TÉCNICO LISBOA

ALGORITMOS E ESTRUTURAS DE DADOS

RELATÓRIO DO PROJETO

WORDMORPH

Grupo 92

Nome: Daniel Filipe Norte Fortunato **Número:** 81498

Nome: José Carlos André Nobre **Número:** 84107

Nome do Docente: Carlos Bispo

ÍNDICE

Descrição do problema

Em traços gerais de modo a resumir a informação do enunciado do projeto foi-nos pedido que desenvolvêssemos um programa que fosse capaz de produzir "caminhos" entre palavras. Entende-se por um caminho entre duas palavras do mesmo tamanho, dadas como ponto de partida e de chegada, uma sequência de palavras de igual tamanho, em que cada palavra se obtém a partir da sua antecessora por substituição de um ou mais caracteres por outro(s) (mutações).

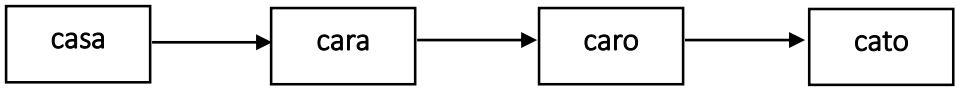
Para que haja forma de distinguir entre os vários caminhos possíveis, quando exista mais do que um, entre duas palavras dadas, é necessário introduzir um custo associado com as mutações. Neste projeto iremos assumir que o custo é quadrático no número de caracteres substituídos entre duas palavras consecutivas. Assim, para mudanças de palavras com mutações simples o custo será 1^2 por cada mutação. Para mudanças de palavras com mutações duplas o custo será 2^2 por cada uma das mutações, e assim sucessivamente (3^2, 4^2, 5^2, etc.).

O que se pretende neste projeto é desenvolver uma aplicação em linguagem C que seja capaz de calcular o caminho de menor custo entre duas palavras ou, se não houver caminho entre elas, dizer que não há caminho.

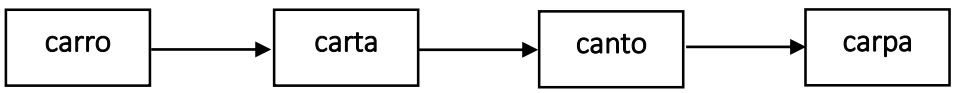
Abordagem do Problema

Começou-se por fazer um paralelismo entre o caminho entre as duas palavras e um grafo. Para implementar esta solução criámos um grafo não direcionado com as palavras relacionadas entre si, por mutações, com listas de adjacências. Por exemplo, duas palavras diferenciadas entre sim por uma só letra, significa que para chegar de uma para a outra é apenas necessário fazer uma mutação simples. Para fazer o caminho entre palavras com outras mutações (duplas, triplas, etc.) utilizámos a mesma abordagem. Assim, se as palavras estivessem relacionadas entre si, por mutações, ligávamos uma à outra na lista de adjacências e assim criávamos todos os caminhos possíveis.

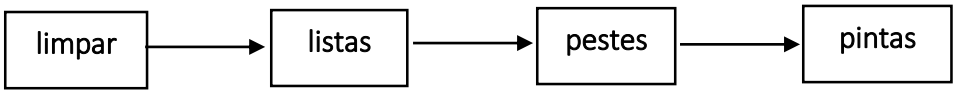
Mutações simples:



Mutações duplas:



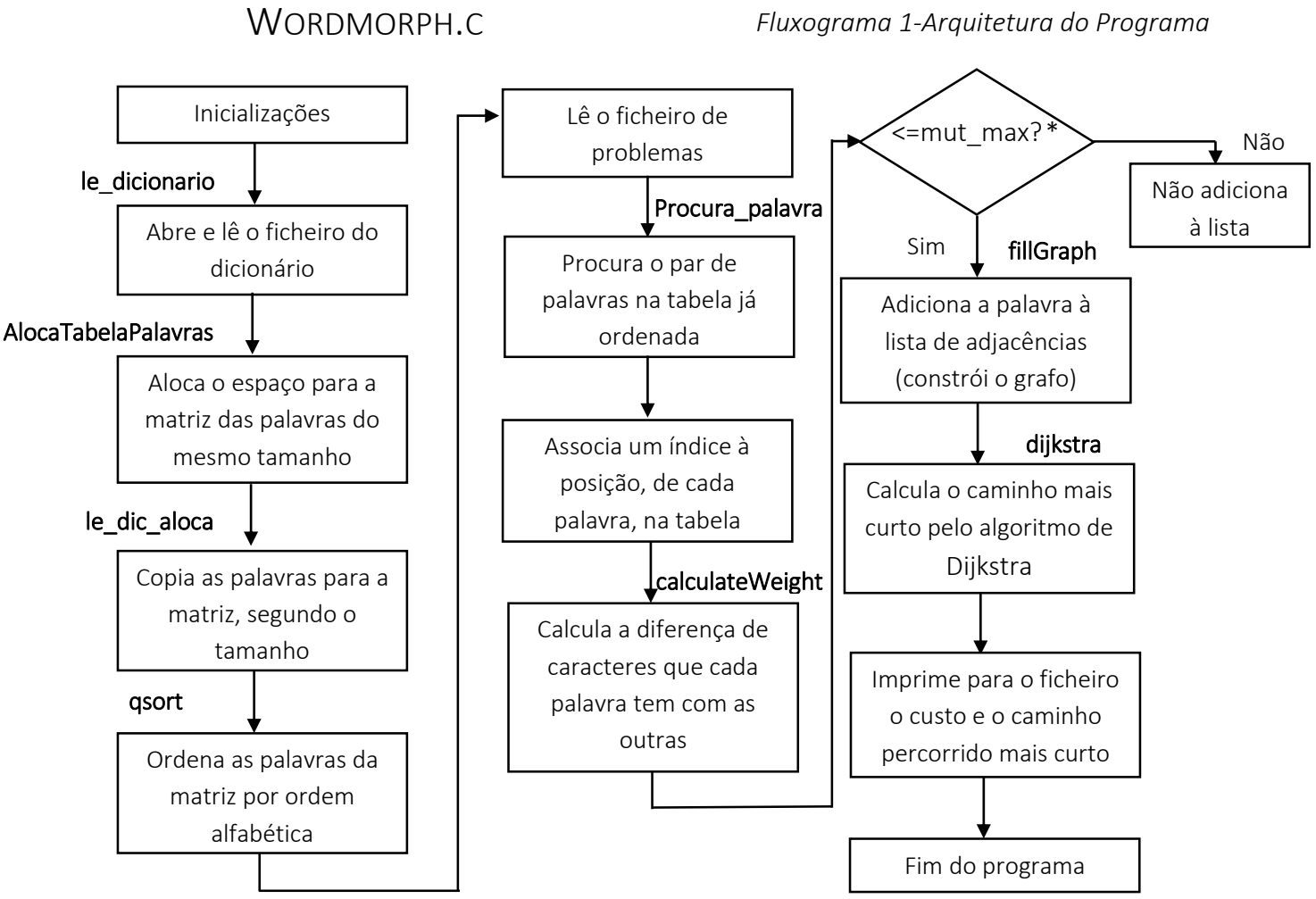
Mutações triplas:



Agora que o grafo está completo e as palavras estão todas relacionadas entre si por mutações falta calcular o caminho mais curto entre o par de palavras dado como argumento. Para isso utilizámos um algoritmo de procura dado nas aulas teóricas, o Algoritmo de Dijkstra. O que este algoritmo faz é, dado um vértice *s* de um grafo com custos positivos nas arestas, encontra uma árvore de caminhos mínimos com raiz *s* no grafo, utilizando uma fila de prioridades. Uma descrição mais detalhada deste algoritmo e de como lá chegámos encontrar-se-á mais abaixo.

Arquitetura do Programa

Fluxograma do programa Wordmorph:



Fluxograma 1-Arquitetura do Programa

*<=mut_max? – se a diferença de caracteres de uma palavra for menor ou igual ao número de mutações máximo passado como argumento no ficheiro de problemas quer dizer que essa palavra se pode transformar na outra trocando um certo número de caracteres que tem de ser menor ou igual que o numero de mutações para que essa palavra seja adicionada à lista de adjacências. Se a diferença for maior que o número de mutações máximo então essa palavra não pode ser adicionada à lista de adjacências.

O Fluxograma anterior mostra a arquitetura do programa e condensa a informação sobre as principais funções e os seus objetivos. Um estudo mais pormenorizado das estruturas de dados utilizadas e funcionamento dos algoritmos é feito mais à frente no relatório.

Descrição das estruturas de dados

Foram várias as estruturas de dados usadas neste projeto:

Usamos uma estrutura do tipo **st_dic** que tem um ponteiro para uma matriz que guardará as palavras do mesmo tamanho que mais tarde serão ordenadas, entre outros campos necessários para guardar certas especificações dos problemas e do dicionário.

Utilizamos também uma estrutura do tipo **LinkedList** que serve para guardar toda a informação relativa à lista de adjacências. Esta contém a estrutura edge e um ponteiro para o próximo nó da lista.

Utilizamos também uma estrutura do tipo **graph** que serve para guardar toda a informação relativa ao grafo como o número de nós e de arestas. Este grafo funcionará com listas de adjacências que será mostrado mais à frente o funcionamento mais detalhado.

Por fim utilizamos duas estruturas que estarão ligadas. Uma do tipo **MinHeapNode** e uma do tipo **MinHeap**. A primeira será utilizada para guardar a informação de cada bloco da fila de prioridades, como o número do vértice e a distância à origem. A segunda será utilizada para guardar toda a informação relativa à fila de prioridades, como o número de vértices, a posição de cada vértice e a própria fila.

Descrição dos Algoritmos

Os principais algoritmos utilizados na procura, ordenação das tabelas, nos cálculos de peso das arestas e no percurso do grafo são os que se encontram nas funções procura_palavra, qsort, calculeWeight, fillGraph e dijkstra.

procura_palavra

Esta função é a que procura as duas palavras passadas como argumento na matriz já ordenada das palavras.

Ideia: se os números na tabela estão ordenados podemos eliminar metade deles comparando o que procuramos com o que está na posição do meio. Se for igual temos sucesso. Se for menor aplicamos o mesmo método à primeira metade da tabela, se for maior aplicamos o mesmo método à segunda metade da tabela.

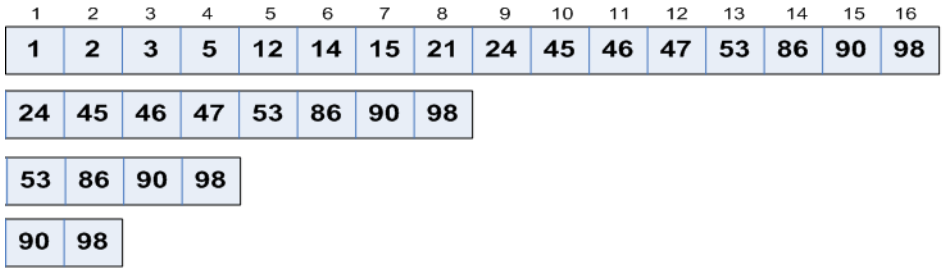


Figura 1-exemplo de execução do algoritmo de procura binária (procurar 90)

qsort

Esta função ordena as palavras do mesmo tamanho na tabela por ordem alfabética. O algoritmo Quicksort é um método de ordenação muito rápido e eficiente, inventado por C.A.R. Hoare em 1960, quando visitou a Universidade de Moscovo como estudante.

Este princípio é composto por 2 passos essenciais:

1. Escolher um elemento do vetor, o pivot (por ex., V[0]);
2. Dividir o vetor em 2 partes (esquerda e direita), em que:
    a) Parte esquerda com os elementos menores do que o pivot;
    b) Parte direita com os elementos maiores do que o pivot.

Ao ser aplicado sucessivamente este princípio a ambas as partes, quando o processo recursivo terminar o vetor está ordenado.

O algoritmo é composto então por 3 passos:

1. Determinar a posição final k do elemento pivot (V[0]) no vetor e colocá-lo nessa posição, dividindo o vetor em 2 partes:
    a) Esquerda = V[0], ..., V[k-1]
    b) Direita = V[k+1], ..., V[tam-1]
2. Aplicar o algoritmo recursivamente à parte esquerda;
3. Aplicar o algoritmo recursivamente à parte direita.

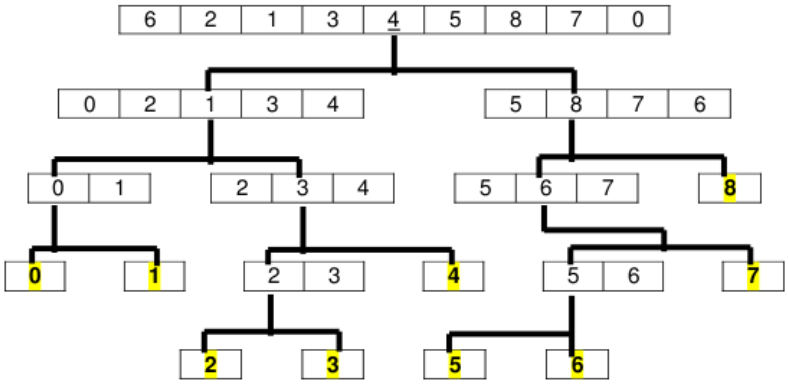
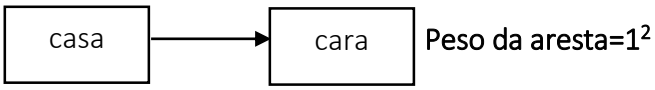


Figura 2-Exemplo de execução do algoritmo Quicksort

calculeWeight

Esta função calcula o peso de cada aresta que liga cada nó do grafo. Esse peso é calculado da seguinte maneira: as duas palavras serão comparadas uma com a outra para saber qual será o peso da aresta que ligará uma à outra. A cada diferença de caracter é incrementada uma variável que no fim será elevada ao quadrado para calcular o peso real, uma vez que cada mutação simples custa 1^2, cada mutação dupla custa 2^2, cada mutação tripla custa 3^2, etc. Será através desta função que, mais tarde, vai ser calculado o custo de cada caminho.

Mutação simples:

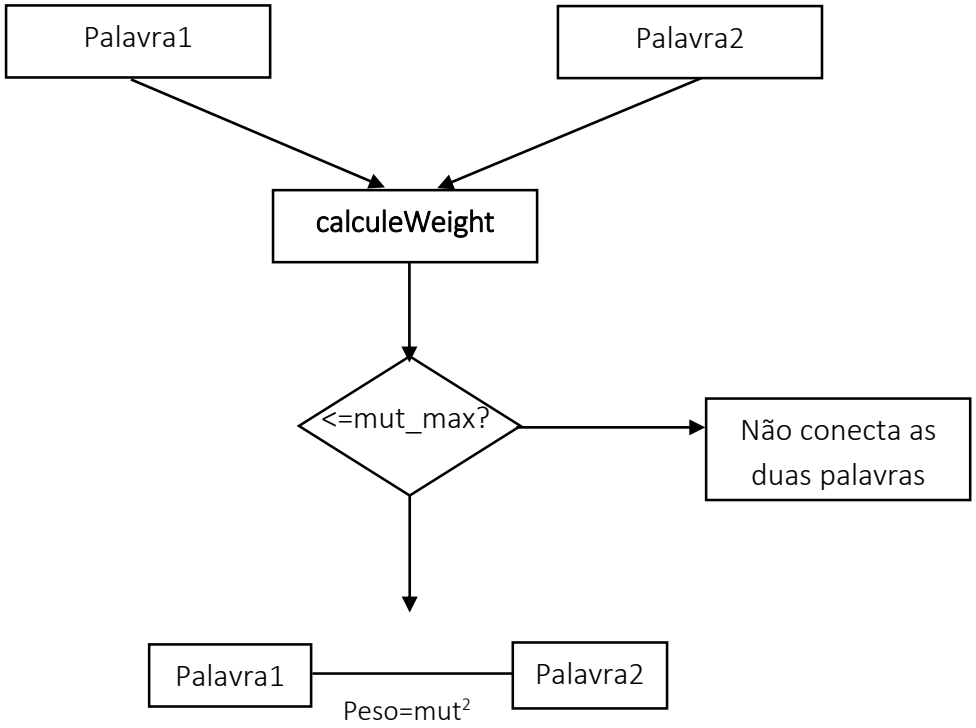


Mutação dupla:

Mutação tripla:

fillGraph

Esta função é a que constrói o grafo já com todas as palavras ligadas entre si com os respetivos pesos das arestas. Para isso tivemos de comparar cada palavra da matriz ordenada com as outras todas. No ficheiro de problemas será passado um número inteiro que será o número máximo de mutações possíveis para aquele problema. Logo no fillGraph será feito a comparação de todas as palavras da matriz ordenada com as outras todas utilizando a função calculateWeight. Se esse valor for menor que o número máximo de mutações a função irá conectar essas duas palavras na lista de adjacências (uma vez que se pode concluir que pode transformar-se uma palavra na outra através de uma mutação menor que a mutação máxima) e a aresta ficará com o respetivo peso, dependendo da mutação feita (simples, dupla, tripla, etc.).



Fluxograma 2-Algoritmo de cálculo do peso das arestas

dijkstra

No algoritmo de Dijkstra, dois conjuntos são mantidos, um conjunto contém a lista de vértices já incluídos no SPT (Shortest Path Tree), o outro conjunto contém os vértices

ainda não incluídos. Com a representação de lista de adjacência, todos os vértices de um grafo podem ser percorridos em tempo O (V + E) usando BFS. A ideia é percorrer todos os vértices do grafo usando BFS e usar um Min Heap para armazenar os vértices ainda não incluídos no SPT (ou os vértices para os quais a distância mais curta ainda não está finalizada). Min Heap é usado como uma fila de prioridades para obter o vértice de distância mínima do conjunto de vértices ainda não incluídos.

Seguir-se-á a explicação mais detalhada e sequencial do algoritmo de Dijkstra:

1. Criar uma Heap de tamanho V onde V é o número de vértices do grafo dado. Cada nó da Heap contém o número do vértice e o valor de distância do vértice.
2. Inicializar a Heap com o vértice de origem (o valor de distância atribuído ao vértice de origem é 0). O valor da distância atribuído a todos os outros vértices é infinito).

3. Enquanto a Heap não está vazia, fazer o seguinte:
    a) Extrair o vértice com o menor valor de distância da Heap. O vértice extraído ser designado por **u**.
    b) Para cada vértice adjacente **v** de **u**, verificar se **v** está na Heap. Se **v** está na Heap e o valor da distância é maior do que o peso da aresta **u**-**v** mais o valor da distância de **u**, então deve-se atualizar o valor da distância de **v**.

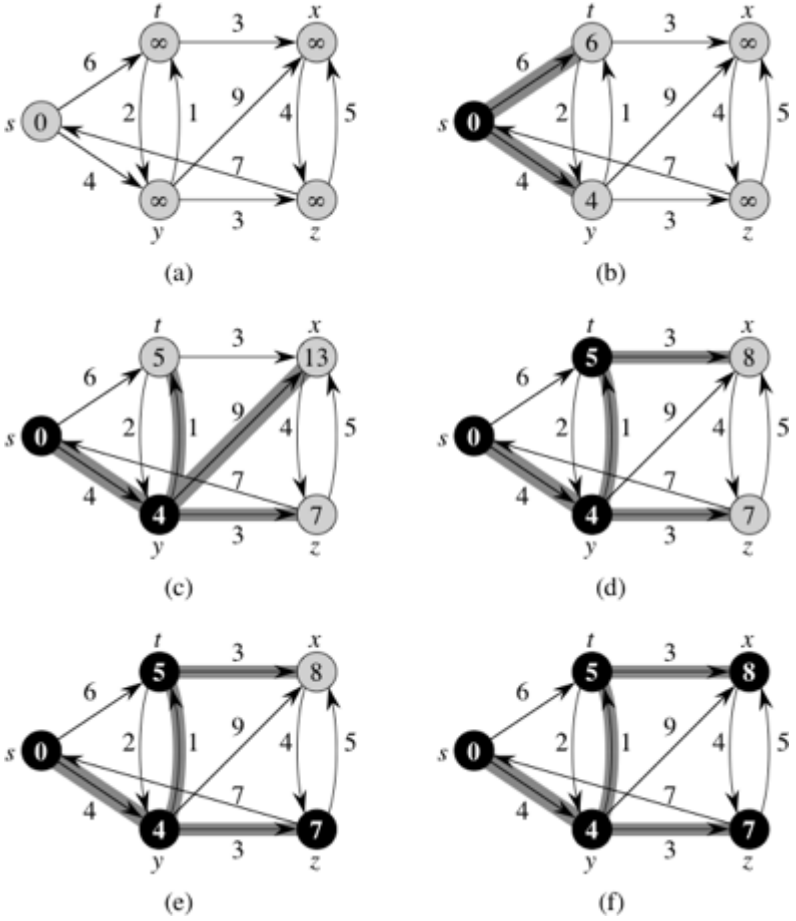


Figura 2-Exemplo de execução do algoritmo de Dijkstra

Descrição dos subsistemas

O programa por nós implementado tem vários subsistemas seguidamente enumerados:

Graph – constituído por graph.c e graph.h onde estão guardadas as funções que estão diretamente ligadas à criação e manipulação de dados nas estruturas de dados. As estruturas de dados estão definidas no ficheiro graph.h.

File – constituído por file.c e file.h que contém as funções diretamente relacionadas com a leitura ou manipulação de dados em ficheiros de texto.

Table – constituído por table.c e table.h onde estão guardadas as funções que estão diretamente ligadas à alocação da matriz das palavras e de procura de palavras na matriz ordenada.

Heap – constituído por heap.c e heap.h onde estão guardadas as funções diretamente ligadas à criação e manipulação de dados na fila de prioridades.

Defs.h que contém a especificação da estrutura edge.

Análise dos Requisitos Computacionais

As várias escolhas para estruturas de dados do problema foram tidas em conta face à maneira como iriam ser acedidas de modo a minimizar a complexidade.

Grafo: na estrutura onde são armazenadas as informações do grafo são criadas, uma lista de adjacência, um inteiro com o número de Vértices e um inteiro com o número de arestas.

Lista de adjacências: na lista de adjacências é alocada memória para cada ligação. Durante as várias etapas de processamento é apenas necessário aceder ao primeiro elemento da lista e seguidamente aos próximos. Ou seja nunca é preciso aceder a um elemento específico pelo que a complexidade quando se quer aceder a um dos termos desta lista, seja o primeiro ou o próximo será O(1). Como esta lista guarda ligações de cada nó a memória alocada é proporcional ao número de nós e ao número de arestas, mas como por cada nó é apenas alocado um ponteiro, a influência do número de arestas na memória alocada é maior.

Fila de Prioridades: a implementação das filas de prioridade por acervos (heaps) permite o melhor equilíbrio no desempenho da procura (e remoção) de elementos com maior e com prioridade qualquer. Vantagens da representação de árvores por tabela:

a) Menor ocupação de memória;
b) Acesso constante a qualquer elemento.

	Inserção	Remover chave máxima	Remover	Encontrar chave máxima	Modificar prioridade	Junção
Heap	lg N	lg N	lg N	1	lg N	N

Tabela 1-complexidade da utilização da fila de prioridades

Principais algoritmos:

Procura_palavra() - A procura binária nunca examina mais do que lg N + 1 números. Mostra que este tipo de pesquisa permite resolver problemas até um milhão de dados com cerca de 20 comparações por procura

Qsort() - usa cerca de N2 /2 comparações no pior caso. Se o ficheiro já estiver ordenado, todas as partições degeneram e o programa chama-se a si próprio N vezes; o número de comparações é de N + (N-1) + (N-2) + … + 2 + 1 = (N + 1)N/2 (mesma situação se o ficheiro estiver ordenado por ordem inversa). Não apenas o tempo necessário para a execução do algoritmo cresce quadraticamente como o espaço necessário para o processo recursivo é de cerca de N o que é inaceitável para ficheiros grandes. Usa cerca de 2NlgN comparações em média.

Dijkstra() - as instruções no loop interno são executadas O (V + E) vezes (semelhante ao BFS). O loop interno tem a operação decreaseKey () que é O(LogV). Portanto, a complexidade do tempo total é O(E + V) * O(LogV) que é O((E + V)* LogV) = O(ELogV).

Funcionamento do programa

Apesar de termos dado o nosso melhor para fazer a implementação do nosso programa de acordo com a menor complexidade e memória os resultados não foram perfeitos, uma vez que o programa não passou em todos os testes solicitados pelo professor.

Exemplo:

Imaginemos um dicionário com as palavras carpa, carro, carta, lerpa, limpa, lista. Admitindo mutações triplas, e querendo o caminho mais curto entre carpa e lista.

A lista de adjacências mostra:

[0]->(1|2^2)->(2|1^2)->(3|2^2)->(4|3^2)

[1]-> (0|2^2)->(2|2^2)

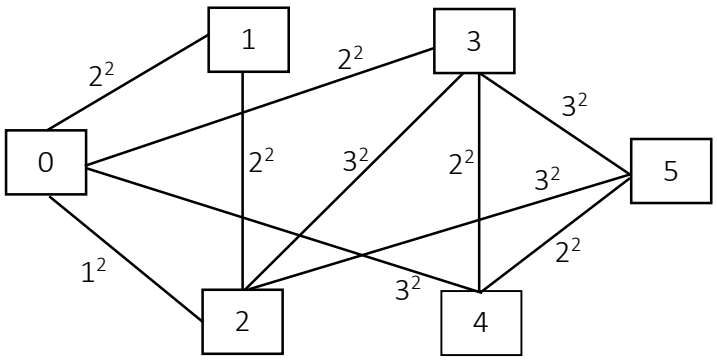
[2]-> (0|1^2)->(1|2^2)->(3|3^2)->(5|3^2)

[3]-> (0|2^2)->(2|3^2)->(4|2^2)->(5|3^2)

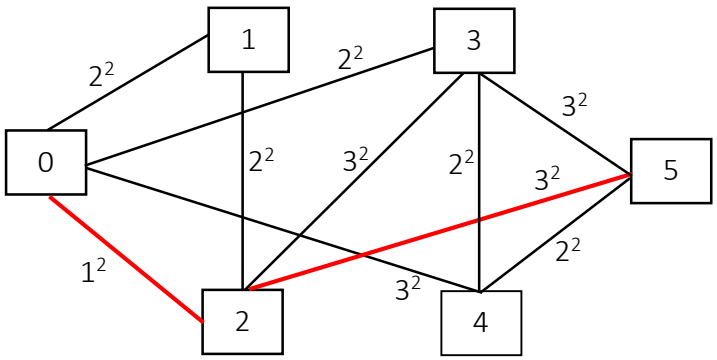
[4]-> (0|3^2)->(3|2^2)->(5|2^2)

[5]-> (2|3^2)->(3|3^2)->(4|2^2)

Construindo o grafo:



Fazendo o cálculo do caminho mais curto:



O caminho mais curto será 0->2->5 que tem um custo de $1^2 + 3^2 = 10$ que é o menor custo possível para ir de carpa para lista.

O ficheiro .path terá impresso:

Carpa 10
Carta
Lista

Para outros dicionários a forma de execução será a mesma, mudando consoante o número de mutações do problema e as palavras do dicionário.

Conclusão

Com este projeto concluímos que todos os assuntos abordados nas aulas e nos laboratórios são importantes para que haja uma diminuição da complexidade num programa como este, em que se está em constante procura e inserção de variáveis. Concluímos também que para que haja uma diminuição significativa na memória e no tempo de execução do programa temos de adaptar as nossas estruturas de dados. São estes os aspetos importantes para que o programa esteja no seu melhor desempenho.

Os laboratórios foram uma componente muito importante, uma vez foram abordados assuntos muito importantes para o projeto, como as listas de adjacências e as filas de prioridades, sendo mesmo utilizado código fornecido pelo professor para a realização deste projeto.

Por fim podemos também concluir que estes assuntos foram, não só importantes para a realização deste projeto de procura de menor caminho entre palavras através de mutações, mas também foram importantes para nos darem uma vista mais ampla sobre o que se passa à nossa volta e como podemos resolver certos assuntos no nosso dia a dia como, por exemplo, na procura de menor caminho para ir a um determinado sítio.

Bibliografia

- Acetatos do Professor
- Código fornecido nos laboratórios
- Relatório Tipo fornecido pelo professor
- http://www.geeksforgeeks.org/greedy-algorithms-set-7-dijkstras-algorithm-for-adjacency-list-representation/
- http://comp.ist.utl.pt/ec-aed/PDFs/Heaps.pdf
- http://ninf.org/w/images/f/f0/Programa%C3%A7%C3%A3o_II_2015_Aula_Te%C3%B3rica_5.2.pdf
- https://www.google.pt/search?q=algoritmo+quicksort&espv=2&biw=1366&bih=638&source=lnms&tbm=isch&sa=X&sqi=2&ved=0ahUKEwiii8eeivLQAhXBoRQKHQ9PB5wQ_AUIBigB&dpr=1#imgrc=493UUzZefgR-1M%3A
- https://www.google.pt/search?q=procura+binaria&espv=2&biw=1366&bih=638&source=lnms&tbm=isch&sa=X&ved=0ahUKEwi9777XtPLQAhXMtRQKHROoCUYQ_AUIBigB&dpr=1#imgrc=W0y8NAoYNWPZAM%3A