TÉCNICO LISBOA

Algoritmos e Estruturas de Dados

Relatório do projeto final

Thomas Berry - 84189
André Cavalheiro – 84000

Grupo - 003

Índice

Descrição do Problema

Neste projeto pretende-se desenvolver um programa que seja capaz de produzir "caminhos" entre palavras. Entende-se por um caminho entre 2 palavras do mesmo tamanho, dadas como ponto de partida e de chegada, uma sequência de palavras de igual tamanho, em que cada palavra se obtem a partir da sua antecessora por substituicao de um ou mais caracteres por outro(s).

Por exemplo, seja a seguinte sequência de palavras, que estabelece um caminho de palavras entre a palavracarro e a palavra pista:

carro

corro

porto

porta

posta

pista

Como para este problema existe mais que 1 caminho entre cada palavra: pretende-se econtrar o caminho com o custo menor, ou seja, aquele cujo somatório do quadrado do numero de caracteres diferentes entre cada palavra consecutiva é menor.

Ainda assim, é possivel encontrar mais que 1 caminho com o mesmo custo associado. No entanto, só é necessário encontrar um que tenha o custo mais baixo.

Abordagem ao Problema

Este trabalho foi dividido em 2 submissões sendo que a primeira teve como objetivo criar funções necessárias para a execução da segunda.

Na primeira parte criámos as funções para carregar os exercícios e dicionários, criámos também funções que localizam cada palavra dos exercícios (inicial e final) no dicionário usando algoritmos descritos mais à frente no relatório.

Completadas estas funções para a primeira parte, procedemos com a execução do algorítmo que descobre o caminho mais curto possível entre dois nós de um grafo, o algoritmo Dijkstra. Para a utilização deste algoritmo é preciso a ciração prévia de vários componentes:

-Em primeiro lugar é necessária a criação da representação do grafo como lista de adjacências.

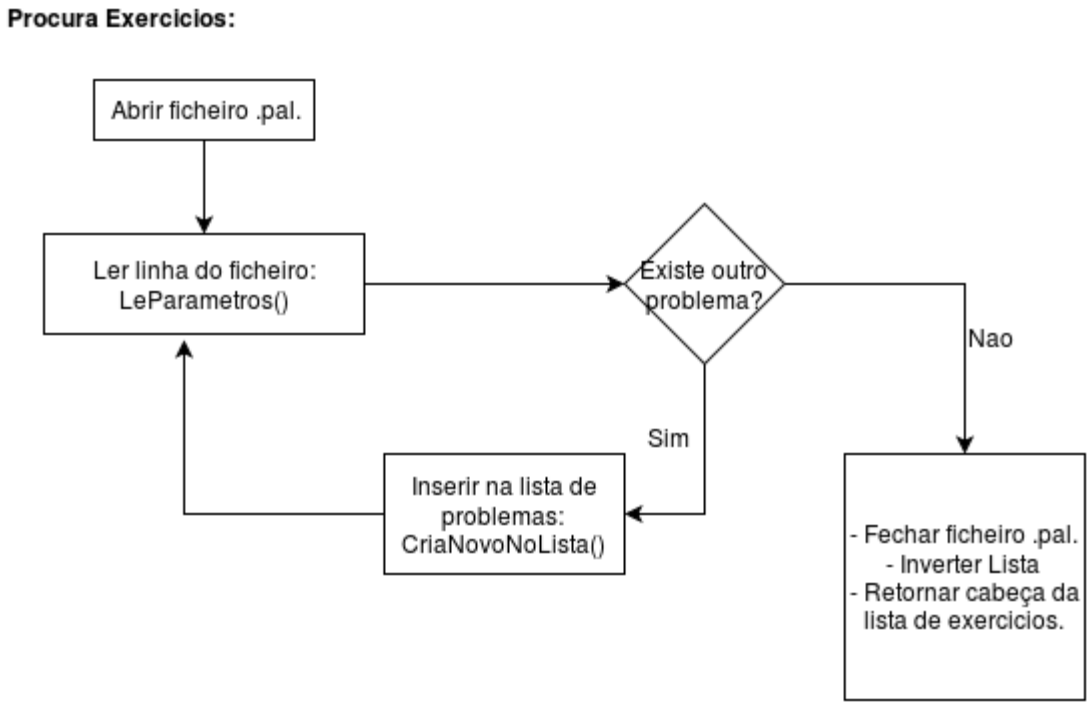
-A criação de funções de criação e manipulação de uma lista prioritária em formato de acervo.

Após todos os componentes estarem devidamente inicializados, bastou correr o algoritmo para imprimir os resultados no ficheiro de saída.

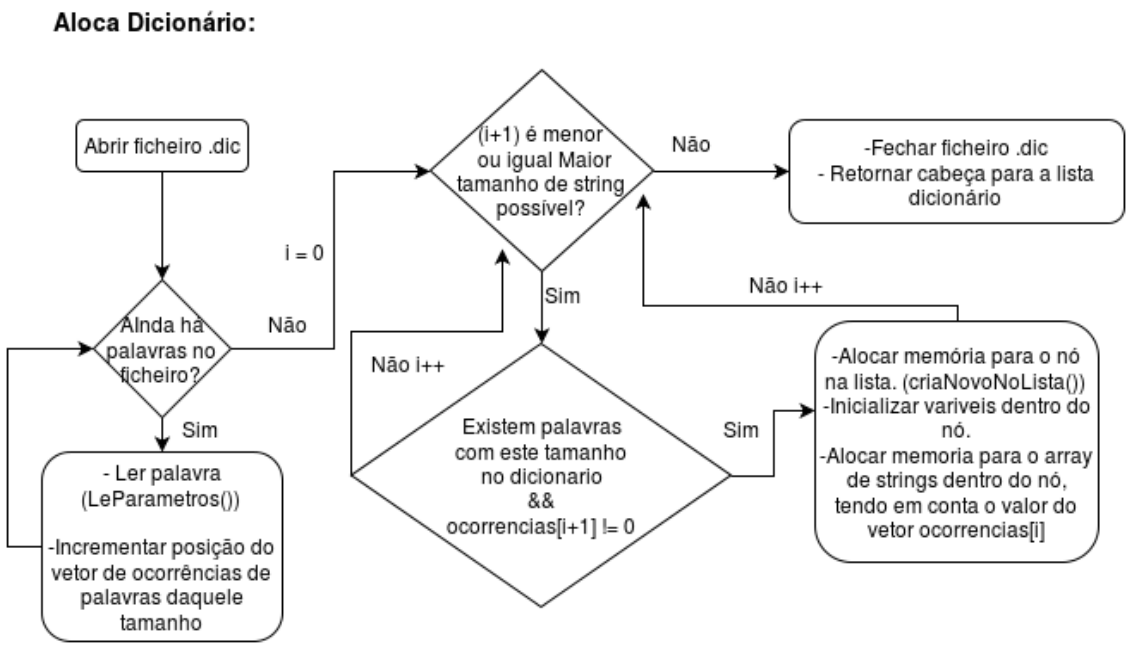
A escolha dos algoritmos e funções a utilizar foram todas feitas tendo por baso o prencípio de menor complexidade possível tendo sempre em conta a quantidade de memória utilizada. Por exemplo embora o uso de tabela de adjacências nos parecesse aliciante ao início pelo tempo de acesso constante aos vertices adjacentes, a quantidade memoria alocada tornava impossível tal implementação. (O espaço alocado seria $n^2 * 4$ bytes (tamanho de variaveis int)).

Arquitetura geral do programa

O programa começa por ler o ficheiro de exercícios, para saber quais as palavras do dicionário é que vale a pena carregar para a memória, ou seja, se não há palavras com 3 caracteres no ficheiro .pal, não há necessidade de carregar palavras com tamanho 3 do dicionário par a memória.

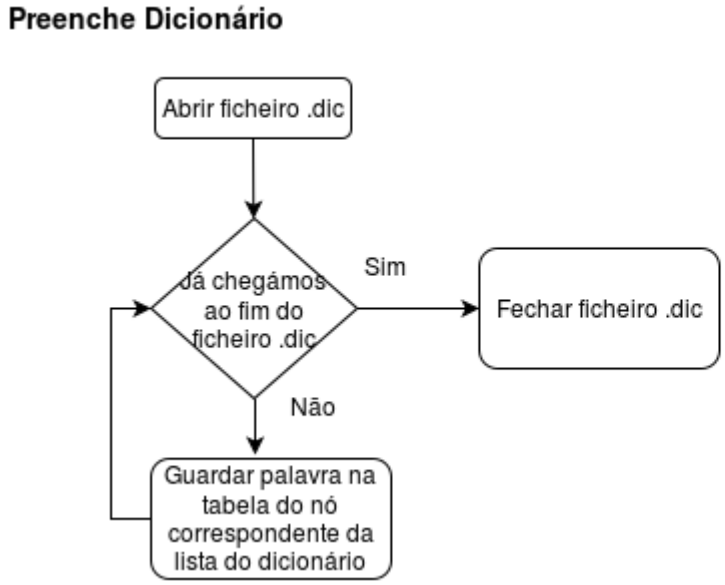


1 - Fluxograma da função Procura exercicios



2 - Fluxograma da função Aloca DIcionário

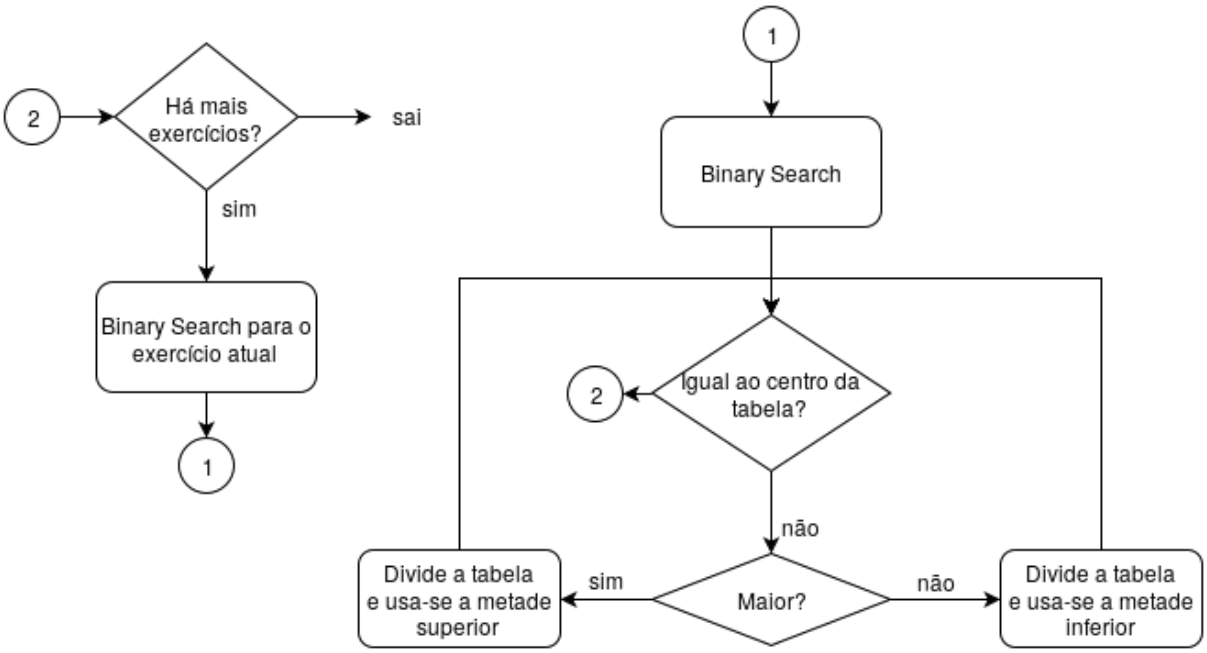
Após carregar os exercícios carregamos o dicionário para a memória.



3 - Fluxograma da função Preenche dicionário

Depois aplicamos o algorítmo QuickSort para organizar as palavras do dicionário.

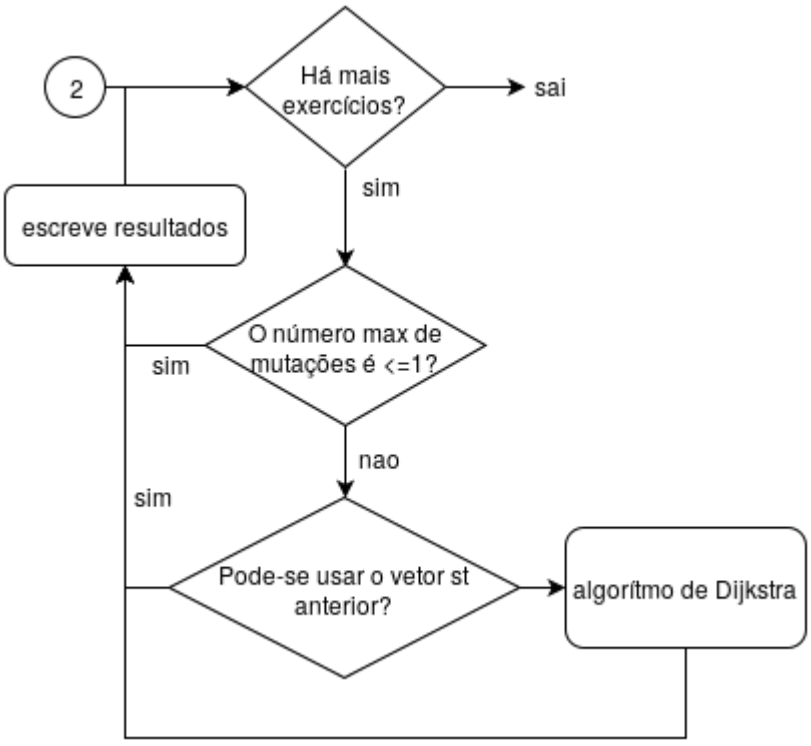
Uma vez organizado, encontramos a posição das palavras dos exercícios no dicionário aplicando o BinarySearch e libertamos as strings palavra_inicial e palavra_final para a memória pois já não são necessárias.



4 - Fluxograma da Função que encontra a posição das palavras

Agora procedemos com a criação dos grafos para todas as palavras do dicionário.

Uma vez criados os grafos aplicamos o algorítmo de Dijkstra para cada um dos exercícios de modo a encontrar o caminho.



5 - Fluxograma da Função que encontra caminhos

Uma vez encontrados todos os caminhos, os resultados são escritos para um ficheiro de saída com a extensão .path, os exercícios e o dicionário é libertado para a memória, e o programa termina.

Descrição das estruturas de dados usasadas

Payload_Dicionário:

Estra estrutura é um nó duma lista simplesmente ligada que contém:

- o número de caracteres que as palavras da tabela deste nó contém.
- o número de palavras na tabela
- a tabela de palavras

Payload_Exercícios:

Estra estrutura é um nó duma lista simplesmente ligada que contém:

- numero de carateres das palavras
- numero maximo de mutações
- palavra inicial
- palavra final
- posição na tabela da palavra inicial
- posição na tabela da palavra final

FilaP:

Esta estrututa representa uma fila prioritáira implementada por acervo que contém:

- um vetor que representa a fila implementada por tabela
- um vetor que armazena a posição dos nós do grafo na tabela
- um inteiro que representa o númro de itens na tabela
- um inteiro que representa o tamanho máximo do acervo

Lista_adjs:

Esta estrutura representa um nó no grafo, implemento por lista de adjêncidas, que contém:

- um ponteiro para o proximo nó adjecente

- o peso da ligação

Descrição dos algoritmos implementados

Binary search

Após a ordenação do dicionário, é necessário encontrar a posição tanto da palavra inicial como final de movo a facilitar o acesso a estas na

lista de adjacências e nos vetores wtf e st. Para isso, decidimos aplicar o algoritmo de Binary search com a palavra inicial e final usando o array pertentente ao nó cuja posição na lista é igual ao número de caracteres das palavras inicial e final.

O algoritmo consiste no seguinte: Comparar a palavra em questão com a palavra no meio da tabela. É de notar que a tabela se encontra ordena, por isso após a comparação conseguimos imediantamente perceber se a palavra que queremos encontrar se encontra numa metade do array ou na outra. Podemos agora comparar de novo com a palavra cujo index se encontra a metade da tabela restante e voltar a dividir a tabela para metade. Assim consecutivamene até nos restar apenas uma posição que é a correspondente à posição da palavra que queríamos encontrar.

Como é fácil de perceber a complexidade do algotimo é O(log(n)), onde n é o número de elementos do dicionario com o tamanho das palavras inicial e final, uma vez que elemina metade de todas as respostas possiveis cada vez que que um loop do algoritmo corre.

QuickSort

A nossa implementação do algoritmo quicksort funciona de maneira recursiva e tem como objectivo a ordenação dos diversos vetores de strings que estão na lista dicionário. Para garantir o bom funcionamento do algorítmo começámos por:

Escolher 1 elemento como pivô, que é o elemento mais à direita da tabela, assim, todos os elementos que vêm alfabeticamente a seguir a esse elemento vão para a direita dele, e os restantes para a sua esquerda. Este é o objetivo da função partition. Após esta função ser executada voltamos a correr a função quick sort para cada uma das 2 subtabelas obtidas, uma à esquerda e outra à direita do indice i.

Quando, aplicamos a função quicksort a uma tabela ordenada ou uma tabela só com 1 elemento a função retorna.

Este algoritmo não possui uma complexidade fixa dependendo dos valores escolhidos como pivôs. No pior caso o algoritmo terá complexidade $O(n^2)$. Mas como o ficheiro do dicionário está completamente aleatório, o algorítmo vai ter uma execução muito melhor que quadrática.

Dijkstra

Sendo o Algoritmo Dijkstra o algoritmo mais efeciente para encontrar o caminho mais curto possivel entre dois vertices de um grafo era a nossa melhor hipotese para resolver os Problemas popostos:

Funcionamento:

Existem 4 ferramentas que guardam informação que o algoitmo Dijkstra vai utilizar para funcionar eficientemente: (1)A lista de adjacências que contêm todos os vertices do grafo bem como as ligações entre as várias palavras. (2) O vetor representante da Fila prioritária vai guardar por ordem de prioridade os vértices do grafo, sendo que essa ordem de prioridade é ditada pelos valores (3)do vetor que representa o peso das ligações entre as palvras (wt). Por fim os resultados obtidos pelo algoritmo serão depositados no (4) vetor que vai representar as ligações da árvore de caminhos mais curtos do grafo (st).

É de notar que o valor do index das palavras na sua zona do dicionário é o mesmo index que se vai usar para aceder a informção sobre essa palavra na lista de adjacências, no vetor wt e st. Daí a importância em organizar as tabelas de dicionário e guardar a posição das palavras finais e iniciais.

Considerando V o número de vértices do grafo e L o de ligações:

-Cada vertice no grafo pode estar ligado a V palavras.

- Para encontrar o peso de cada vertice adjacente demora tempo constante uma vez que temos guardado o indice dele no vetor de pesos. Para reajustar o acervo o custo é O(log(N)), sendo que, conformo o algoritmo vai correndo este N torna-se cada vez mais pequeno.

Assim sendo, na pior das situações, em que cada vertice está ligado a todos os outros o tempo de execução do algoritmo é menor que O (L +Vlog(V)).

Descrição dos Subsistemas criados

Criação e manipulação do acervo:

fila_prioritaria.c e fila_prioritaria.c

Funções de iniciação de acervo:

FilaP *FpriorIni(int);

Funções de inserção e remoção de elementos do acervo:

```
void   FInsere(FilaP * fp, int item, unsigned short weight[]);
void   FInsereDirec(FilaP * fp, int item)
int  FRemove(FilaP * fp, unsigned short weight[]);
```

Função de verificação do tamanho da heap:

```
void Fpisfree(FilaP * fp);
```

Funções de acerto do acervo:

```
 void  FixUp(FilaP * , int idx, unsigned short weight[] );
 void  FixDown(FilaP *, int idx, int n, unsigned short weight[] );
```

Função de libertação de memória:

```
void FPfree(FilaP *  fp);
```

implementação de algoritmos de ordenação

ordenador.c e ordenador.h

Funções de ordenação do dicionario:

```
void ArrumaDicionario(t_lista * dicionario);
void encontraposicao( t_lista * exercicios, t_lista * dicionariocabeca );
```

Função de comparação de duas strings

```
int  stringcompare ( char* a, char* b );
```

Algoritmos:

```
void mergesort(Type * table, int low, int high, int comparer(Type, Type) );
void merge(Type * table, int low, int mid, int high, int comparer(Type, Type) );
int  binarysearch(Type* table, int l, int r, Type item, int comparer(Type, Type) );
int  compfunc( const void * a, const void * b );
void quicksort(Item a[], int l, int r);
int  partition(Item a[], int l, int r);
```

criação e manipulação de listas

list.c e list.h

Funções de iniciação da lista:

```
t_lista  *iniLista (void);
```

Funções de acesso lista:

```
Item      getItemLista (t_lista *p);
t_lista  *getProxElementoLista(t_lista *p);
int       numItensNaLista (t_lista *lp);
```

Função de manipulação:

```
t_lista  *criaNovoNoLista (t_lista *lp, Item this, int *err);
t_lista * InverteLista( t_lista * l );
```

Função de libertação de memória:

```
void      libertaLista(t_lista *lp, void freeItem(Item));
```

Subsistemas próprios do programa:

Leitura e escrita de ficheiros de texto

file.c declarado em prototipos.h

Função de abertura de ficheiros

```
FILE   *AbreFicheiro ( char *nome, char *mode );
```

Funções de leitura de ficheiros

```
char   *LePalavra ( FILE * f );
int     LeParametros(FILE * f, char ** inicial, char ** final, int *quant);
```

Funções relacionadas com o guardo de informação

```
t_lista*AlocaDicionario (FILE * f,short numchars[MAX_STR]);
void    PreencheDicionario(FILE * f, t_lista * dicionario, short numchars[MAX_STR]);
t_lista*ProcuraExercicios(FILE* ficheiro, short numchars[MAX_STR],short *nmutmax);
```

Funções relacionadas com o ficheiro de saída

```
char   *ConstroiNome ( char*  nomeantigo );
void    EscreveFicheiro(char* nome, t_lista * exercicios, t_lista * dicionario);
```

Criação e manipulação do Grafo

garfo.c declarado em protipos.h

Funções de criação de grafos

void cria_grafo(payload_dicionario * payld, short nmutmax);

void cria_todos_grafos(t_lista * dicionario,short nmutmax[]);

Função que compara duas palavras

int comparer2(char * p1, char* p2, short n);

Funções relacionadas com encontrar o caminho entre duas palavras

void encontracaminhos(t_lista * dicionario, t_lista * exercicios, char* nomeficheiro);

int dijkstra(int ini, int fini, lista_adjs** lista , int num_v, short max_mut, unsigned short ** _st, unsigned short ** _wt);

void printcaminho(FILE*fp, unsigned short* st, unsigned short n, char** palavras);

Libertação de memoria alocada para as listas que guardam as palavras do dicionario dos problemas

libertador.c declarado em prototipos.h

Função libertadora de memória alocada para a lista dicionário e de problemas

void Freedoom (t_lista* dicionario, t_lista * caminhos);

Libertação do conteúdo dos nós de listas

void freepldicionario(Item aaa);

void free_exercicios(Item aaa);

Função alocadora de memória e verificadora de erro

void * x_malloc(size_t size);

Raiz

Declaração de estruturas, constantes e inclusão de bibliotecas para o subsistema raiz (estrt_const_bibliot.h)

Prototipagem da maioria das funções do programa (prototipos.h)

Execução do código raiz (main.c)

Requesitios Computacionais do Programa

Ao longo da execução do projeto tivemos de fazer escolhas quanto aos algorítmos e estruturas a utilizar, sendo que a escolha foi sempre feita de modo a reduzir a complexidade temporal e o uso de memória.

Lista de exercícios

Para armazenar os exercicios obtou-se pela utilização, cujo consumo de memória é diretamente proporcional ao numero de exercícios. Apesar das listas serem se complexidade linear para o acesso, tal problema não se verifica para este trabalho pois queremos percorrer os exercícios pela mesma ordem que são lidos. Logo, o acesso é, essencialmente, **O(1)**.

Lista de Tabelas para o dicionário

Dado o elevado número de palavras do dicionário, criou-se tabelas de palavras por cada “subdicionário” ,ou seja, 1 subdicionário por número de caracteres na palavra. Cada subdiocionário foi colocado numa lista simplesmente ligada pois, como há poucos dicionários, não mais que quarenta, a velocidade para encontrar o dicionário certo para resolver um exercício é curta. As tabelas de palavras têm acesso constante, indispensável para aplicar o algorítmo binarysearch, que faz muitos acessos à tabela.

Lista de adjacências para o grafo

Dado que o grafo é esparso, pois há poucas ligações entre palavras para um pequeno número de mutações máximo, concluiu-se que é ideal a utilização de uma lista de adjacências para o grafo, cujo consumo de memória é proporcional a V (número de palavras) + E (número de ligações), que, neste trabalho, é muito inferior a V^2. Apesar do aceso ao grafo ser proporcional a V, a alternativa, a utilização de uma matriz adjacências consome demaisada memória.

Exemplo de aplicação

A seguir temos o caminho produzido pelo programa para a palavra inicial farmácia e para a palavra final remédios com um número máximo de mutações 4:

farmacia 21

farmacos

fariamos

feriamos

geriamos

gemiamos

remiamos

remedios

Como é evidente, é provável haver mais que 1 caminho entre duas palavras a seguir vemos o caminho produzido pelo programa do professor:

farmacia 21

farmacos

fariamos

feriamos

teriamos

temiamos

remiamos

remedios

Apesar dos caminhos, serem diferentes, ambos apresentam o mesmo custo, 21 pontos, logo, ambos são válidos.