Instituto Superior Técnico

Mestrado Integrado em Engenharia Electrotécnica e de Computadores

Algoritmos e Estruturas de Dados

2016/2017 – 2º Ano, 1º Semestre

Wordmorph

Grupo 14

Eduardo Alexandre Silva da Costa

84037 – eduardo.silva.costa@tecnico.ulisboa.pt

Eduardo Miguel Ferreira Cabral de Melo

84038 – eduardo.de.melo@tecnico.ulisboa.pt

Docente: **Carlos Bispo**

Índice

Descrição do problema

Foi-nos proposto a realização de um programa que, dado um ficheiro dicionário de caracteres latinos e um ficheiro com pares de palavras, achasse o "caminho" de palavras de menor custo entre os pares, estando todos os pares de palavras no dicionário. Estes pares de palavras teriam obrigatoriamente o mesmo tamanho, e o caminho entre estas seria uma sequência de palavras também com o mesmo tamanho, as quais diferenciariam da imediatamente anterior por uma substituição de um ou mais caracteres. Esta substituição dá-se pelo nome de mutação. Cada mutação entre palavras tem um peso igual a n^2, sendo n o número de caracteres substituídos de uma palavra para a outra. Cada par de palavras no ficheiro de problemas tem um número máximo de mutações permitidas para achar o caminho mais curto, não podendo este ser excedido. Estes caminhos quando encontrados, são escritos num novo ficheiro, juntamente com o peso deste. Se não existir caminho, tal indicação também é escrita no ficheiro. É de notar que se assume que todos os ficheiros de problemas terão de ser válidos, não se efectuando nenhuma verificação da validade destes.

Abordagem ao problema

Começámos por verificar a validade dos argumentos de entrada, certificando-nos que estes existem e que a sua extensão é válida. Depois, lemos o dicionário e o ficheiro de problemas, para sabermos quais os tamanhos de palavras e número de mutações que vão ser necessários para a criação dos grafos. De seguida criam-se os grafos por lista de adjacências, 1 para cada tamanho de palavras necessário. Grafos criados, começam-se a resolver os problemas. Cada problema é lido e calcula-se o caminho de menor peso deste, através do algoritmo de procura **A***, que será descrito pormenorizadamente mais à frente. Encontrado o caminho escreve-se este no ficheiro, e quando todos os problemas são resolvidos, o programa termina.

Arquitectura do programa

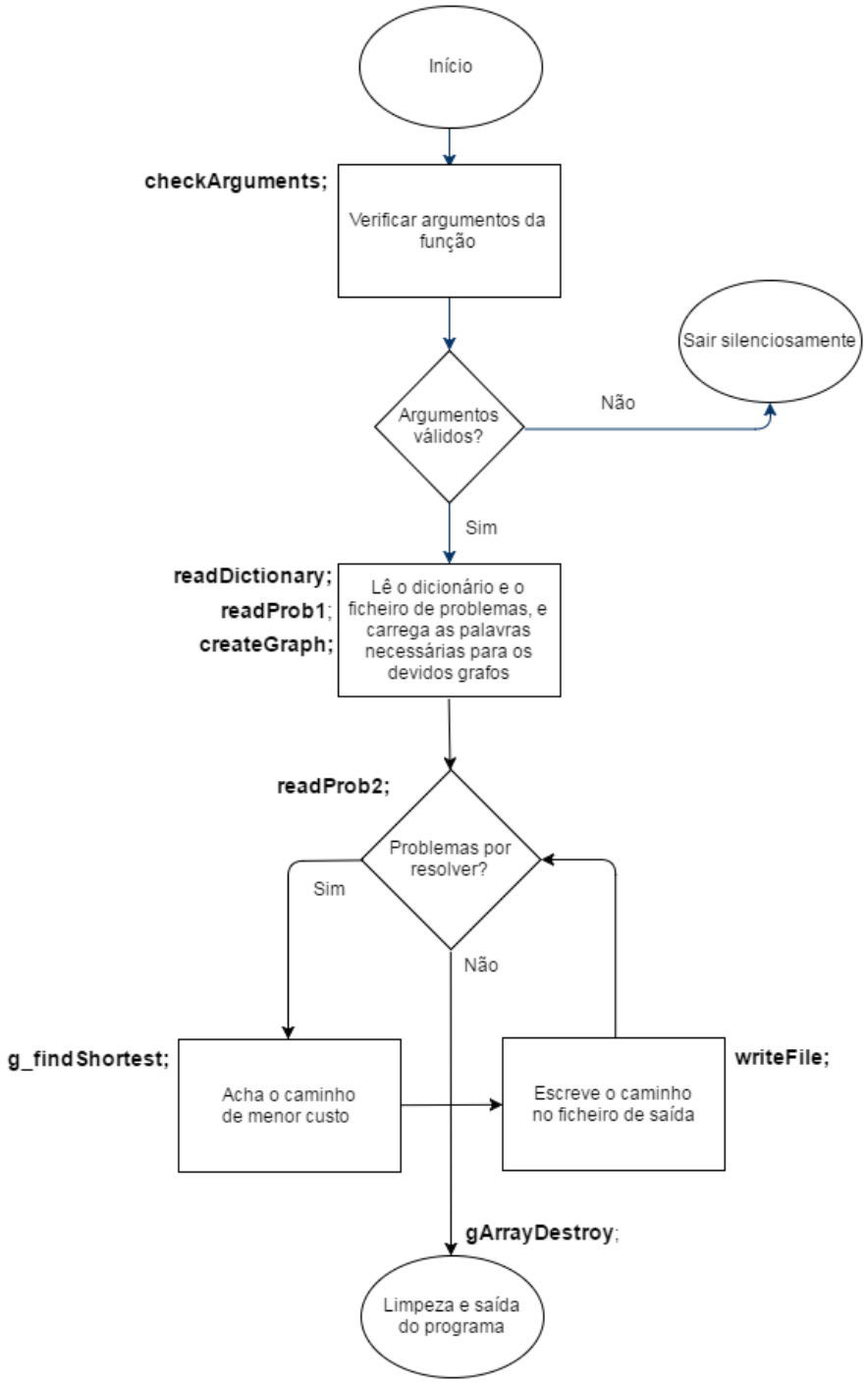

Figura 1

Fluxograma do programa Wordmorph

O fluxograma apresentado na figura acima, descreve o nosso programa muito resumidamente, focando-se apenas nas decisões e funções mais cruciais. Pode-se observar que este se encontra dividido em 2 módulos principais: carregamento de dados e tratamento de problemas.

Descrição das estruturas de dados

Primeiramente, criámos uma tabela de *struct _words,* cujo tamanho é igual ao número de caracteres da maior palavra no dicionário. Esta é indexada pelo tamanho das palavras (ou seja, a entrada 0 da tabela conterá somente palavras de tamanho 1, a entrada 1 palavras de tamanho 2, e assim sucessivamente). Esta tabela serve para guardar em memória as palavras do dicionário necessárias para a resolução dos problemas e posteriormente utilizada para construir grafos.

Assim sendo, criámos uma tabela de *struct _graph,* cujo tamanho também será igual ao número de caracteres da maior palavra do dicionário. Esta, tal como a tabela de *struct _words,* também é indexada pelo tamanho das palavras. Esta tabela serve para guardar em memória os grafos que virão a ser necessários para a determinação do caminho ótimo entre as palavras. A estrutura *graph* é um tipo de dados abstracto que, neste caso, vai ser utilizado para encontrar caminhos ótimos entre *strings*. Esta estrutura foi implementada através de uma tabela de listas de adjacências, devido a compromissos de memória. Esta tabela tem tamanho variável para cada grafo, sendo este igual ao número total de palavras guardadas no grafo correspondente (número de vértices). O grafo tem também uma tabela de *Items* (void *), com tamanho igual ao número de vértices.

As listas de adjacências na nossa implementação são listas de *Items* que, quando estes são acedidos, são interpretados como uma *struct _edge*.

Por fim, para realizar o algoritmo de procura nos grafos, foi necessária a implementação de um acervo utilizando uma tabela, sendo esta com o tamanho do número de vértices do maior grafo a ser utilizado. O acervo permite-nos organizar elementos por ordem de prioridade de uma forma bastante eficiente. Este acervo é representado por *struct _h_node*. Porém, nós criámos um acervo usando uma variável estática. Isto implica que só está disponível para utilização um acervo. Tal não prejudica o desempenho do nosso programa pois só é calculado um caminho de cada vez.

Descrição dos algoritmos

São usados 3 algoritmos principais, sendo estes encontrados nas funções **findIndex**, **g_findShortest**, e a **qsort** própria do C.

qsort

Esta função, implementada em C, implementa o algoritmo quicksort, que consiste na ordenação de uma dada tabela com certos elementos. Este algoritmo utiliza o método da divisão e conquista numa tabela com elementos do mesmo tipo. Para tal, efetuam-se os seguintes passos:

- Escolhe-se um pivot, ou seja, uma posição qualquer da tabela (ou sub-tabela);
- Percorre-se a tabela a partir da esquerda até encontrar um elemento maior ou igual ao pivot;
- Percorre-se a tabela a partir da direita até encontrar um elemento menor ou igual ao pivot;
- Trocam-se os 2 elementos;
- Repetem-se os 3 passos anteriores até os ponteiros que percorrem a tabela se cruzarem. Quando tal acontece, trocar o pivot com a posição mais à esquerda da tabela da direita, colocando-o na sua posição final;
- Repetir os passos anteriores todos recursivamente para as sub-tabelas com os elementos menores e com os elementos maiores.

findIndex

Esta função implementa o algoritmo da procura binária numa tabela de *strings*. Este algoritmo consiste na procura de um dado elemento numa tabela com elementos do mesmo tipo, sendo esta procura também efetuada pelo método da divisão e conquista. Assim sendo, serão realizadas sucessivas divisões da tabela ao meio, até ser encontrado o elemento procurado ou ser constatado que este não existe na tabela. É bastante importante que a tabela esteja previamente ordenada (no nosso

caso, ordenada alfabeticamente pela função **qsort**), pois se não estiver o algoritmo não funciona. Na figura abaixo, encontra-se um exemplo ilustrativo da execução do algoritmo:

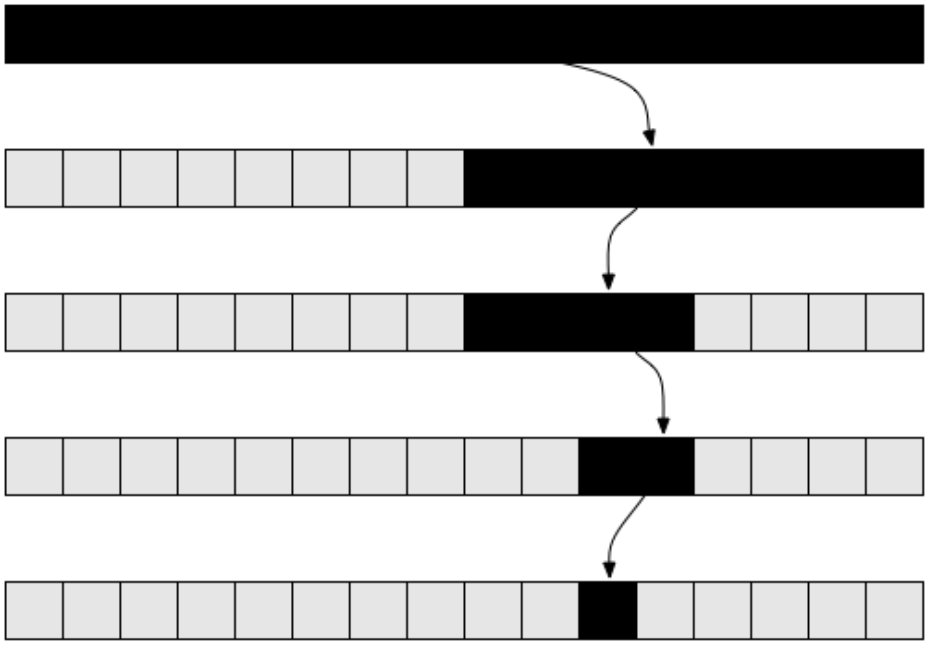

Figura 2

Exemplo de um algoritmo genérico de procura binária

A preto encontram-se representadas as zonas da tabela onde é possível estar localizado o elemento que estamos a procurar.

Portanto, quando a tabela é dividida ao meio, 1 de 3 coisas podem acontecer:

- A *string* que procuramos encontra-se na posição onde dividimos a tabela, e assim termina a procura;
- A *string* que procuramos é "alfabeticamente menor" que a *string* na posição de divisão e encontra-se à esquerda da divisão;
- A *string* que procuramos é "alfabeticamente maior" que a *string* na posição de divisão, e encontra-se à direita da divisão.

g_findShortest

Esta função implementa o algoritmo de procura de caminho **A***, ligeiramente adaptado ao nosso programa. Este algoritmo é usado para encontrar o caminho de menor custo entre dois vértices de um grafo, usando, no nosso caso, um acervo que lhe permita definir as prioridades dos vértices. Este algoritmo é uma variante do algoritmo de procura de *Dijkstra*, acrescentando a este uma função heurística para tentar prever qual o caminho de menor custo.

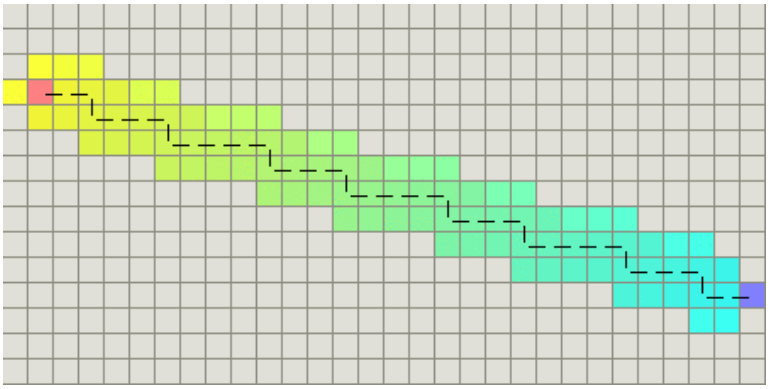

Figura 3

Exemplo de um algoritmo genérico **A***. Como se pode observar, em vez de avaliar todos os vértices adjacentes primeiro (como o de Dijkstra), avalia os que são mais prováveis de pertencer ao caminho de menor custo primeiro.

Peso real: peso acumulado pelas arestas do caminho + peso da aresta entre o vértice atual e o vértice a ser analisado.

Peso heurístico: peso real + peso estimado entre o vértice a ser analisado e o vértice final.

Abaixo encontram-se os passos usados na nossa implementação deste algoritmo na função **g_findShortest** (Nota: quando o peso heurístico de um vértice é alterado, a prioridade do acervo é atualizada. O mesmo ocorre quando é removido o vértice de maior prioridade do acervo, que está relacionada com o peso heurístico):

- Inicializar o acervo e todos os vértices com pesos real e heurístico infinitos, com excepção do vértice inicial, que terá como peso real zero e como peso heurístico a distância estimada entre o vértice inicial e o final;

- Remover o vértice de maior prioridade do acervo e verificar se este é o vértice final. Se for, ou se não houver nenhum vértice para remover, salta para o último passo.
- Verificar o peso da aresta entre o vértice de maior prioridade retirado e um vértice adjacente deste. Se este for maior que o peso máximo permitido, esta aresta não é válida para o cálculo do caminho e este passo será repetido com o próximo vértice adjacente. Se não existirem mais vértices adjacentes, retorna ao passo 2.
- Verificar se peso da aresta + peso real do vértice de maior prioridade retirado é menor que o peso real atual do vértice adjacente. Se assim for, este é atualizado, assim como o seu peso heurístico e a tabela de caminho. Volta ao passo 3.
- Retornar o peso real do último vértice (custo do caminho mais curto) e destruir acervo.

Este algoritmo é, no pior dos casos, igual ao algoritmo de procura de Dijkstra, se a distância retornada pela função heurística for menor ou igual à distância real entre os dois vértices. Se tal não acontecer, este algoritmo não tem garantias de achar o caminho mais curto. No entanto, na nossa implementação, a função heurística cumpre os requisitos para o algoritmo **A*** ser viável.

Descrição dos subsistemas

Carregamento de dados

Este subsistema lê os ficheiros dicionário e de problemas e utiliza a estrutura *words* para criar os grafos, utilizados na resolução dos problemas. As funções que compõem este sistema encontram-se essencialmente nos ficheiros ***file.h*** **e** ***file.c***.

file.c

checkArguments - Verifica se os argumentos da função são válidos, de maneira a que o programa seja inicializado corretamente.

readDictionary - Lê o dicionário e carrega as palavras necessárias para a memória.

readProb1 - Lê o ficheiro de problemas para fazer uma selecção dos tamanhos de palavras necessários de carregar para a memória.

cmpWords - Calcula o número de caracteres diferentes entre duas *strings* de tamanhos iguais.

createGraph - Inicia e carrega o grafo, utilizando as palavras previamente guardadas em memória.

Tratamento de problemas

Este subsistema é responsável pela leitura do ficheiro de problemas, pela resolução dos mesmos e pela escrita no ficheiro de saída dos caminhos de palavras obtidos. As funções que compõem este sistema encontram-se nos ficheiros ***file.h*** **e** ***file.c***, e também nos ficheiros **graph.c** e **graph.h**.

file.c

readProb2 - Lê o ficheiro de problemas.

findIndex - Retorna o vértice de uma dada *string* de uma tabela ordenada.

writeFile - Escreve o ficheiro de saída para problemas não triviais.

writeExplicit - Escreve o ficheiro de saída para problemas triviais.

graph.c

g_findShortest - Encontra o caminho mais curto entre dois vértices e retorna o peso deste.

Análise dos requisitos computacionais

Foram escolhidos tipos de dados abstractos de modo a respeitar os limites de memória impostos e de forma a minimizar o tempo de execução dos algoritmos.

Words

A memória ocupada pelas estruturas *Words* é proporcional ao número de palavras que esta guarda e ao tamanho destas. Porém, as tabelas de palavras destas estruturas são todas transferidas para os grafos, que irão guardar os ponteiros para as tabelas das estruturas *Words*.

Grafos

Os grafos utilizam a maior parte da memória ocupada pelo programa. Esta ocupação aumenta com o número de vértices, com o tamanho das palavras e com o número de mutações (arestas). Por exemplo, um grafo que permita mutações de 4 caracteres com palavras de tamanho 8, considerando que existem no dicionário aproximadamente 41.500 palavras com esse tamanho, atinge cerca de 900MB de pico de memória. A criação dos grafos na função **createGraph** localizada no ficheiro **file.c**, é das operações mais dispendiosas também em termos de tempo. Esta função tem uma complexidade de O(V^2), sendo V o número de vértices do grafo. É também de notar que quanto mais denso for o grafo (maior número de mutações permitidas), mais dispendiosa vai ser a função **createGraph**. É de notar que o grafo também contém uma tabela de lista de adjacências, que cresce em memória com o número de arestas do grafo. A complexidade do acesso a um dado *Item* de uma lista de adjacências no pior caso é O(A), sendo A o número de vértices adjacentes de um dado vértice, visto que como a lista é simplesmente ligada, terá que se percorrer todos os elementos desde o início da lista até ao elemento de chegada.

Acervo

O acervo ocupa quase nenhuma memória em comparação com a ocupada pelos grafos, sendo esta proporcional ao número de vértices do maior grafo. As funções **h_fixDown** e **h_fixUp** são de complexidade O(log V), tornando o acervo numa estrutura de dados eficiente, poupando uma quantidade significativa de tempo em comparação com uma fila prioritária.

Algoritmos

qsort

Este algoritmo implementado em C é utilizado para ordenar as tabelas de *strings*, de forma a posteriormente poder ser utilizada com sucesso o algoritmo de procura binária. Este algoritmo tem complexidade de caso médio O(n log n), de melhor caso também O(n log n), e de pior caso O(n^2).

findIndex

Esta função implementa o algoritmo de procura binária, utilizado para descobrir o número do vértice correspondente a uma dada palavra. Este algoritmo tem uma complexidade de caso médio O(log n), de pior caso também O(log n), e de melhor caso O(1).

g_findShortest

É a função que contém o algoritmo principal do programa, o algoritmo de procura **A***. Este algoritmo vai determinar qual o caminho com menor custo entre as palavras do ficheiro de problemas, que se encontram representadas nos vértices dos grafos. Este algoritmo tem uma complexidade no pior caso de O(E), sendo E o número de arestas do grafo.

Análise do desempenho do programa

Inicialmente o nosso programa era demasiado lento. Após uma investigação ao código, percebemos que estava a ser gasto tempo em demasia na criação dos grafos. Tal ocorria porque ao compararmos a diferença de caracteres entre duas palavras, caso esta fosse maior que o número máximo de mutações permitidas, continuava a comparar caracteres até ao final das *strings*, em vez de passar imediatamente à próxima aresta.

O programa calcula o caminho de menor custo em todos os testes fornecidos pela equipa docente, testes realizados por nós e também os testes finais de submissão. Estes testes de submissão final, não foram todos passados com sucesso. Tal deveu-se a exceder o limite de tempo permitido.

Não podemos concluir se o nosso ficheiro não passou nesses testes somente por falta de otimização do código, pois com 4 dias para terminar o período limite de submissão, foi-nos impossível fazer uma submissão sem ser acompanhado por outras, fosse que horas fossem, impossibilitando uma avaliação concreta do código.

Exemplo

Tomaremos como exemplo o seguinte ficheiro de problemas:

casa bela 2

Depois de verificarmos a validade dos argumentos, abrimos e lemos o ficheiro de dicionário fornecido, vendo qual o tamanho da sua maior palavra, e alocando uma tabela de estruturas *words* com esse tamanho. De seguida, é aberto o ficheiro de problemas para ver quais as estruturas *words* que irão ser necessárias na criação dos grafos. Neste caso, só irão ser necessárias as estruturas correspondentes às palavras de tamanho 4. O ficheiro de problemas é fechado, lê-se mais duas vezes o dicionário, é alocada a estrutura *words* e preenchida pelas palavras do dicionário.

Fecha-se o ficheiro dicionário, e de seguida ordena-se a tabela de palavras por ordem alfabética.

De seguida, começa a criação dos grafos. Neste caso, porém, só será criado 1 grafo (para as palavras de tamanho 4), pois os grafos só serão criados se as estruturas *words* correspondentes a estes forem válidas. Este grafo será criado com 2 mutações máximas entre palavras.

Grafo criado, abre-se o ficheiro de problemas e o de saída. É alocada uma tabela para representar a *search tree* e outra para representar o acervo. É lido o problema e é corrido o algoritmo de procura para o mesmo. Após ter encontrado a solução, escreve o caminho encontrado para o ficheiro de problemas, que neste caso será:

casa 3
cala
cela
bela

Como não existem mais problemas, fecham-se ambos os ficheiros e é libertada toda a memória alocada pelo programa.

Bibliografia

Acetatos das aulas teóricas de AED;

Gamedev.net:
http://www.gamedev.net/page/resources/_/technical/artificial-intelligence/dijkstras-algorithm-shortest-path-r3872

Stanford:

http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

Mkwd.net:

https://www.mkwd.net/binary-search-algorithm-in-php/