## 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** **3**

**2 Abordagem do problema** **3**
2.1 Procura do caminho de menor custo . . . . . 3

**3 Implementação do programa** **3**
3.1 Descrição das estruturas de dados . . . . . 3
3.2 Descrição dos algoritmos . . . . . 4

**4 Subsistemas funcionais** **4**

**5 Complexidade do programa** **4**

**6 Análise do programa** **5**

**7 Exemplo** **5**

**8 Bibliografia** **5**

# 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. Chegou-se então à conclusão de que seria necessário guardar as palavras do dicionário num local que fosse facilmente acessível, e que não seria necessário guardar todas as palavras, apenas as relevantes, tal como não é necessário fazer todos os grafos.

## 2.1 Procura do caminho de menor custo

A escolha óbvia para resolver este problema foi recorrer ao algorítmo 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 algorítmo, chegou-se então à conclusão que é necessária a criação de um grafo representado por listas de adjacências (o algorítmo não é igual para matrizes de adjacência).

# 3 Implementação do programa

Dividiu-se então o programa em três partes distintas: uma parte referente às estruturas de dados, uma parte que se relaciona diretamente com o problema a resolver, e por fim todas as restantes operações que não pertencem a nenhum dos anteriores grupos. No código, estas partes equivalem aos ficheiros datastructs.c, words.c e utils.c respectivamente.

Para evitar alocações de memória desnecessárias, percorrem-se todas as linhas do ficheiro de dicionário duas vezes. Uma para contar o número de palavras que existem de cada tamanho de palavra, outra para copiar as palavras do ficheiro para a matriz que representa o dicionário, cujo tamanho é alocado baseando-se no número obtido anteriormente. Assumiu-se que nunca existiriam palavras de mais de cem letras.

Como mencionado na secção anterior, rapidamente chegou-se à conclusão de que é desnecessário criar grafos que não vão ser utilizados. Por isso, também percorre-se uma vez pelo ficheiro de problemas sem os começar a resolver, para saber que tamanhos de palavras existem no ficheiro, e qual número máximo de caracteres que mudam entre duas palavras do caminho.

## 3.1 Descrição das estruturas de dados

Como referido acima, foi claro que seria necessária a criação de 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.

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 terá de existir um grafo para cada tamanho de palavras, decidiu-se usar um vetor 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 algorítmo de Dijkstra requer um acervo, também este definida de forma genérica no código. Neste caso, o acervo foi implementado usando uma fila prioritária, que por sua vez foi 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 onde o vértice está na heap (q→vert_pos), que evita que se tenha de fazer procura linear cada vez que se quer encontrar um vértice na heap.

### 3.2 Descrição dos algoritmos

Claramente, o algorítmo 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 da heap é o vértice a que se quer chegar.

## 4 Subsistemas funcionais

Como já referido, existem três subsistemas funcionais:

**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 algorítmo de Dijkstra.

**words.c** Encontra-se tudo 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.

## 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, a complexidade 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 (menos do que $|v^2|$ ligações). Assim, esta implementação do algorítmo 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 aonde está cada vértice na heap, fazendo com que a procura seja $O(1)$.

# 6 Análise do programa

Na supefície, o problema é 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. Um destes descuidos foi a procura de um determinado vértice dentro da *heap* - a abordagem *naïve* utilizada inicialmente tinha complexidade $O(n)$, o que considando as vezes que era necessário descobrir o vértice, "estragava" a rapidez do algorítmo 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.

# 7 Exemplo

Considerando como exemplo um ficheiro de problemas com:

```
massa pasta 3
pasta pasta 1
pasta parte 2
```

E considerando que um dicionário 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, como por exemplo:

```
juiz largo
massa parte
pasta rei
   ...
```

Então, os passos que se efectuam para resolver o ficheiro de problemas enunciado são:

1. Lê-se o ficheiro de problemas de forma a identificar o número máximo de mutações entre palavras de um determinado tamanho, ignorando casos triviais como duas palavras iguais ou duas palavras que apenas diferem em uma letra.
2. Guarda-se as palavras com tamanhos relevantes à resolução numa matriz.

# 8 Bibliografia