INSTITUTO SUPERIOR TÉCNICO

MESTRADO EM ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES

ALGORITMOS E ESTRUTURAS DE DADOS

2016/2017 – 1ª SEMESTRE

PROJETO FINAL - RELATÓRIO

WORDMORPH

Grupo nº 53:

João Cardoso, nº 84096
e-mail: joao.pedro.cardoso@tecnico.ulisboa.pt

Sara Henriques, nº 81261
e-mail: sara_r_henriques_96@hotmail.com

Docente: Prof. Carlos Bispo

ÍNDICE

DESCRIÇÃO DO PROBLEMA

Neste projeto pretende-se implementar um programa que seja capaz de produzir "caminhos" entre palavras. Por caminho entre duas palavras com o mesmo tamanho (dadas como ponto de partida e de chegada), entende-se uma sequência de palavras (com este mesmo tamanho) sendo que cada palavra se obtém a partir da sua antecessora por substituição de um ou mais caracteres. Todas as palavras, tanto as de partida e de chegada como as que irão pertencer ao caminho, deverão constar num dicionário, fornecido para o efeito.

Por exemplo, seja a seguinte sequência de palavras, que estabelece um possível caminho de palavras entre **"carro"** e **"pista"**:

carro
porto
porta
pista

Neste caso, a segunda palavra é obtida a partir da primeira por substituição de 3 caracteres, a terceira é obtida a partir da segunda por substituição de apenas 1 caracter e a última é obtida a partir da terceira por substituição de 2 caracteres, tendo sido realizadas, portanto, uma mutação tripla, uma mutação simples e uma mutação dupla para transformar a palavra **carro** em **pista**. Por mutação entende-se a transformação de uma palavra noutra por substituição de um ou mais caracteres.

Poderão haver então muitos outros caminhos possíveis entre estas duas palavras. De forma a distinguir os diferentes caminhos possíveis, apesar de poder existir apenas um caminho, ocorrerá um custo associado às mutações. O custo presente no projeto será quadrático no numero de caracteres substituídos entre duas palavras consecutivas (N^2), isto é, este exemplo possuirá um custo igual a 14 (mutação tripla -> $3^2 = 9$, mutação dupla -> $2^2 = 4$, mutação simples -> $1^2 = 1$, logo, custo total = 9 + 4 + 1 = 14).

Finalmente, o que se pretende com este projeto é desenvolver uma aplicação em linguagem C que consiga calcular o caminho de menor custo entre duas palavras de igual tamanho. O ficheiro de entrada poderá conter mais do que um problema (semelhantes a este exemplo, mas possivelmente serão problemas para vários tamanhos de palavras), cada um acompanhado pelo número máximo de caracteres que se podem substituir numa mutação entre duas palavras consecutivas. Por esta razão, alguns problemas podem, como esperado, não possuir solução.

ABORDAGEM AO PROBLEMA

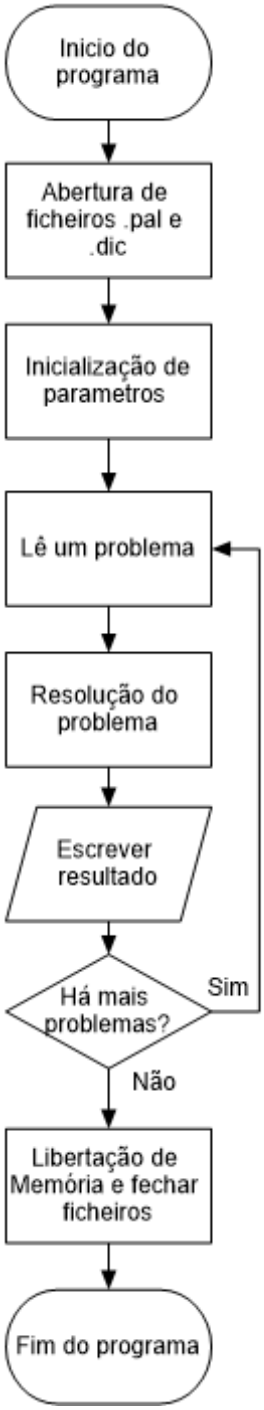
Primeiramente, decidiu-se que iria ser necessário apenas guardar as palavras do dicionário necessárias para cada problema, pois, para cada um só interessará as palavras de igual tamanho ao das palavras de partida e chegada. Então, reunindo informação obtida em aulas teóricas e em discussões de ideias com os docentes, optou-se por utilizar um grafo ponderado com listas de adjacências (visto tratar-se de um grafo esparso) como estrutura de dados do programa, em que cada vértice é uma palavra e cada peso de uma aresta entre duas palavras é o custo para transformar uma palavra na outra.

Para cada problema criar-se-á um grafo, caso não tenha já ocorrido um problema para este tamanho de palavras. Para ordenar as palavras alfabeticamente recorreu-se ao algoritmo de Quicksort e, para de seguida se encontrar as palavras no grafo, recorreu-se ao algoritmo de Procura Binária.

Seguidamente, para se encontrar o caminho mais curto, optou-se pelo uso do algoritmo de Dijkstra. Para este fim, utilizou-se a estrutura de dados heap (acervo), em que cada nó será representado por um vértice e pelo custo do caminho entre a palavra de partida e esse mesmo vértice. Assim, os elementos da heap serão os vértices do grafo associado a esse tamanho de palavra.

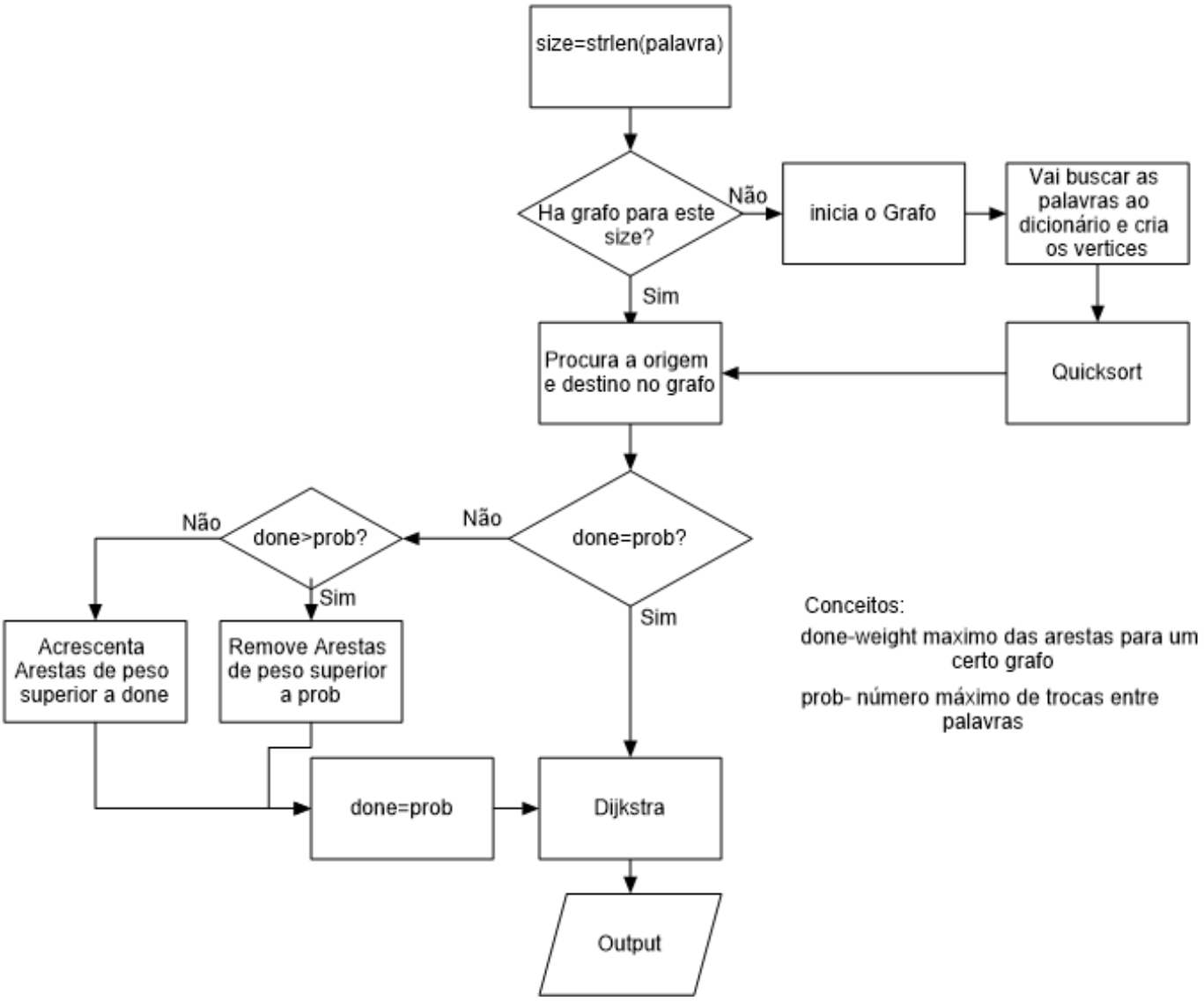
Contudo, uma descrição detalhada dos tipos de dados e dos algoritmos utilizados será apresentada a jusante desta breve abordagem.

ARQUITETURA DE PROGRAMA



O fluxograma anterior explica o desenrolamento do programa no seu geral. Tudo à parte da escrita do resultado e resolução de problemas são funções muito simples. Este fluxograma é apenas para ter uma ideia do funcionamento. O main.c verifica os argumentos, abre os ficheiros .dic e .pal e corre a função ReadPal que trata de interagir com todo o resto do programa (resolução, output e libertação de memoria).

O ReadPal para resolver um problema faz o seguinte.



Há que realçar que done é um vetor de 0 a max, sendo max a maior palavra do dicionário. O pal.c tem um grafo** que aponta para max+1 ponteiros para grafos, 1 para cada tamanho de palavra (e sim, a posição 0 será sempre NULL mas a conveniência ganha é maior do que o custo desse ponteiro). O prob é o inteiro que vem do problema lido.

O ReadPal invoca funções do grafo.c (criação dos grafos, quicksort e procura binária), do heap.c (dijkstra) e do ficheiro.c (output).

DESCRIÇÃO DAS ESTRUTURAS DE DADOS

No projeto foi usado um grafo e uma heap (fila prioritária em heap) para usar o algoritmo de Dijkstra. Por isso a divisão em duas partes é o mais adequado.

GRAFO.C

graph	Grafo
Int V	Número de vértices no grafo
Int E	Número de arestas no grafo
Vertex* adj	Ponteiro para vetor de vértices

Vertex	Vértice do grafo
Char* word	Palavra associada ao vértice
Link* head	Lista de adjacências

link	Nó da lista de adjacências
Int v	Índex no grafo do adjacente
Int weight	Peso da aresta (custo)
Link* next	Próximo elemento da lista

Estas estruturas estão todas escondidas e apenas o fornecedor pode mexer nas mesmas. Estas *structs* são usadas nas funções do próprio grafo, no ReadPal (um grafo**) e no heap.c (vários valores que são precisos e são dados através de funções no grafo.c, ex. **int getV (graph* g)** que vai buscar o V do grafo que entra como argumento).

HEAP.C

Heap	Acervo
Int n_elements	Número de elementos da heap
Int size	Tamanho máximo da heap
Int *pos	Vetor com os índices dos vértices na heap
Heap_node ** array	Vetor com os nos da heap

Heap_node	Nó da heap
Int v	Vértice desse no
Int dist	Distancia do vértice à source

Estas são o tipo de dados usado para criar um acervo. A *struct* heap contem a informação precisa para ele enquanto o heap_node é a estrutura do nó que representa um vértice do grafo. O vetor pos ajuda a transição de grafo para heap, pos[v] = i onde v é um vértice do grafo e i o índex correspondente no heap_node array.

DESCRIÇÃO DOS ALGORITMOS

QUICKSORT

- Ordenação dos vértices por ordem alfabética

Argumentos de Entrada: grafo* g, int l, int r

Output: Void

Sendo "g" um ponteiro para o grafo que está a ser criado para a resolução de problema.

O inteiro l é a borda inferior e o inteiro r é a borda superior.

Este Quicksort é uma adaptação ao Quicksort "normal" (em que é um vetor de inteiros a ser ordenado) para ser um vetor de vértices a ser a ordenado em que cada um possui uma *string* que será usada para diferenciar "vértices" em mais "altos" e mais "baixos".

O Quicksort é uma função recursiva que faz a partição (vai buscar um pivot, que já está na sua posição final) e "divide" a tabela em dois, chamando o Quicksort para a parte inferior (l = 0, r = pivot – 1) e para a parte superior (l = pivot + 1, r = occur) sendo o "occur" o número de palavras neste grafo.

SEARCHWORD

- Procura binária

Argumentos de Entrada: grafo* g, char* str

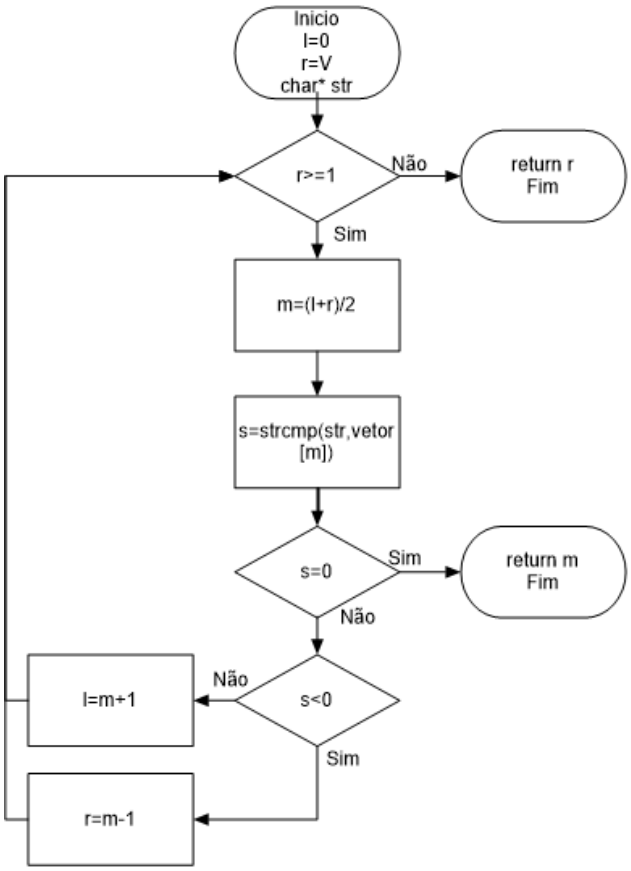
Output: int

A função deste algoritmo é encontrar o índex da string str.

Tendo a borda inferior (l = 0) e a superior (r = g->V) , faz a media (m) dos elementos e compara "str" com a *string* do vértice m. Se maior que 0, altera o l para m+1; se mais pequeno altera r para m-1.

Ou seja divide a tabela sempre em 2 até encontrar a palavra desejada.

O fluxograma da função é o seguinte:



Sendo o vetor[m] a palavra do vértice v.

CREATEADJLIST

- Criação da lista de adjacências

Argumentos de Entrada: graph* g, int prob, int done

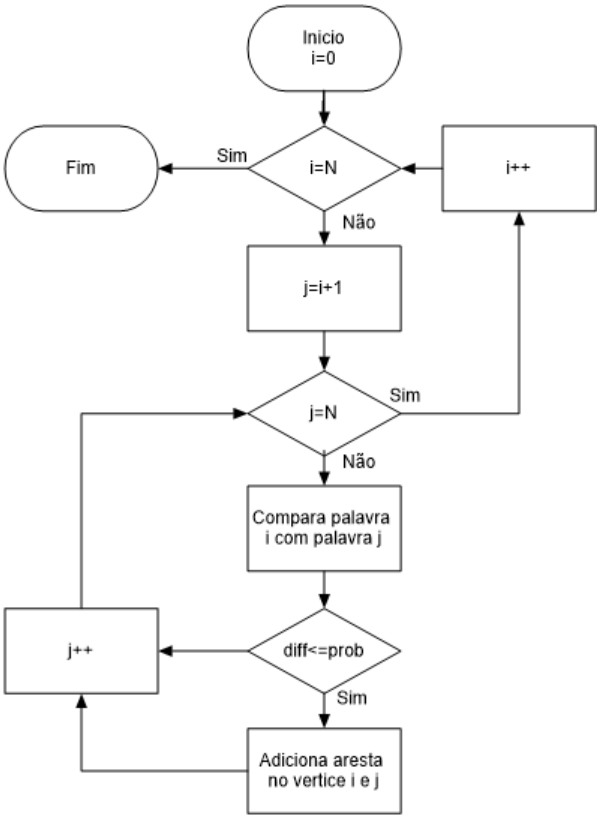
Output: Void

Com o objetivo de encontrar os adjacentes de cada vértice criou-se o CreateAdjList e o DifferentChar. Esta última compara duas palavras e retorna o número de caracteres diferentes até ao inteiro "prob", ou seja, se comparar a palavra divisão e funções, ambas de 7 letras e o prob for 3 quando comparar a letra "i" e "c" (ignorando caracteres especiais, logo ã=a e ç=c) verá que são diferentes e sai da comparação.

A criação da lista de adjacências é bastante simples. A função vai de 0 a g->V ou seja a todos os índices do vetor e compara com todas as posições á sua frente por exemplo:

Se estamos no ciclo 50, essa palavra vai comparar com as palavras de 51 a g->V (ordem dos milhares em dicionários normais). Usa-se a função DifferentChar e se for menor ou igual ao prob cria-se a aresta nos 2 vértices, quer dizer que se adiciona um nó as listas de cada vértice.

O fluxograma da função é o seguinte:



DECREASEKEY

Inicialmente encontra o vértice atual na heap (acervo). De seguida, atualiza o custo do caminho entre a palavra de partida (ou de origem) e a palavra pertencente a este vértice, para um custo menor. Por fim, faz com que a heap verifique a sua condição a partir do nó localizado no início da função, isto é, o custo associado a este nó não poderá ser inferior ao custo dos seus nós ascendentes, realizando trocas entre os mesmos, até que o custo do nó atual seja superior ao custo dos seus nós ascendentes ou até que o nó seja o primeiro nó da heap, pois, o objetivo é possuir uma heap ordenada decrescentemente pela prioridade dos custos associados a cada nó, ou seja, o primeiro nó terá o custo mais pequeno (maior prioridade).

DELETEMINNODE

De uma forma muito geral, este algoritmo retira da heap o nó com menor custo associado. Para se chegar a este fim ter-se-á conta que o nó com menor custo será então o primeiro nó da heap. Assim, troca-se o primeiro nó (com menor custo) com o último nó da mesma, decrementando o seu número de elementos. Finalizando, para que a heap continue a verificar a sua condição, realiza-se Fix Down (algoritmo detalhado seguidamente) sobre o atual primeiro nó (antigo último nó).

FIXDOWN

O algoritmo de Fix Down reordena o acervo (heap), a partir de um certo nó, deste modo: enquanto não restabelecer a condição de acervo ou atingir uma folha, troca este nó com o nó descendente que possuir menor custo.

Portanto, primeiramente, o algoritmo verifica qual dos dois nós descendentes ao nó atual possui menor custo. Se esse custo for maior que o do nó atual, está-se na ordem correta, portanto não se altera mais nada na heap. Contudo, se for menor, os nós trocam-se e continuam-se a realizar as mesmas ações mais uma vez para o nó atual (nó inicial), até a ordem estar correta ou se atingir uma folha, tal como referido anteriormente.

DIJKSTRA

O algoritmo de Dijkstra, de uma forma muito geral, calcula o menor caminho entre duas palavras, isto é, calculará o caminho que apresenta menor custo entre a palavra de partida e qualquer outra palavra da heap.

Para começar, todas os custos associados a cada nó da heap deverão ser iguais a "infinito", exceto o custo da palavra de origem que deverá ser zero (sempre que for necessário diminuir o custo associado a um nó dá-se uso ao algoritmo DecreaseKey, que já foi apresentado em detalhe anteriormente), pois, o custo do caminho entre ela mesma é zero.

De seguida, retira-se da heap o nó com menor custo (inicialmente o nó com maior prioridade é o nó correspondente à palavra de partida) e entra-se num ciclo no qual é analisada a lista de adjacências deste mesmo nó. Para cada nó da lista de adjacências, se o novo custo (soma do custo associado ao nó ao qual pertence a lista de adjacências com o custo entre este nó e o nó presente na lista) for menor que o custo atual associado a este mesmo nó, atualiza-se o custo e guarda informação acerca do caminho de palavras (com o auxílio do vetor **st**). Este processo será repetido até o nó com menor custo ser o nó com a palavra de chegada ou possuir custo "infinito" (neste caso, não haverá caminho entre a palavra de partida e a palavra correspondente a este nó) ou, ainda, até a heap ficar vazia, ou seja, já estarão calculados todos os menores caminhos entre a palavra de partida e as palavras associadas a cada nó da heap.

DESCRIÇÃO DOS SUBSISTEMAS

GRAFO.C

FUNÇÕES DE INICIALIZAÇÃO:

graph ***InitGraph**(int V):

Cria um grafo com V vértices

graph** **AllocG**(graph** G,int max):

Cria um vetor de grafos com tamanho "max", que é o tamanho da maior palavra do dicionário.

link* **NewNode** (int v, int weight, link*next):

Cria um nó para uma lista de adjacências e inserte-o nessa mesma lista.

FUNÇÕES DE SORT E PROCURA:

Void **SortAndSearch** (graph* g,int occur, char* str1,char* str2, int* indexSG):

Aplica o quicksort e a procura binaria para a str1 e str 2 (palavras que vem em cada problema). IndexSG significa Index Source-Goal, ou seja, um vetor de 2 posições, a primeira com o índice da palavra str1 e o outra com o índice da palavra str2.

Void **quicksort**(graph* g, int l, int r):

Ordena um vetor de vertices com base na palavra contida em cada um deles.

int **partition**(graph* g, int l , int r):

Função invocada pelo quicksort para fazer as partições.

Int **SearchWord**(graph* g, char* str):

Procura a palavra num array ordenado, neste caso num array de vértices ordenados.

FUNÇÕES DE CRIAÇÃO/DESTRUIÇÃO DO GRAFO:

Graph* **InsertWordsG**(graph* g,FILE*dic,int sz,int occur):

Insere as palavras do dicionário no grafo, o sz é o tamanho das palavras e o occur é o numero de vezes que esse tamanho de palavras aparece no dicionário.

Void **CreateAdjList**(graph* g, int prob, int done):

Adiciona arestas aos vértices. Serão criadas arestas entre 2 vértices se a diferença for maior que "done" e menor ou igual que "prob". (Ou seja se existe um grafo com arestas para um prob de tamanho 2, e o novo prob e de tamanho 3, adiciona as arestas de peso 3 ao grafo.

Graph* **InsertEdge** (graph* g, int src, int dest, int weight):

Cria uma aresta entre 2 vertices.

Void **RemoveE**(graph* g, int problem):

Remove as arestas que sejam maiores que prob. Há que referir que se compara o quadrado de problema com o weight, pois o weight está ajustado para depois o custo estar certo.

FUNÇÕES DE AUXILIO:

Int **DifferentChar**(char* s1, char* s2 ,int prob):

Retorna o valor da diferença entre as palavras se for menor ou igual a prob, se não retorna prob+1.

Char* **getword**(graph* g,int v):

Retorna a palavra do vertice v.

Int **getw**(link* aux):

Retorna o peso da aresta do nó passado como argumento.

Int **getv**(link* aux):

Retorna o v (vértice, ou seja, o índex no vetor de vértices) que está referenciado neste nó.

Link* **gethead**(graph* g,int v):

Retorna a cabeça da lista de adjacências.

Link* **getnext**(link*aux):

Retorna o adjacente seguinte.

Int **getV**(graph*g):

Retorna o número de vértices do grafo.

Void **Freeg**(graph**G,int max,int*occur):

Liberta o vetor de grafos e em cada um, o seu vetor de vértices.

HEAP.C

FUNÇÕES DE ALOCAÇÃO DE MEMORIA/FREE DA HEAP

heap* **CreateHeap**(int size) :

Cria uma heap com tamanho "size" (aloca memória e inicializa os seus parâmetros) para auxiliar o algoritmo de Dijkstra.

void **FreeHeap**(heap *h) :

Liberta toda a memória alocada para a heap "h".

heap_node* **NewHeapNode**(int v, int dist) :

Cria um novo nó na heap com vértice "v" e custo associado "dist" (aloca memória e inicializa parâmetros).

FUNÇÕES DE MANIPULAÇÃO DA HEAP

void **DecreaseKey**(heap *h, int v, int dist) :

Encontra o vértice "v" na heap "h", diminui o seu custo (passa a ser "dist") e reordena a heap.

heap_node* **DeleteMinNode**(heap *h) :

Retira da heap "h" o nó com menor custo (maior prioridade).

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

Reordena a heap "h" a partir do nó "k".

int* **Dijkstra**(graph *g, int* indexSG, int* cost) :

Calcula o menor caminho (caminho com menor custo) entre a palavra de partida e a de chegada.

DIC.C

Int **ReadDicBiggestW**(FILE* dic):

Lê o dicionário e encontra a maior palavra, retornando o seu tamanho.

Void ReadDicOccur(FILE* dic, int* occ):

Lê o dicionário e encontra a instância de cada palavra, guardando num vetor de ocorrências.

FICHEIRO.C

Void **checkArg** (cchar* dic, char* pal):

Verifica os argumentos de entrada

FILE* **openFile** (char* str):

Abre o ficheiro com o nome str em modo read

Void **foutput** (int*st,graph* g, FILE* out,int dest, int co):

Faz output do resultado obtido usando a função STA.

Char* **STA**(graph* g, int a,int* st,FILE* out, int cost)

Função recursiva que escreve no ficheiro. Se entre 2 palavras houver um caminho que use 2 palavras entre o source e o dest, o st[a] dá-nos o índex de uma delas e st[st[a]] dá-nos para a outra.

ANÁLISE DOS REQUISITOS COMPUTACIONAIS:

Quando foi planeado o programa, foi tido em conta o tempo de execução como prioridade sobre a memória. Logo foi escolhido tabelas em deterioramento de listas exceto para o problema da adjacência que se escolheu listas por ser grafos esparsos, obviamente depende do prob, que quanto mais próximo do tamanho da palavra, mais denso fica.

Acesso Vetores de grafos, heaps e vértices: O(1)

Acesso a um nó a lista de adjacências: Pior caso O(N) mas apenas para o **RemoveE** terá de acontecer, pois, as outras funções que acedem a lista de adjacências usam sempre todos os adjacentes logo será igual a um vetor o tempo de acesso a cada um individualmente.

Em questão de memória, escolheu-se guardar os grafos para cada tamanho de palavra pois a sua construção tem um custo de tempo dispendioso por isso, se fosse para construir e destruir o grafo cada vez que era feito, levaria exponencialmente mais tempo a concluir um ficheiro de problemas. Quando é preciso um número de mutações maior, acrescenta-se ao grafo. Assim é mais rápido do que no caso de acrescentar todas as arestas possíveis e depois ignora-las no dijkstra.

COMPLEXIDADE DAS FUNÇÕES PRINCIPAIS:

Para saber a complexidade geral do programa depende de várias incógnitas, V, o número de palavras no dicionário, P, o número de mutações possível e E, o número de arestas do grafo (quanto maior o P, maior o E) .

Deixando a leitura do dicionário de parte (que é feita varias vezes, 1 vez para apurar o max, outra para as ocorrências e mais X para por as palavras no grafo).

QUICKSORT:

O quicksort vai dividindo o vetor em secções cada vez mais pequenas, mas a sua complexidade varia entre NlogN (Caso médio) e N^2(pior caso), como já foi discutido nas aulas teóricas.

SEARCHWORD:

A procura binária terá uma complexidade proporcional a log n porque divide o vetor sempre em 2, reduzindo o número de problemas em 2 a cada ciclo.

CREATEADJLIST:

Sendo P o prob e N o número de palavras que estão no vetor, que é o número de palavras que estão no dicionário de certo tamanho que se aplica ao problema em questão, a complexidade será do tipo $\left(\frac{N^2-N}{2}\right) * P$. Chega-se a complexidade quadrática ($O(N^2)$) porque o número de objetos a comparar será $(N-1)+(N-2)\ldots.+1+0$ e o resultado da soma será $M*(M+1)/2$ sendo M = N - 1.

***DIJKSTRA*:**

Recebendo um grafo g, com E arestas e V vértices, a sua complexidade será O(ElogV)

A priority queue implementada em heap pode ter até V elementos, cada extração leva O(logV) por isso será O(NlogV) se for preciso retirar todos os Nodes da heap.

Considerando também o tempo gasto a percorrer as listas e a decrescer a prioridade teremos O(ElogV), juntado os dois teremos O(ElogV) porque N será O(E) para grafo conectados.

Em questão de memória, para resolver cada problema, é alocada uma heap nova.

FUNCIONAMENTO DO PROGRAMA

Inicialmente, o programa passou em apenas 7 testes, num total de 20, pois, o custo calculado pelo programa nem sempre era o custo mais pequeno. Foram realizados alguns ajustes e o número de testes passados foi evoluindo para 9, até que se chegou aos 16 testes passados. Para esta subida acentuada foram feitas alterações no algoritmo de Dijkstra, que se encontrava inicialmente um pouco incompleto a nível de restrições, e no grafo presente, pois, este continha mais arestas do que o suposto e estas posteriormente, para alguns casos, entrariam no calculo do caminho mais curto levando a resultados errados.

Por fim, com alguns ajustes de modo a aumentar maioritariamente a rapidez de funcionamento do programa, o nosso projeto acabou por passar 17 testes (num total de 20). Ainda houve alguns esforços e alterações para chegar ao objetivo de 20 testes passados, contudo, devido à sobrecarga excessiva do servidor de submissões nos dias que se avizinhavam ao dia da entrega, foi acordado entre ambos os elementos do grupo não realizar mais nenhuma submissão de modo a garantir um final de 17 testes passados (e não menos, se possível) e também de forma a não contribuir para a já excessiva sobrecarga do servidor, tentando, assim, não prejudicar grupos que terão mais urgência em submeter o seu projeto com bons resultados, visto que o resultado final de 17 testes passados já consegue demonstrar o trabalho e dedicação que foram depositados neste projeto.

EXEMPLO DE EXECUÇÃO

INPUT:./wordmorph português.dic teste.pal

Onde teste.pal é a seguinte linha “bacoco regiao 1” e portugues.dic o dicionário completo

OUTPUT:

bacoco 10

bacoro

baforo

bafora

bafara

rafara

refara

refira

refiro

regiro

região

Para resolver este problema, é criado o G[6], size = 6, prob = 1, antes do CreateAdjList done[6] = 0, depois done[6] = 1.

BIBLIOGRAFIA

- Acetatos 7 (algoritmo de Quicksort), 8 (para o grafo), 10 (algoritmo de Dijkstra) e 11 (para a heap) das aulas teóricas.

- Laboratórios 2 (para manipulação de listas, de vetores e/ou tabelas e de *string*'s), 4 (para a manipulação do grafo) e 6 (para a manipulação da heap).

- https://en.wikipedia.org/wiki/Dijkstra's_algorithm