapa.

void **PQrestart**();

Reinicia o contador com o número de posições validas na fila de prioridades.

functional.c

dicionario* **create_dic**(void);

Alocação de espaço para as várias tabelas da estrutura dicionário.

void **start_dic**(dicionario*, char**);

Preenche a estrutura dicionário com a informação pretendida.

Item **guardar_prob**(Item, FILE*, int*, int*);

Aloca e preenche uma lista com os problemas a resolver.

Item* **criar_tabela_adj**(Item*, int, int, char**);

Alocação de memória e inicialização de todos os parâmetros das listas de adjacencias.

Item **init_aresta**(int, int);

Aloca memória e inicializa os valores das arestas.

int **cmp_char**(char*, char*, int);

Compara duas palavras e retorna o número de caracteres diferentes.

int* **vetor_num**(int);

Alocação de memoria para um vetor.

void **achar_caminho**(Item*, int, int, int*, int*, int, int);

Encontra o caminho mais curto da origem ao destino a partir de pesos e de uma fila de prioridade.

solucao **guardar_sol**(int, int, int*, int*, char**);

Aloca e preenche uma tabela com as soluções aos problemas.

void **free_prob**(Item);

Liberta a memoria alocada para a estrutura problema e dos seus respetivos campos.

void **free_tab_adj**(Item*, int);

Liberta a memoria alocada para a lista de adjacencias.

void **free_adj**(Item);

Liberta a memoria alocada para uma aresta.

void **free_sol**(solucao*, int);

Liberta a memória da tabela de soluções.

void **free_sol_est**();

Função vazia.

ANÁLISE DOS REQUISITOS COMPUTACIONAIS

Durante a realização do projeto a escolha das estruturas de dados a utilizar foi sempre com o objetivo de otimizar em termos de tempo a execução do programa e encurtar ao máximo a memória necessária.

Dicionário:

Para a estrutura dicionário optou-se pela utilização de uma matriz pois assim o acesso a uma palavra tem complexidade O(1), o que é bastante rentável pois ao longo do programa por diversas vezes se tem de aceder a palavras do dicionário.

Problemas:

Para a estrutura problemas utiliza-se uma lista logo sempre que se percorre a lista tem se uma complexidade de O(P), o que neste caso é irrelevante pois é sempre pretendido ler todos os elementos do início ao fim, não existindo qualquer inconveniente de não se poder aceder a posição x de memoria sem passar pelas outras. Já na criação da lista, sempre que se lê um problema o custo de inserção é de O(1).

P – Numero de problemas.

Soluções:

Nesta estrutura os acessos são diretos por se tratar de uma tabela, ou seja, custo é O(1), e o custo para correr a lista a associada a cada posição também pode ser considerada de custo simples, pois quando se pretende aceder a essa informação quer-se percorrer toda a lista.

Grafo:

Para o grafo foi utilizou-se uma tabela de listas de adjacências, sendo o acesso à lista O(1) e o acesso a um elemento da lista de custo O(A), o qual não é desfavorável, pois sempre que se quer aceder à lista é para verificar todas as suas posições sendo o custo a um elemento equivalente a O(1). A inserção de um elemento também é de custo O(1), sendo assim bastante vantajoso em termos quer de tempo quer de memoria ocupada.

A – numero de arestas.

Principais algoritmos:

organizar_tabelas( ):

Este algoritmo, consoante o numero de palavras dadas de um certo tamanho, ordena as mesma com custo O(N^2) para um numero de palavras abaixo de um certo limite e com complexidade O(N*lg(N)) para um numero de palavras acima desse limite. Este limite é escolhido com base em dados estatísticos de forma a otimizar o tempo gasto nesta operação. Isto acontece pois o algoritmo usado quando o número de palavras está acima

do limite é mais complexo e tem mais operações do que o outro que é mais simples, mas menos eficiente para grandes quantidades de dados.

N – número de palavras.

find_position( ):

Este algoritmo procura palavras no dicionário com custo O(lg(N)), pois como a tabela esta organizada, compara com o valor a meio da tabela e conforme seja maior ou menor irá dividir a tabela a meio e repetir o processo ate achar a palavra, fazendo consecutivos cortes para metade da tabela.

N – número de palavras.

achar_caminho( ):

O custo desta função, no máximo, será de E*lg(V), pois no limite percorrera todas as arestas das varias listas e ira ter de fazer para cada um lg(V) trocas, pois é usado um acervo na lista de prioridades.

E – número total de arestas.

V – número de vértices.

guardar_solucao( ):

Neste algoritmo o custo é de proporcional ao numero de palavras que compõem a solução, ou seja O(S).

S – número de palavras que compõem o caminho da origem ao destino

criar_tabela_adj:

A execução desta função implica ½*C*N^2 comparações, pois são necessária fazer as comparações entre todas as palavras, para se saber o número diferente de caracteres entre elas para a criação da lista de adjacências. Logo, como C << N na grande maioria dos casos e C nunca tem valores significativamente elevados, o custo associado a esta operação é O(N^2).

C – número de caracteres da palavra.

N – número de palavra.

FUNCIONAMENTO DO PROGRAMA

Depois de fazer alguns dos teste disponibilizados pelos docentes, submeteu-se o projeto no website, o primeiro resultado foi 10 testes passados em 20, com o erro de inexistência de caminho para problemas que tinham solução.

Analisamos e corrigimos o erro, que era na inicialização do PQ e do respetivo mapa.

Submetemos outra versão, passou a 7, descobriu-se que o erro era o critério de paragem do algoritmo de procura binária. Desta vez testou-se todos os testes disponibilizados pelo corpo docente e o programa realizou todos corretamente, tirando o ultimo que demorava um tempo algo "desagradável".

Mais algumas alterações menores e ajustes foram feitos para colocar o programa mais rápido e sem erros, por exemplo para o caso de problemas que permitiam mudanças de x caracteres e as palavras do problema difiram em um número menor de letras, apenas se considera o número de caracteres diferentes. Passou em todos os testes fornecidos pelos professores e em 19 dos 20 problemas do website, com erro de execução. Esse era causado no fim do programa aquando, para um dado tamanho, apenas e só existe problemas com 0 mutações permitidas ou caracteres diferentes. Corrigido esse passou-se a ter erro de excesso de tempo. Como o programa apenas tinha um erro de execução antes, e este apenas se dava já no final do programa, a quando a escrita do ficheiro, pensamos que o excesso de tempo possa ser devido a estarem sempre várias pessoas ao mesmo tempo a submeter o projeto, pois as alterações produzidas apenas obrigam a adicionar processos de ordenação e de procura binaria, os quais não nos parecem ser os causadores de tal atraso.

EXEMPLO DE APLICAÇÃO

Segue-se um pequeno exemplo com a descrição da aplicação no cálculo do caminho mais curto entre duas palavras.

Para um ficheiro dicionário com as palavras:

hoje ar casa comida haja ir seno caja bola

E sendo o ficheiro problema:

hoje casa 2

1º - O programa cria os vetores necessário para o programa (ponteiro para a estrutura dicionário, ponteiro para a lista de adjacências (grafo de palavras), ponteiros para a lista de soluções, para a lista de problemas e para uma estrutura problema, vetor que indica se

existe problemas com aquele tamanho, inicializado a 0, tal como o vetor de mutações máximas).

2º - Verifica se os inputs estão corretos, caso não estejam termina o programa.

3º - Cria a estrutura dicionário, e dentro da estrutura cria a matriz que vai guardar as palavras (com o numero de linhas STR_SIZE, que tem de ser sempre superior ao tamanho da palavra máxima) assim em cada linha vão ficas as palavras de tamanho correspondente ao índice, cria-se o vetor numero de palavras (também com o tamanho STR_SIZE) que vai guardar o numero de palavras de cada tamanho, e a matriz de inteiros position (numero de linhas = STR_SIZE) que vai guardas as posições das palavras.

4º - Abre-se o ficheiro dicionário que contem as palavras.

5º Coloca-se o numero de palavras de cada tamanho no vetor da estrutura dicionário (point->num_pal) , para tal lemos palavra a palavra do dicionário e incrementamos o valor de point->num_pal[Tamanho_palavra_lida].

No nosso caso o vetor point ->num_pal ficaria {0, 0, 2, 0, 5, 0, 1, 0, 0, 0, 0, ……,0}

6º - Fechamos o ficheiro dicionário.

7º - Inicializamos a matriz de palavras (point->palavras) e a matriz posições (point->pos) da estrutura dicionário.

Na criação da matriz de palavras, criamos um vetor de palavras de cada tamanho (point->palavras[tamanho_palavra], com o tamanho do número de palavras desse tamanho que existem, se não houverem palavras de certo tamanho o vetor tem comprimento 0. Dentro desse vetor (point->palavras[ ]) cria-se o espaço necessário para armazenar as palavras (point->palavras[ ][ ]).

8º - Preenchemos a matriz de palavras que acabamos de criar copiando diretamente do ficheiro dicionário.

No nosso exemplo iriamos ficar com uma matriz de palavras:

	ar	Ir			
	hoje	casa	haja	sena	Caja
	Comida				

…….

9º - De forma semelhante criamos a matriz de posições da estrutura do dicionário, e preenchemo-lo com as posições. No nosso exemplo ficamos com a matriz point->pos:

….

	0	1			
	0	1	2	3	4
	0				

….

10º - Abrimos o ficheiro problemas e criamos uma lista de problemas. À medida que vamos criando problemas (enquanto ficheiro problemas não chegar ao fim) inicializamos a estrutura problema com os dados do ficheiro problemas e atualizamos também o vetor num_max_mut (numero máximo de mutações) e o vetor que indica se existe problemas daquela tamanho, existe[ ]. Um pormenor a ter em consideração, se o problema pedir mutações de tamanho 5 para palavras tamanho 6, mas a diferença entre as palavras forem apenas 3 mutações, o número máximo de mutações torna-se 3.

No nosso exemplo a lista apenas terá um problema, com a palavra destino casa, palavra de partida hoje, mutações 2.

O vetor existe[ ] = {0, 0, 0, 0, 1, 0, 0, …,0} e o vetor num_max_mut[ ] = {0, 0, 0, 0, 2, 0, …., 0}.

11 º - Fecha o ficheiro com os problemas.

12º -Cria um vetor de soluções com o numero de problemas que existem. No nosso caso o vetor apenas tem uma posição.

13 º - Nesta parte começa a resolução dos problemas, só acontece quando o vetor existe[i] é diferente de 0. No nosso exemplo isto apenas acontece quando i = 4 (palavras tamanho 4).

A resolução começa por ordenar a tabela de posições das palavras de acordo com a ordem alfabéticas das mesmas, para tal utiliza-se o QuickSort que já foi anteriormente explicado.

No exemplo a tabela de posições de tamanho 4 irá ficar:

	4	1	2	0	3

14º - Cria-se o grafo das palavras de tamanho pretendido (tabela de adjacências) com o tamanho do número de palavras daquele tamanho. Comparando-se as palavras, sempre que para passar de uma palavra para a outra, o número de mutações seja necessário ao número de mutações máximas, cria-se um novo nó na lista, e nesse nó insere-se uma aresta entre as palavras onde guarda-se o índice da palavra no dicionário e o peso de passar de uma para a outra.

No exemplo ficamos com:

[hoje] -> (haja)

[casa]->(haja) ->(caja)

[haja]->(hoje) -> (casa) -> (caja)

[seno]->

[caja]->(casa) -> (haja)

15º - Inicializamos a PQ, a Pos, o pesos e o caminho que se usam no algoritmo Dijkstra. Enquanto houver problemas na lista de problemas, e caso os problemas sejam para o grafo que criamos, usamos a procura binaria (explicada anteriormente) para encontrar a posição da palavra destino e da palavra inicial.

No exemplo: Posiçao_inicial = 0; Posição_final =1

16º - Vamos procurar o caminho mais curto entre as palavras, usando o algoritmo Dijkstra, que já foi explicado anteriormente, o que vai alterar o nosso vetor caminho e pesos. No exemplo estes vão ficar:

Pesos:

0	5 + 1 =6	4	INFINITO	4+1=5

Caminho:

0	4	0	-1	2

17º - Guardamos a solução na nossa lista de soluções e libertamos o grafo, os vectores caminho, pesos, PQ e Pos.

Volta-se a repetir este processo desde o ponto 12, até não haver mais problemas a resolver.

18º - Escrevemos o ficheiro soluções e fazemos free de toda a memoria alocada que ainda não foi libertada.

REFERENCIAS

Robert Sedgewick, 1988, Algorithms in C, Addison-Wesley.

Corpo docente de AED, 2016, Acetatos.

Corpo docente de AED, 2016, Guia do projeto.