Instituto Superior Técnico

Mestrado em Eng. Electrotécnica e de Computadores

Algoritmos e Estruturas de Dados

2016/2017 – 2º Ano, 1º Semestre

Relatório do Projecto

Wordmorph

Grupo nº: 110

Carlos Alexandre Marques Alves da Silva
nº81323 e-mail: carlosmarques.personal@gmail.com

Alexandre Thane Coutinho
nº81574 e-mail: alexandretcoutinho@gmail.com

Docente: **Carlos Bispo**

Indice

Descrição do problema

Resumindo a informação presente no enunciado do projecto o que se pretende desenvolver é um programa que recebe um ficheiro de problemas (sendo cada problema formado por duas palavras e um numero maximo de caracteres que podem ser mudados numa só mutação da palavra) e um dicionario de palavras e produz um ficheiro de soluções que contém todas as soluções dos problemas (sendo cada solução o peso mínimo de mutação da palavra 1 na palavra 2 e uma lista de palavras desde a palavra 1 até a palavra 2 das mutações que a palavra 1 teve que fazer para chegar a dois transformando-se sempre numa palavra presente no dicionário e nunca ultrapassando o maximo de mudança de caracteres por palavra).

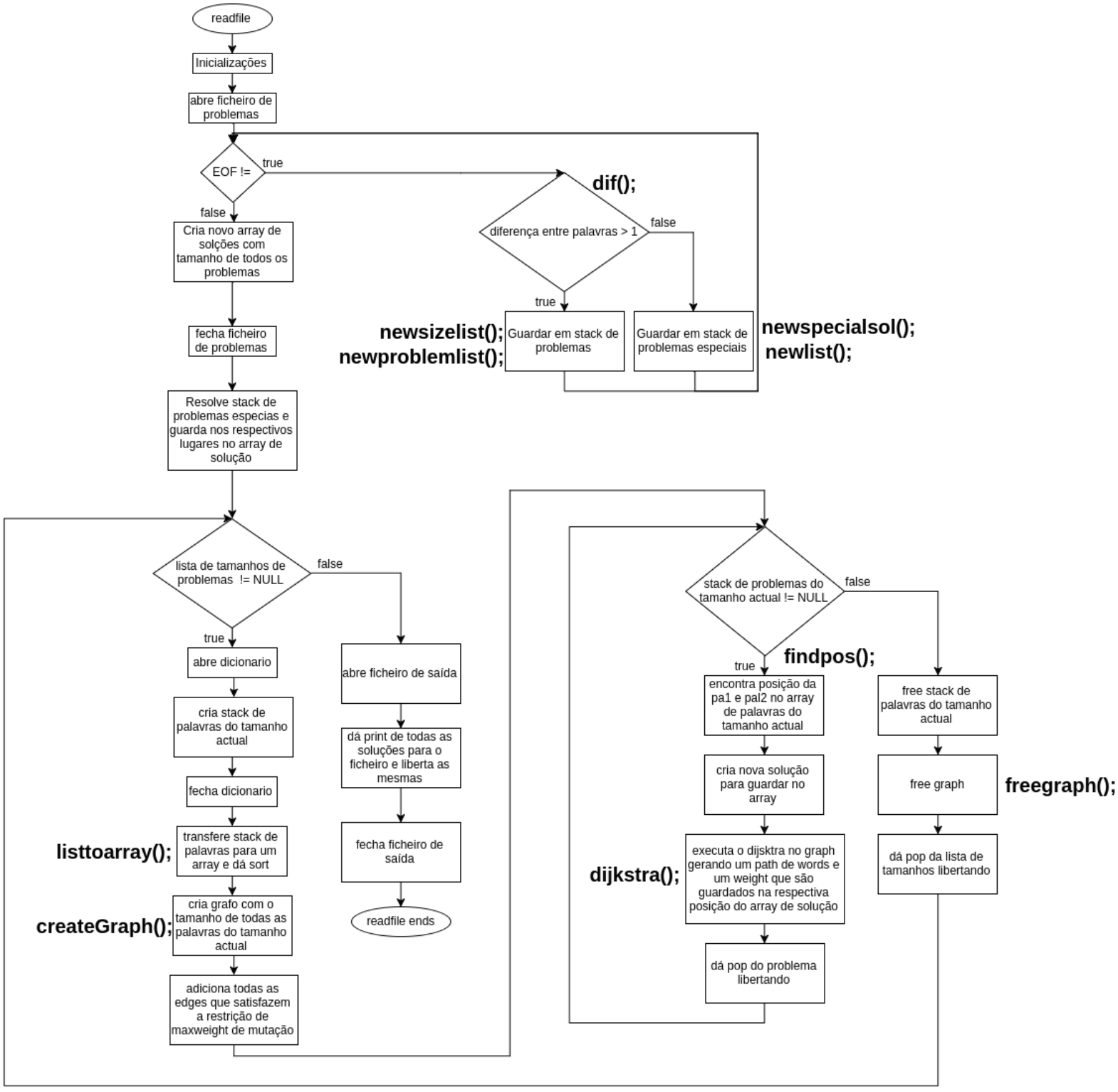
Abordagem ao problema

Após se ter chegado à conclusão que cada palavra poderia ser considerada um vertice de um grafo e cada edge teria o peso da transformação entre as palavras dos vertices que conectava percebeu-se que o melhor algoritmo para resolver este problema determinando o caminho mais curto entre duas palavras seria o Dijkstra.
Primariamente tentou se alocar um grafo com uma matriz de adjacencias guardadando todos os pesos entre todos os vertices e chegou-se a conclusão que usava demasiada memória desnecessária. Então tomou-se outra solução, criar uma lista de tamanhos de palavras e guardar em cada tamanho todos os problemas desse tamanho de seguida criar-se um array de soluções baseado no número de problemas extraídos depois disso percorrer a lista de tamanhos de problemas e resolver todos os problemas guardando as suas soluções nas respectivas posições do array baseado na ordem e allocando apenas o necessário a solução de cada tamanho (um grafo por tamanho e uma array de palavras por tamanho).
Para finalizar acrescentaram se os edge cases de quando as palavras são iguais e então a solução é as duas palavras com peso de mutação zero e de quando as palavras têm apenas a diferença de um caracter em que a solução é novamente as duas palavras com peso de mutação um.

Arquitectura do programa

Fluxograma do programa:

Readfile.c



O fluxograma anterior mostra a arquitectura do programa e condensa a informação sobre as principais funções e os seus objectivos.

O programa inicia e evoca a função readfile() que por sua vez abre o ficheiro de problemas (.pal) e lê os problemas e usando a função dif() verifica se são casos especiais ou não.
Caso não seja caso especial e ainda não exista aquele tamanho na stack de tamanhos usa newsizelist() para dar push do novo tamanho para o stack de tamanhos (struct _sizelist que guarda tamanho da palavra e um ponteiro para o proximo elemento da stack) e dps dá push do problema para um stack de problemas num array com maximo tamanho de palavras 100 (struct _problemlist que guarda a ordem do problema, ambas as palavras, o peso maximo por mutação e um ponteiro para o proximo stack node).
Caso seja um caso especial usa newlist() para criar um stack com as duas palavras e guarda isso numa stack de problemas especiais usando o weight para guardar a ordem do problema (struct _problemlist).
De seguida cria um array de soluções do tamanho do numero de problemas que existem (que leva um weight, uma stack de palavras chamada path e um ponteiro para o proximo node na stack) após isso enquanto a stack de problemas especiais não tiver vazia vai se dando pop do elemento no topo da stack de problemas especiais e guarda-se a sua path na respectiva solução e calcula-se o weight da solução fazendo strcmp entre as duas palavras.
Agore vai se ao topo da stack de tamanhos e se for diferente de NULL cria se uma lista de words desse tamanho apartir do dicionário (.dic) de seguida usa se a função listtoarray() para transformar essa lista num sorted array de palavras. Cria se também o grafo específico desse tamanho com a função createGraph() (strutc _graph que leva um V que são o numero de vertices e um ponteiro de adjacencias chamado array) e adiciona-se todas as edges que não violem a regra da maxmutation.
De seguida resolve todos os problemas desse tamanho usando a função findpos() para encontrar a posição das palavras do problema no sorted array de palavras do mesmo tamanho e a função dijkstra() para calcular o melhor caminho e o peso do mesmo.
Quando se termina todos os problemas desse tamanho da-se free no array das palavras e no graph. Usando a função freegraph() para dar free ao graph. E segue-se para o proximo tamanho na stack e repete-se o processo anterior até encontrar o fim da stack ou seja NULL.

No final do exercicío da print do array de soluções que fomos preenchendo ao longo do programa.

Descrição das estruturas de dados

struct _problemlist *pcache[100]

Foi decidido que a melhor maneira de guardar os problemas seria guardar-los em varios stacks dependendo do seu tamanho para facilitar a sua solução sendo apenas necessario ter um graph e um array de palavras do mesmo tamanho allocado ao mesmo tempo. Chegou-se à conclusão que seria mais rápido ter um ponteiro para todos os tamanhos que o nosso programa considera (até 100) e depois ter um stack de tamanhos de problemas que têm problemas por resolver.

É necessário uma allocar *pcache para cada tamanho diferente de problema.

sizelist *psizes

Por causa da implementação da cache de problemas precisamos de uma stack que contêm os tamanhos diferentes que têm problemas para resolver. Assim temos esta stack que contém os weights e cada node também tem um pointer para o proximo node no stack.

É necessário dar push a uma psize para cada tamanho diferente de problema.

int maxmutweight[100]

Guarda-se também a mutação máxima para problemas de um certo tamanho para adicionar as edges que não violem esse valor criando assim um graph que tem todas as edges necessarias para fazer o dijkstra mas que não tem edges desnecessarias que nunca iram ser usadas porque violavam todos as mutações máximas.

São necessários 100 int's.

struct _problemsol *spsol

Concluiu-se que a melhor maneira de guardar os problemas especiais era usar a struct _problemsol visto que continha um int weight que é usado para guardar a ordem do problema especial, um path que é usado para guardar uma struce _list com ambas as palavras.

É necessário dar push de todos os problemas especiais.

struct _problemsol **sol

Chegou-se à conclusão que a melhor maneira de guardar as soluções seria num array de soluções em que o index do array corresponde à ordem dos exercicios permitindo assim introduzir soluções ordenadas directamente.

Alloca-se o array consoante o numero de problemas dentro do ficheiro de problemas.

struct _list *header

Usa-se uma stack de words para retirar todas as palavras de um determinado tamanho do ficheiro dicionario (.dic)

É necessários dar push a todas as palavras de um determinado tamanho no dicionário (.dic).

char **wcache

Transforma-se a stack de words de um determinado tamanho num array organizado de palavras.

É preciso um wcache por cada tamanho diferente de problema a resolver mas apenas um é usado de cada vez.

struct Graph* graph

Guarda numero de vertices e um pointer para uma AdjList array com o tamanho igual ao numero de vertices.

É preciso um graph por cada tamanho diferente de problema a resolver mas apenas um é usado de cada vez.

struct AdjListNode* newNode

É usado para guardar adjacencias entre 2 vertices. Guarda o index de dest o peso da aresta e um pointer para o proximo valor na lista.

São precisos 2 AdjListNode's por cada edge adicionada.

struct AdjList* array

É usado para dar point para lista de adjacencias. É apenas um ponteiro.

É preciso allocar AdjList para cada vertice.

struct MinHeap* minHeap

Foi concluído que a melhor maneira de criar uma priority queue para o dijkstra era usando um heap. Guarda-se a size da heap, a sua capacidade, as posições dos seus nodes e um array de MinHeapNode's do tamanho da capacidade.

É necessário uma heap por problema na lista de problemas.

struct MinHeapNode* minHeapNode

Node da heap que contém o index do vertice e o peso da distancia ao mesmo.

É necessário um node destes por cada vertice do graph.

Descrição de algoritmos

Os principais algoritmos utilizados no projecto são os que se encontram nas funções **quicksort_strs**(), **findpos()**, **dijkstra()**.

quicksort_strs()

Quicksort é uma algoritmo de divisão e conquista. O algoritmo primeiro divide um array grande em dois arrays mais pequenos, o array com os elementos maiores e o array com os elementos menores de seguida chama-se recursivamente para os dois arrays resultantes.

Passo a passo da implementação do **quicksort_strs()**:

1) Troca valor random com o fim da do array
2) Partitioning: reorganiza o array de maneira a que todos os elementos com valores menores que o pivot ficam antes do pivot e todos o que tem valores maiores ficam depois do pivot
3) Move o pivot para o seu lugar
4) Evocar recursivamente a função **quicksort_strs()** com ambos os sub-arrays

findpos()

Binary search é um algoritmo que encontra a posição de um valor definido numa sorted array. O algoritmo compara o valor dado com o centro da array: se for igual então encontramos a solução; se for maior então chama recursivamente **findpos()** com o novo array desde o meio do array actual até ao final do array; se for menor então chama recursivamente **findpos()** com o novo array desde o inicio do array até ao meio do array actual.

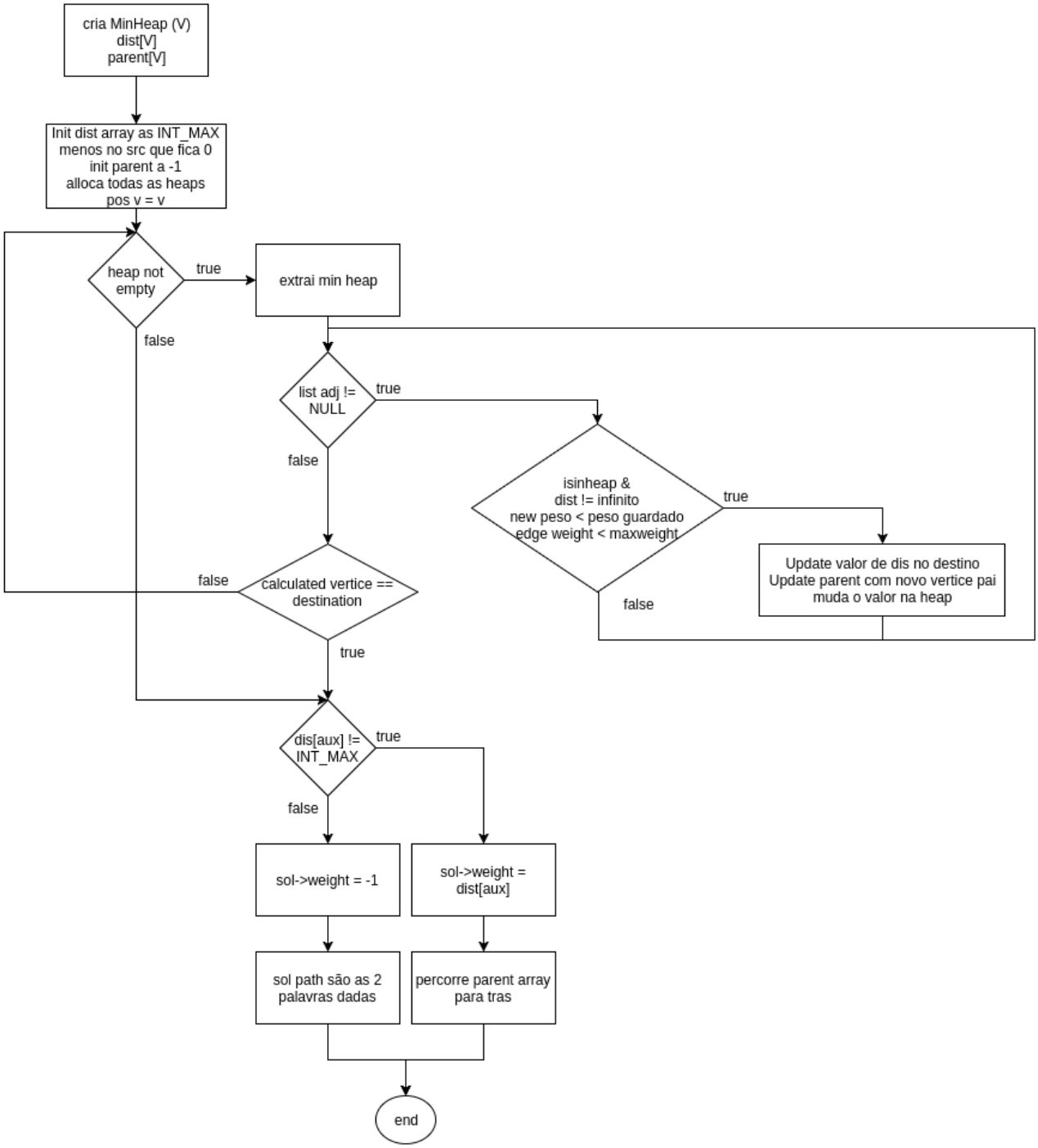
dijkstra()

O algoritmo de Dijksra é um algoritmo para encontrar o caminho mais curto entre vertices de um graph.

Passo a passo da implementação do **dijkstra()**:

1. Criar MinHeap, dist e parent com tamanho igual ao tamanho dos vertices do graph
2. Inicia a distancia acumulada como INT_MAX para todos os vertices menos o src que inicia com distantcia acumulada 0, inicia parent a -1, alloca todas as heap nodes, inicia todas as pos dos vertices no seu respectivo index
3. Heap não vazia (caso esteja vazia salta para 13)
4. Extrai valor min de heap
5. Lista de Adjacencias não chegou ao NULL (se está no NULL salta para passo 11)
6. Se dest está na heap e dist não é INT_MAX e o peso novo é menor que o peso guardado e o peso da edge respeita o maxmut condition
7. Update valor de dist no destino
8. Update parent com novo vertice pai
9. Mudar valor do vertice na heap
10. Proxima adjacencia e salta para o passo 5
11. Se o index da heap que acabamos de calcular for igual a destino (se não salta para 3)
12. Se o peso acumulado do vertice que acabamos de calcular for diferente de INT_MAX então o peso da solução é distancia acumulada nesse vertice caso contrario peso é igual a -1
13. Caso exista solução percorre parent array para trás até chegar à src guardando todas as palavras numa lista caso contrario mete apenas as 2 palavras dadas na lista

Fluxograma do dijkstra:



Descrição dos subsistemas

O programa por nós implementado tem o subsistema dijkstra, contituido por dijkstra.c e dijkstra.h onde estão guardadas as funções que estão directamente ligadas à execução do algoritmo de Dijkstra. As estruturas de dados ligadas a execução do algoritmo de Dijkstra estão definidas no ficheiro dijkstra.h.
O subsistema list constituído por list.c e list.h que contém as funções directamente relacionadas com a criação de stacks. As estruturas de dados ligadas a estruturação dos problemas e das soluções estão definidas no ficheiro file.h.
O subsistema dif constituído por dif.c e dif.h que contém as funções directamente relacionadas com o calculo da diferença entre dois caracteres.
O subsistema quicksort constituído por quicksort.c e quicksort.h que contém as funções directamente relacionadas com a execução do quicksort.

dijkstra.c

Funções de allocação de memória

struct AdjListNode* **newAdjListNode**(int dest, int weight);

- Creates new node for adjlist with given dest and weight

struct Graph* **createGraph**(int V);

- Creates graph based on number of vertices given

struct MinHeapNode* **newMinHeapNode**(int v, int dist);

- Adds new node to heap with destination v and weight dist

struct MinHeap* **createMinHeap**(int capacity);

- Creates heap of given capacity

void **addEdge**(struct Graph* graph, int src, int dest, int weight);

- Adds edge to given graph with src, dest and weight

Funções de manipulação de heap

void **swapMinHeapNode**(struct MinHeapNode** a, struct MinHeapNode** b);

- Swaps given heap nodes a and b

void **minHeapify**(struct MinHeap* minHeap, int idx);

- Recursive heapify's heap from given point

int **isEmpty**(struct MinHeap* minHeap);

- checks if heap is empty

struct **MinHeapNode*** extractMin(struct MinHeap* minHeap);

- Extracts the lowest value of the heap

void **decreaseKey**(struct MinHeap* minHeap, int v, int dist);

- Change value of given edge in the heap to dist

bool **isInMinHeap**(struct MinHeap *minHeap, int v);

- Verify if given edge is in heap

Dijkstra

void **dijkstra**(struct Graph* graph, int src, int dest, problemsol **sol, char **wcache, int order, int maxmut);

- Performs the dijkstra algorithm in graph with src being the pos of first word and dest being the pos of second word also takes sol to store calculated path and weight, wcache to calculate paths based on indexes, problem order store solution in right order and problem maxweight to only use edges that don't violate that value

Funções de libertação de memória

void **freegraph**(struct Graph* graph, int V);

- Free given graph

list.c

Funções de allocação de stacks

list ***newlist**(char *word, list *next);

- Allocs new list stack node and pushes it to the stack

problemlist ***newproblemlist**(int order, char *pal1, char *pal2, int maxmut, problemlist *next);

- Allocs new problem list stack node and pushes it to the stack

sizelist ***newsizelist**(int size, sizelist *next);

- Allocs new size stack node and pushes it to the stack

problemsol ***newproblemsol**(int weight, list *path);

- Allocs new problem solution stack node

problemsol ***newspecialsol**(int weight, list *path, problemsol *next);

- Allocs new special solution stack node and pushes it to the stack

dif.c

Função de cálculo

int **dif**(char *a, char *b, int size);

- Calculates dif between 2 words of given size

quicksort.c

Funções de manipulação

void **swap_str_ptrs**(char **arg1, char **arg2);

- Swaps string pinters

void **quicksort_strs**(char *args[], unsigned int len);

- Executes quicksorting on array of given size

Análise dos requesitos computacionais

As várias escolhas para estruturas de dados do problema foram feitas com a diminuição do tempo de acesso em mente.

struct problemlist *pcache[100]:

O facto de ser um array permite ser acedido directamente com complexidade **O(1)** o que ajuda imenso visto que se quisesse guardar os problemas por tamanhos diferentes numa lista ia necessitar algum tipo de sorting que iria fazer o acesso mais lento. Também não gasta muita memória visto que são apenas pointers e que mesmo quando são mais que pointers são apenas os testes do exercício em mão que em comparação com outras necessidades de memória faz esta parecer ainda melhor visto as vantagens de velocidade.

sizelist *psizes:

O facto de ser uma lista com apenas os diferentes tipos de tamanho que devem ser testados leva a que tenha apenas que percorrer esta lista e sei quais são os pcahes que devo visitar e trabalhar, evitando ter que correr o array todo da pcache. A complexidade de acesso a esta estrutura pode tmb ser considerada **O(1)** visto que sempre que preciso do valor ele se encontra no topo da stack e não necessito fazer qualquer procura.

int maxmutweight[100]:

O facto de ser um array facilita o acesso sendo apenas de complexidade **O(1)** e como são apenas 100 int's também é considerado um tamanho reduzido.

struct problemsol *spsol:

Sendo que sempre que preciso um valor desta estrutura este encontra-se no topo da stack podemos considerara complexidade de acesso **O(1)** porque não existe qualquer tipo de procura. Guarda os problemas especiais mas não gasta mais memoria do que precisa porque é exclusivo desta stack não existindo na stack de problemas normal.

struct _problemsol **sol:

Sendo um array facilita o acesso sendo apenas de complexidade **O(1)**. Guarda todas as soluções, com todas as palavras de caminho entre a pal1 e a pal2, apenas em edge cases em que precisasse de calcular muitos de milhares de problemas e cada um tivesse um path de solução mais ou menos grande podia precisar de demasiada memória mas nesses casos seria provavel que não ter este tipo de solução faria o problema gastar ainda mais memória ou demorar muito.

struct _list *header:

Sendo que sempre que preciso um valor desta estrutura este encontra-se no topo da stack podemos considerara complexidade de acesso **O(1)** porque não existe qualquer tipo de procura. Guarda todas as palavras do mesmo tamanho o que não gasta mais memoria do que precisa porque apenas alloc paralavras 1 vez e dps na transferencia de list para arry mudo so os pointers (o que poupa tempo e memória).

char **wcache:

Sendo um array facilita o acesso sendo apenas de complexidade **O(1)**. Guarda todas as palavras de um determinado tamanho o que normalmente não é um problema pq não existem assim tantas palavras e o array só mantém as words o tempo que precisa libertanto logo após completar todos os problemas daquele tamanho.

Utilização de Heap:

Foi concluido que a heap daria a melhor solução tendo em conta a que oferecia uma boa priority queue com complexidade de **O(log n)** para encontrar o minímo e também **O(log n)** para verificar se MinHeap node está na priority queue.

struct Graph* graph:

Consome alguma memoria dependo do tamanho do dicionario (.dic) visto que crio um grafo com tamanho de todas palavras do tamanho a ser testado. Onde consome mais memória e gasta bastante tempo é quando se alloca todas as edges (isto depende da maxmutweight encontrada no ficheiro de problemas para um dado tamanho sendo que só guarda aqueles valores que não violam a regra maxima) no grafo e essa é uma das principais razões para usar um sistema se separa os problemas em tamanhos diferetes para poder calcular o graph, usar e depois apaguar quando não preciso mais. Acesso rapido (complexidade **O(1)**) ao tamanho de graph e a específica lista de adjacências mas se precisar de especifica adjacencia preciso de percorrer a lista até encontrar mas como nunca faço não é relevante.

Principais algoritmos:

quicksort_strs();

O quicksort têm uma complexidade em média de **O(n log n)** mas pode alcançar **O(n²)** raramente. Tendo em conta isto e comparando com outros algoritmos de sorting chegamos a conclusão que quicksort é a melhor opção.

findpos();

A binary search têm complexidade em média de **O(log n)** o que a torna muito superior à outra hipótese imediata de percorrer o array todo causando uma constante complexidade de **O(n)**.

dijkstra();

O algoritmo de dijkstra mostra uma complexidade de **O(E*log V)** onde E é o numero de edges e V é o numero de vertices no graph.

Funcionamento do programa

O teste máximo passado pelo nosso codigo foi 15 por termos começado as submissões tarde e já termos ficado overlaped com muitas submissões restringindo a velocidade do processador nos ultimos testes. De qualquer maneira acreditamos que baseado na arquitectura do nosso programa e como lidamos com os problemas conseguiriamos um 18 não passando nos testes 19 e 20 por não termos em consideração no nosso codigo casos como o gato e o rato que sabemos como resolver mas não interessa porque não podemos submeter uma nova solução.
A solução para o caso do gato e rato seria quando se calculasse as diferenças entre gato e as outras palavras e se verificasse que só existiam caminhos de 2 que não são menores que o peso directo entre gato e rato então não calculava o dijkstra porque teria já a solução.