Instituto Superior Técnico

Engenharia Eletrotécnica e de Computadores

Algoritmos e Estruturas de Dados

Wordmorph

Autores:

Carolina Lima	83993	carolina.guariglia@tecnico.ulisboa.pt
Carina Fernandes	84019	carina.m.fernandes@tecnico.ulisboa.pt

Grupo 23

2016/2017 - 2º Ano - 1º Semestre
14 de Dezembro de 2016

Índice

1 Descrição do problema

O problema enunciado consiste na procura de caminhos entre palavras. Estes caminhos são, na verdade, uma sequência de palavras de tamanho igual no qual cada palavra se obtêm mudando uma ou mais letras da palavra anterior, até chegar à palavra desejada. Para além disso, todas as palavras pertencentes ao caminho têm obrigatoriamente de pertencer a um dicionário. Dicionário é definido neste caso como uma lista de palavras, que não têm obrigatoriamente de ser todas do mesmo tamanho.

Como é possível a existência de vários caminhos, distingue-se cada passo do caminho pelo número de caracteres que são mudados. No fim, os custos dos caminhos somam-se para chegar ao custo final. Por exemplo, um passo no qual se mude um caracter têm o custo de 1^1, um passo em que se mudam dois caracteres têm um custo de 2^2, e assim sucessivamente.

O objectivo final deste projecto é, dado duas palavras e o número máximo de trocas que se podem efectuar em cada passo, retornar o caminho de menor custo entre essas duas palavras.

2 Abordagem do problema

Para resolver o problema, decidiu-se dividi-lo nas seguintes partes: guardar as palavras do dicionário e procurar o caminho propriamente dito. Após uma análise do pedido e das ferramentas disponíveis para o resolver, chegou-se então à conclusão de que seria necessário guardar as palavras do dicionário numa estrutura de dados que fosse facilmente acessível, e que não seria necessário guardar todas as palavras do mesmo, apenas as relevantes à resolução do problema. Analogamente se concluiu que não seria necessário criar todos os grafos nem representar nestes todas as palavras de um determinado tamanho.

2.1 Procura do caminho de menor custo

A escolha óbvia para resolver este problema foi recorrer ao algoritmo de Dijkstra, que procura o caminho mais curto entre dois vértices de um dado grafo desde que nenhuma das arestas do mesmo tenha peso negativo, o que é o caso deste problema. Para podermos utilizar então o algoritmo, chegou-se então à conclusão que é necessária a criação de um grafo representado por listas de adjacências (o algoritmo não é igual para matrizes de adjacência).

A utilização deste algoritmo também pressupunha a construção de um acervo e das funções necessárias à sua manipulação.

Tal como é mais tarde enunciado, foram sendo realizadas otimizações e alterações ao longo da elaboração deste projeto tendo esta contudo partindo do acima descrito.

3 Implementação do programa

A resolução do problema foi concretizada através da elaboração do programa cujo fluxograma se encontra representado na Figura 1.

O programa implementado foi dividido em três módulos diferentes, um dedicado às estruturas de dados e às operações sobre elas realizadas, o equivalente ao ficheiro datastructs.c, um que se relaciona diretamente com o problema a resolver, ou seja o ficheiro words.c, e por fim um com todas as restantes funções que não se inseriam em nenhum dos anteriores grupos referidos, o ficheiro utils.c. Tal como pode ser observado na figura acima, a resolução do problema é realizada na função *problemSolver*, que se encontra no ficheiro words.c, sendo esta constituída por uma série de funções que dividem a resolução do problema em várias tarefas mais específicas que facilitam à compreessão do racíocínio aplicado.

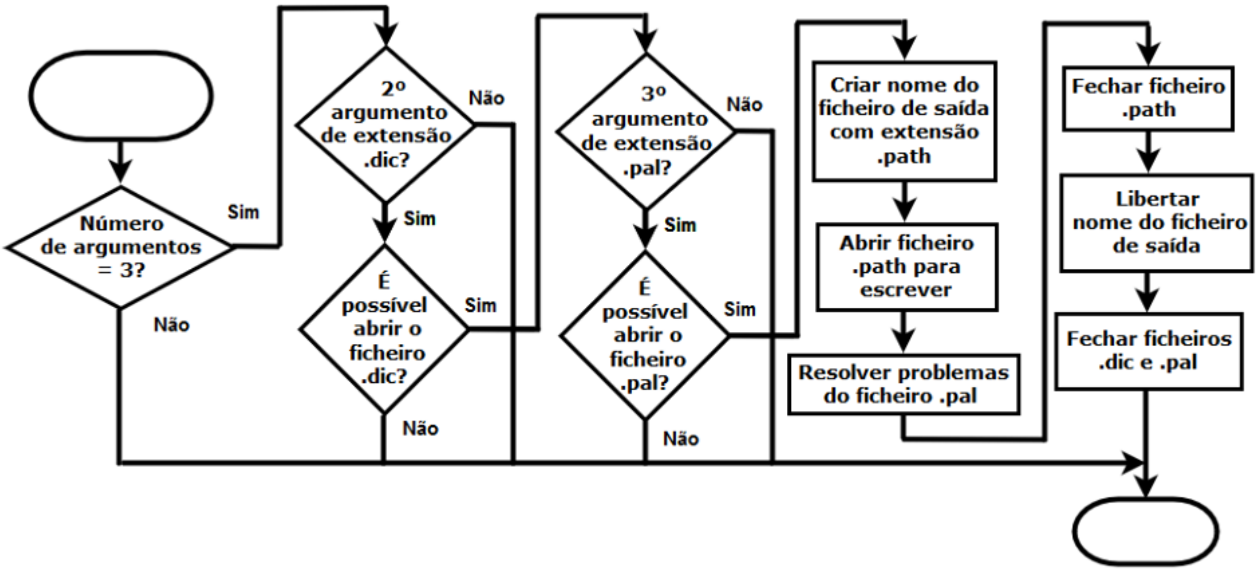

Figura 1: Fluxograma da função main do programa desenvolvido.

Isto é evidente na Figura 2, onde se encontra representado o fluxograma desta mesma função.

Esta função e as primeiras três funções suas constituintes estão disponíveis no ficheiro words.c visto estarem dedicadas diretamente à resolução do problema em questão. Considerando em primeiro lugar a função *initDictionary*, é logo possível inferir pelo nome que se trata de uma função relacionada com a construção do dicionário usado ao longo do programa.

De facto esta função é constituída por três funções, também estas no ficheiro words.c, que permitem a construção e preenchimento de uma matriz dicionário que contém para cada tamanho de palavra cujo problema implica a criação de um grafo, as palavras presentes no ficheiro .dic desse mesmo tamanho.

De seguida, na função *problemSolver* é abordada a necessidade da criação de grafos, sendo alocada memória para um vector de grafos que irá conter todos os utilizados durante o programa, sendo estes criados e preenchidos, caso sejam necessários, na função seguinte, *initGraphs*. Nesta função, para cada tamanho de palavra relevente, isto é, para cada tamanho cujos problemas não podem ser resolvidos diretamente, será criado um grafo implementado com listas de adjacência.

É de notar que serão colocadas na lista de adjacência de cada palavra apenas aquelas que diferem desta um número de caracteres menor ou igual ao número máximo de caracteres mudados entre duas palavras de um determinado tamanho, tendo estes sido previamente determinados e guardados num vector na função *initDictionary*. Desta forma não só se garante o funcionamento do algoritmo a implementar como também se descarta a possibilidade de colocar todas as palavras de um certo tamanho na lista de cada palavra individual.

Criados os grafos é então considerada a função *solveAllProblems* que irá ler o ficheiro de problemas e resolve-los um a um, imprimindo os seus resultados para um ficheiro de saída. Nesta função serão resolvidos diretamente os casos que dispensam a necessidade de cálculo de caminho, isto é, os casos onde as palavras do problema ou são a mesma ou diferem em apenas um caractere. Em ambos estes casos o caminho, assim como o seu custo, é imediato sendo então escrito logo no ficheiro de saída.

Nos restantes casos será utilizado o algoritmo de Dijkstra que irá determinar o caminho mais curto. Antes de mais será determinado o índice da palavra de origem e da palavra final no vector com palavras do mesmo tamanho, da Matriz Dicionário sendo então assim possível proceder ao cálculo do caminho.

O funcionamento do algoritmo implementado está abaixo representado, na Figura 3.

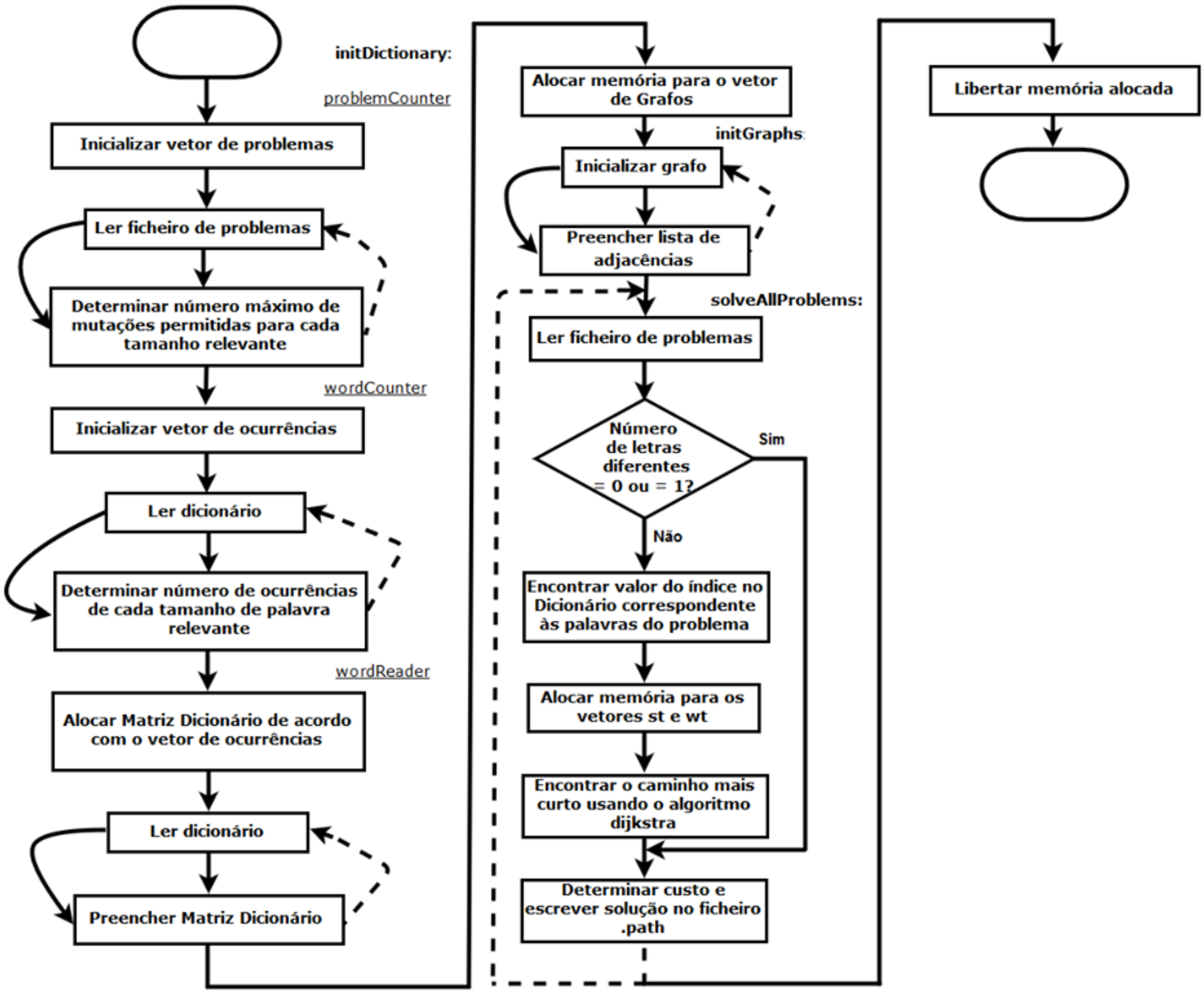

Figura 2: Fluxograma da função problemSolver.

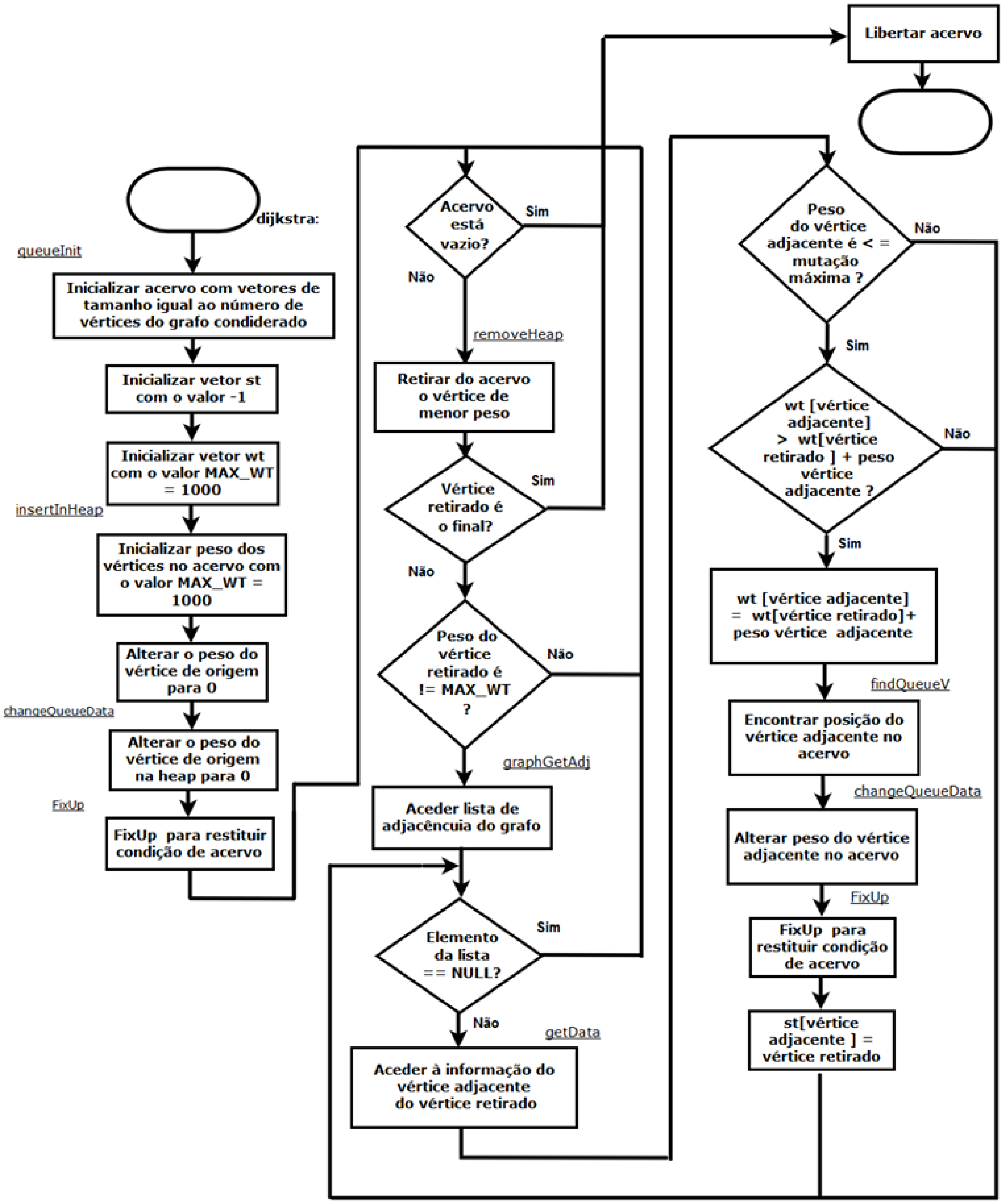

Figura 3: Fluxograma do algoritmo de Dijkstra.

Após determinado o caminho este será então escrito, assim como o seu custo associado, no ficheiro de saída. Tanto a funçao que calcula o custo, calculateTotalCost, como função que escreve o caminho, a função *printPath*, se encontram também no ficheiro words.c, sendo a função *printPaths* uma função recursiva de modo a garantir a correta ordem da escrita da solução.

Nas secções seguintes deste relatório serão abordados com maior atenção os restantes subsistemas criados assim como as estruturas de dados implementadas.

3.1 Descrição das estruturas de dados

A estrutura de dados que foi considerada a mais adequada para guardar o dicionário foi uma matriz, ou seja um vector de vectores, para diminuir os custos de memória e de acesso.

Como referido acima, foi claro que seria necessária a representação do grafo tendo como base uma lista de adjacências, então foi implementada uma lista genérica. Esta é definida usando $void*$ (definido em datastructs.h, e a partir daqui referido como Item), para os quais se pode passar qualquer ponteiro, assegurando a generalidade da implementação. O grafo em si é um grafo adirecional e acíclico, o que facilitou a implementação.

As listas definidas foram então utilizadas na definição do grafo pois este é implementado usando uma lista de adjacências. Os grafos foram definidos de forma genérica, com um reparo - a implementação dos dados usados na lista estão visíveis ao cliente. Todos os novos elementos desta lista são inseridos na primeira posição (head).

Como é possível haver a necessidade de construir um grafo para cada tamanho de palavras, decidiu-se usar um vector para guardá-los, no qual o índice corresponde ao tamanho das palavras representadas no grafo (i.e. all_arrays[3] contêm o grafo das palavras de tamanho 3).

Por fim, a descrição do algoritmo de Dijkstra requer um acervo, também este definido de forma genérica no código. Neste caso, o acervo foi implementado usando uma fila prioritária, que foi por sua vez implementada usando um vector. Este vector tem um tamanho fixo, definido na sua incialização, mas usou-se outra vaŕiavel (q→first) para que exista um "vector virtual" dentro do vector maior. Outra das variáveis que foi útil foi um vector indexado por vértices que diz a posição de um vértice no acervo (q→vert_pos), o que evita que se tenha de fazer procura linear cada vez que se quer encontrar um vértice no acervo.

3.2 Descrição dos algoritmos utilizados

O algoritmo utilizado para a criação do grafo é de complexidade $O(n^2)$, visto que não foi possível encontrar uma solução mais simples computacionalmente. Porém, não é necessário comparar todas as palavras a todas as palavras se se usar o algoritmo descrito pelo pseudo-código abaixo:

```
para i pertencente ao dicionário:
  para todo j < i:
    se i e j tiverem menos ou o mesmo número de letras diferentes que o custo máximo:
      ligar j a i
      ligar i a j
```

Este código reduz a complexidade do algoritmo de criação do grafo para $O(\frac{n^2}{2})$, o que não é ideal mas é melhor do que a alternativa.

Contudo, é claro que o algoritmo crucial ao programa foi o algoritmo de Dijkstra, usando a variação que encontra o caminho mais curto entre dois vértices e não a que encontra o caminho mais curto entre um vértice e todos os outros vértices do grafo, ou seja, parando quando o elemento que sai do acervo é o vértice a que se quer chegar.

Por fim, para a impressão da solução do problema para o ficheiro foram utilizados mais dois algoritmos: um para calcular o custo total e um para descobrir qual o caminho mais curto a partir do vector st, que é um argumento da função que implementa o algoritmo de Dijkstra.

O cálculo do custo consiste em, com o vector st, descobrir todos os passos da solução (a ordem dos passos é indiferente). Assim, soma-se então o custo de cada passo até que o valor do vector st seja -1, ou seja, que se tenha chegado ao vértice inicial. A impressão do custo teve de ser pensada de forma diferente, visto que a ordem é relevante neste caso. Por isso, para garantir que os resultados estariam correctos, considerou-se que a forma mais simples seria implementar recorrendo a um algoritmo recursivo.

Este algoritmo, sendo recursivo, necessita de uma condição de saída. Neste caso, a função retorna sem chamar-se a si mesma se argumento passado equivale ao índice do vector que representa o vértice original. Caso esta condição não se verifique, a função chama-se a si mesma e imprime a palavra equivalente para o ficheiro de saída.

4 Subsistemas funcionais

Como já referido, existem três módulos implementados:

datastructs.c Implementa os três tipos de dados essenciais - listas ligadas, grafos e acervos. Inclui as funções requiridas por cada tipo de dados, como a inicialização, libertação de memória e funções de interface para que o cliente possa interagir com as estruturas.

utils.c Funções variadas que não fariam sentido estar noutro ficheiro. Tem funções que lidam com a abertura de ficheiros, comparações, e a implementação do algoritmo de Dijkstra.

words.c Aqui encontra-se tudo aquilo que se relaciona diretamente com a resolução do problema, como por exemplo a determinação da maior permutação e a leitura das palavras dos ficheiros de dicionário e problemas.

4.1 datastructs.c

Funções de manipulação de listas simplesmente ligadas:

- node * initLinkedList();

  Função que inicializa uma lista simplesmente ligada.

- node * newNode(Item data, node *next);

  Função que cria um novo nó de uma lista simplesmente ligada.

- Item getData(node *cur);

  Função que devolve a data num nó de uma lista.

- node * insertList(node *first, Item item);

  Função que cria e insere um novo nó de uma lista simplesmente ligada no início da lista.

- node * nextNode(node *cur);

  Função que devolve o próximo nó de uma lista simplesmente ligada.

- node * changeNodeData(node *old, Item new_data);

  Função que altera a data de um nó de uma lista.

- void freeLinkedList(node *head, void (* freeItem)(Item));

  Função que liberta a memória alocada para uma lista simplesmente ligada.

Funções de manipulação de grafos implementados com lista de adjacência:

- graph * graphInit(int vert_num);

  Função que inicializa um grafo com um número de vértices introduzido.

- int graphGetVert(graph *g);

  Função que devolve o número de vértices de um grafo.

- node ** graphGetAdj(graph *g);

  Função que devolve o vector de listas de adjacência de um grafo.

- int getGraphVertex(Item info);

  Função que devolve o valor do vértice num node de uma lista de adjacência.

- int getGraphWeight(Item info);

  Função que devolve o peso do vértice num node de uma lista de adjacência.

- void freeGraph(graph *g, void (* freeItem)(Item));

  Função que liberta a memória alocada para um grafo. Liberta ainda cada lista de adjacência.

Funções de manipulação de acervo:

- queue * queueInit(int size);

  Função que inicializa um acervo, tendo os seus dois vectores tamanho igual ao introduzido na função.

- Item * queueGetData(queue *q);

  Função que devolve o vector data de um acervo.

- void changeQueueData(queue *q, int idx, Item new_value);

  Função que altera o conteúdo de um elemento do vector data de um acervo.

- int emptyHeap(queue *q);

  Função que devolve se um acervo está vazio ou não.

- void insertInHeap(queue *q, Item data, int (* compItem)(Item item1, Item item2));

  Função que insere data na primeira posição livre do vector data de um acervo e faz FixUp para recuperar a condição de acervo .

- void fixUp(queue *q, int idx, int (* compItem)(Item item1, Item item2)) ;

  Função que realiza FixUp de modo a recuperar a condição de acervo.

- void fixDown(queue *q, int idx, int n, int (* compItem)(Item item1, Item item2)) ;

  Função que realiza FixDown de modo a recuperar a condição de acervo.

- int findQueueV(queue *q, int vertex);

  Função que devolve a posição de um determinado vértice no acervo.

- int getQSize(queue *q) ;

  Função que devolve o atual tamanho do acervo.

- Item removeHeap(queue *q, int (* compItem)(Item item1, Item item2));

  Função que retira o elemento de menor peso do vector data do acervo, e reordena através do FixDown de modo a manter a condição de acervo.

- void freeHeap(queue *q) ;

  Função que liberta a memória alocada para um acervo.

Função que cria um novo elemento g_data *:

- g_data * newGData(int weight, int vertex)

  Função que cria um novo elemento do tipo g_data *, o preenche e o devolve.

4.2 utils.c

- void * allocate(size_t);

  Função que funciona como o malloc mas com verificação de erros.

- FILE * fcheck(char *, char *);

  Verifica se um ficheiro tem uma determinada extensão e abre-o de seguida.

- char* outputFileExtension(char * name_input);

  Dado o nome de um ficheiro, retorna o nome do ficheiro com uma extensão diferente (a extensão está *hard-coded*).

- int calculateDifferentLetters(char *word1, char *word2, int cost);

  Dadas duas palavras, calcula o número de letras diferentes entre elas. O parâmetro cost é uma otimização - se o custo já ultrapassar o custo máximo que se quer calcular já não vale a pena iterar pela palavra.

- int compInts(Item i1, Item i2);

  Dados dois inteiros, retorna 1 se o primeiro for maior do que o segundo, -1 se o segundo for maior do que o primeiro e 0 se forem iguais.

- int compWeight(Item item1, Item item2);

  De forma semelhante à função acima, dados dois g_datas, retorna 1 se o peso do primeiro for maior do que o segundo, -1 se o peso do segundo for maior do que o primeiro e 0 se os pesos forem iguais.

- void lowerWeight(queue *q, int idx, Item new_weight);

  Serve para baixar a prioridade de algo numa fila prioritária - neste caso, baixa o peso de um elemento da fila.

- void dijkstra(graph *g, int s, int end, int max_step, int *st, int *wt);

  Implementa o algoritmo de Dijkstra, usando uma fila prioritária.

- void writefirstOutput(FILE * fp, char * word, int cost);

  Escreve a primeira linha da solução para o ficheiro, visto que é diferente das restantes.

- void writeOutput(FILE * fp, char * word);

  Escreve uma linha da solução para o ficheiro.

- void freeMatrix(char ***mat, int *size, int init_size);

  Liberta uma matriz de caracteres. Note-se que assume-se que o tamanho da matriz pode ser variável, por isso é que se passa o vector size.

4.3 words.c

- void problemCounter(FILE *prob, int *problem_array);

  Itera pelo ficheiro de problemas, contando qual a iteração máxima para cada tamanho de palavra.

- void wordCounter(FILE *input, int *occurrences, int *problems);

  Itera pelo ficheiro de dicionário, contando quantas palavras de cada tamanho existem. Só conta tamanhos de palavras relevantes para a resolução do problema.

- void wordReader(FILE *input, char **output[MAX_STRING], int *size_array);

  Sabendo quantas palavras existem de cada tamanho, guarda as palavras relevantes para a resolução numa matriz de caracteres.

- void initDictionary(FILE *prob, FILE *dic, char **dictionary[MAX_STRING], int *to_solve, int *word_count);

  Chama as três funções descritas acima, para facilitar a criação da matriz de palavras.

- void initGraphs(graph **all_graphs, int *max_change, int *size_array, char ***dict);

  Inicializa os grafos necessários para a resolução do problema, tendo em conta a permutação máxima entre palavras.

- void printPath(FILE *output, int w_size, int *st, int origin_v, int final_v, char **dic[MAX_STRING], int cur);

  Imprime o caminho entre duas palavras, usando um algoritmo recursivo. Não imprime nem a primeira nem a última palavra.

- int calculateTotalCost(int *st, int final_v, char **dic);

  Calcula o custo total do caminho, dado o vector st e o vértice final.

- void solveAllProblems(FILE *input, FILE *output, graph **all_graphs, char **dictionary[MAX_STRING], int *size_array);

  Itera pelos problemas do ficheiro, lida com eles da forma apropriada (chamando ou não o algoritmo de Dijkstra) e chama as funções que escrevem os resultados para o ficheiro.

- void problemSolver(FILE *dic, FILE *prob, FILE *path);

  A função principal do problema - chama as funções que criam os grafos, o dicionário, que resolvem os problemas e que libertam a memória.

- void freeAllGraphs(graph **all_graphs);

  Resolvidos todos os problemas, liberta os grafos um a um seguidos do vector que os guardava.

5 Complexidade do programa

Examinando o programa, chega-se à conclusão de que existem duas grandes partes que contribuem para a complexidade. Em ambos os casos, v representa o número total de palavras com o mesmo tamanho e e representa o número de ligações que existem entre palavras.

Criação de grafo Para inserir todas as palavras no grafo, é necessário comparar todas as palavras duas a duas. No código, como para cada palavra do dicionário só é necessário comparar as palavras anteriores a esta, o que acontece é $1 + 2 + 3 + ... + v = \frac{v^2}{2} + \frac{v}{2}$, que é majorado por v^2 concluindo-se então que a complexidade será $O(v^2)$.

Em termos de memória, aloca-se um vector de ponteiros para grafo, que podem ou não apontar para NULL dependendo da existência do grafo. Cada grafo vai ter a sua própria lista de adjacências, alocando então um vector de ponteiros para node, que também podem ou não apontar para NULL. No caso de não apontarem para NULL, apontam então para a cabeça de uma lista de tamanho variável que contêm todos as palavras a que uma dada palavra está ligada. Então, conclui-se a complexidade em termos de memória será de $O(e^2)$, visto que numa lista de adjacências de um grafo não direccionado uma ligação é reperesentada por dois nós.

Algorítmo de Dijkstra No caso do problema a resolver, observa-se que o grafo que representa as ligações é um grafo esparso, isto é, com menos de $|v^2|$ ligações. Assim, esta implementação do algoritmo de Dijkstra, usando uma lista de adjacências e um acervo, corre em $\Theta((|e|+|v|)log(|v|))$, como há uma estrutura de dados auxiliar que indica a posição de cada vértice no acervo, fazendo com que a procura seja $O(1)$.

Todas as vezes que a função que implementa o algoritmo de Dijkstra é chamada, aloca-se memória para o acervo, que depende do número de palavras com o mesmo tamanho. Cada elemento saído do acervo é logo libertado após o fim da sua avaliação, sendo o acervo libertado no fim da chamada, visto já não ser necessária.

6 Análise crítica do programa

À supefície, o problema aparenta ser simples - no entanto é necessária alguma cautela para alguns casos limite e para alguns descuidos que podem aumentar e muito o tempo de execução do problema. Por exemplo, após uma implementação inicial do algoritmo de Dijkstra, observou-se que o tempo de execução era muito superior ao esperado. Um dos erros efectuados incialmente envolvia a procura de um determinado vértice dentro do acervo - a abordagem *naïve* utilizada inicialmente tinha complexidade $O(n)$, o que considando as vezes que era necessário descobrir o vértice, "estragava" a rapidez do algoritmo de Dijkstra.

Outro problema, que foi mais trivial de resolver, foi o facto de que não foram antecipados certos tipos de problemas, nomeadamente que a palavra de partida fosse igual à palavra de chegada e que a diferença entre as palavras fosse apenas de uma letra.

Porém, a parte menos direta do projecto foi de facto implementar as estruturas de dados de forma genérica, ou seja, que pudessem ser facilmente utilizadas para outros programas. Como a implementação estava incluída no ficheiro datastructs.c e não em datastructs.h, o compilador era incapaz de aceder a dados que se encontrassem dentro da estrutura de forma direta - de modo que foi necessário criar uma função de acesso para cada variável de cada estrutura.

Para além disso, foi necessário passar como argumento funções auxiliares - isto é funções escritas à parte, desenhadas para lidar com cada tipo específico, como por exemplo funções de comparação. Como os argumentos das funções e os tipos de dados estavam definidos para usar Items, não foi possível tomar certas coisas como certas (e.g. funções de comparação). Estas últimas causaram alguns problemas graças aos ponteiros.

Apesar destes pequenos contratempos, todas as decisões, quer em termos de estruturas de dados utilizadas, quer em termos de lógica, foram tomadas visando obter um melhor equilíbrio em termos da complexidade do implementado e da quatidade de memória utilizada.

Visto que o programa passou nos 20 testes aplicados considera-se que o programa criado teve um desempenho satisfatório.

7 Exemplo

De seguida será ilustrado o funcionamento do programa de forma a demonstrar o racíocinio implementado, acima descrito.

É de notar que se pressupõe que o ficheiro .dic se trata de um ficheiro com palavras não acentuadas, sem hífen, separadas por espaços ou *newlines* e que inclui todas as palavras que se encontram no ficheiro de problemas. Sendo o ficheiro de dicionário deste tipo, com nome dexemplo.dic:

```
gotas bisturi agente corpo
pausa trampolim parte zebra casta
cobertor liberdade casto gorro
```

Em relação ao ficheiro .pal é então presumido, por sua vez, que este contem para cada problema duas palavras do mesmo tamanho e um inteiro, sendo este o custo associado ao caminho. Sendo este pexemplo.pal:

```
aguenta aguento 3
gotas casta 3
cobertor cobertor 4
zebra pausa 2
```

O programa desenvolvido irá antes de mais confirmar as extensões dos ficheiros introduzidos na linha de comandos. Visto que ambos possuem a extensão correta será criado o nome do ficheiro de saída com o mesmo do .pal mas com a extensão .path, ou seja pexemplo.path.

De seguida será lido o ficheiro pexemplo.pal e criado um vector, problem_array, inicializado a zero com tamanho MAX_STRING = 100, com o custo máximo de mutação para cada tamanho de palavra representado no ficheiro de problemas relevante à resolução. Isto é, na posição 5 terá o valor 3 visto ser a mutação máxima permitida entre palavras de tamanho 5 e tanto na posição 6 como na 8, manterão o valor 0 visto que os únicos problemas envolvendo palavras de tamanho 6 e de tamanho 8 podem ser resolvidos diretamente sem a necessidade de grafos. Na função seguinte será preenchido o vector de número de ocurrências de tamanhos de palavras no ficheiro .dic, estando este inicializado a zero.

Visto que o único tamanho com um valor na posição respectiva do problem_array maior que zero é o 5 serão contadas o número de palavras de tamanho 5 no ficheiro, ficando então na posição 5 deste vector o valor 7.

Seguidamente será então criada a matriz que representa o dicionário, que terá, na posição 5:

```
gotas
corpo
pausa
parte
zebra
casto
gorro
```

Após alocada memória para um vector de grafos, que apenas irá conter um grafo com as palavras de tamanho 5, será preenchida a sua lista de adjacência tendo cada palavra na sua lista apenas palavras que dela diferem 3 ou menos caracteres.

```
Adj[0] para gotas :  gorro
Adj[1] para corpo :  gorro -> casto
Adj[2] para pausa :  parte
Adj[3] para parte :  casto -> pausa
Adj[4] para zebra :  NULL
Adj[5] para casto :  parte -> corpo
Adj[6] para gorro :  corpo -> gotas
```

Agora que já foi o grafo necessário à resolução do problema será lido o ficheiro .pal e resolvidos os problemas um a um. Para o primeiro as palavras do problema apenas apresentam uma diferença de um caracter, sendo então a sua solução imediatamente escrita no ficheiro .path como:

```
aguenta 1
aguento
```

Para o segundo problema o mesmo já não se verifica sendo portanto necessaŕio recorrer ao algoritmo de Dijkstra. Antes de mais, serão comparadas as palavras do problema com as do mesmo tamanho inseridas no dicionário, de modo a determinar o valor do vértice inicial e final correspondente ao índice destas palavras no vector de palavras do mesmo tamanho contido na matriz que representa o dicionário.

Neste caso o vértice inicial terá o valor 0 e o final o valor 5.

Já no algoritmo será inicalizada o acervo, cujos vectores terão ambos um tamanho inicial de 7, sendo o peso de cada vértice definido como MAX_WT. Serão ainda inicializados os vectores st e wt, também de tamanho 7, com o valor -1 e MAX_WT, respetivamente. O peso do vértice inicial 0, tanto no acervo como no vector wt, passará para 0 e será de seguida realizado o fixUp do acervo para o restaurar.

Na tabela 1 apresenta-se uma ilustração da evolução do acervo ao longo do algoritmo.

Acervo Inicializado:	0	1	2	3	4	5	6
Após alterada prioridade:	**0**	**1**	**2**	**3**	**4**	**5**	**6**
Removido o de menor peso:	1	3	2	6	4	5	
st[6] = 0 ; wt[6] = 9 Depois de alterado e realizado FixUp:	6	1	2	3	4	5	
Removido o de menor peso:	1	3	2	5	4		
st[1] = 6 ; wt[1] = 13 Depois de alterado e realizado FixUp:	1	3	2	5	4		
Removido o de menor peso:	3	5	2	4			
st[5] = 1 ; wt[5] = 22							

Tabela 1: Evolução do acervo para o problema 2

É de notar que após a remoção do vértice de menor peso do acervo, serão verificados os vértices seus adjacentes para a otimização do caminho. Isto é, depois de retirado o vértice 0 será analisado o vértice 6 que apresenta um melhor caminho passando este a ser o de menor peso no acervo. Analogamente será considerado o vértice 1 e por fim o vértice 5 que corresponde ao vértice final.

Índice:	0	1	2	3	4	5	6
st [Índice]:	-1	6	-1	-1	-1	1	0

Tabela 2: Vetor st solução do problema 2

Quando o vértice retirado do acervo corresponder ao vértice final é possível então terminar este processo, estando o caminho representado no vector st.

O vector st obtido está representado na tabela 4.

Obtido este vector será impresso o caminho no ficheiro .path, sendo esta escrita realizada rescursivamente.

```
gotas 22
gorro
corpo
casto
```

Para o terceiro problema a situação é trivial e análoga à do primeiro problema, visto que as duas palavras são iguais, sendo portanto resolvida diretamente e impressa no ficheiro .path como:

```
cobertor 0
cobertor
```

O quarto problema é, por sua vez, análogo ao segundo problema, estando a evolução do acervo representada na Tabela 3.

Acervo Inicializado:	0	1	2	3	4	5	6
Após alterada prioridade:	**4**	**0**	**2**	**3**	**1**	**5**	**6**
Removido o de menor peso:	0	3	2	6	1	5	
Removido o de menor peso:	3	6	2	5	1		
Removido o de menor peso:	6	5	2	1			
Removido o de menor peso:	5	1	2				
Removido o de menor peso:	1	2					
Removido o de menor peso:	2						

Tabela 3: Evolução do acervo para o problema 4

Neste caso não foi encontrado um caminho que satisfizesse o custo máximo permitido entre mutações, isto é, no máximo 2 caracteres de uma vez.

Índice:	0	1	2	3	4	5	6
st [Índice]:	-1	-1	-1	-1	-1	-1	-1

Tabela 4: Vetor st solução do problema 4

Obtido este vector será indicado no ficheiro .path a inexistência de caminho da seguinte forma:

```
zebra -1
pausa
```

8 Bibliografia

Na realização deste projeto foram consultados os seguintes *websites*:

- http://www.algolist.net/
- http://stackoverflow.com/
- http://www.cprogramming.com/algorithms-and-data-structures.html