2

|  |
| --- |
| Estruturas de Dados AvançadasRelatório do Projeto de Avaliação – Fase 2 |

Aluno: Miguel Areal – 29559 – a29559@alunos.ipca.pt

Docente: Luís Ferreira

Conteúdo

[Estruturas de Dados Avançadas 1](#_Toc167038886)

[Relatório do Projeto de Avaliação – Fase 2 1](#_Toc167038887)

[Introdução 3](#_Toc167038888)

[Descrição da estrutura do projeto 4](#_Toc167038889)

[Instruções de compilação e execução 5](#_Toc167038890)

[Bibliotecas utilizadas 5](#_Toc167038891)

[Estruturas de Dados 6](#_Toc167038892)

[Funcionalidades 7](#_Toc167038893)

[Leitura e inicialização do grafo através de um ficheiro 7](#_Toc167038894)

[Exibição do grafo 8](#_Toc167038895)

[Soma de pesos de um caminho 8](#_Toc167038896)

[Alterar peso de uma aresta 9](#_Toc167038897)

[Remoção de aresta 10](#_Toc167038898)

[Adição de aresta 11](#_Toc167038899)

[Encontrar a maior soma de pesos possível 12](#_Toc167038900)

[Funções Auxiliares 13](#_Toc167038901)

[Casos de teste 14](#_Toc167038902)

[Caso 1 14](#_Toc167038903)

[Bibliografia / Referências 14](#_Toc167038904)

## Introdução

A motivação para este projeto surge da necessidade de aplicar conceitos avançados de teoria dos grafos e programação em C para resolver problemas computacionais de com algum nível de complexidade. Este projeto está inserido na disciplina de estruturas de dados avançadas e algoritmos, com um foco especial na implementação de soluções eficientes para manipulação e análise de grafos.

Neste contexto, a segunda fase do projeto visa desenvolver uma aplicação capaz de calcular o somatório máximo possível de inteiros a partir de uma matriz, utilizando regras específicas de conexão entre os elementos. A aplicação será construída sobre estruturas de dados dinâmicas que representam um grafo, vértice, aresta, e caminhos permitindo a modelagem e resolução de problemas complexos através da utilização de algoritmos de procura.

O objetivo principal desta fase é aprender a definir e manipular estruturas de dados dinâmicas, nomeadamente grafos, e aplicar algoritmos para encontrar soluções ótimas em problemas de somas. Para alcançar este objetivo, foram estabelecidas várias tarefas, incluindo a definição de estruturas de dados, modelagem do problema, implementação de algoritmos de procura e manipulação de grafos, e desenvolvimento de funções para cálculo de somas e identificação de caminhos.

## Descrição da estrutura do projeto

Este projeto está organizado de forma a separar claramente os componentes do código-fonte, dados e documentação:

* Os diretórios *doc/* e *data/* mantêm a documentação gerada em *Doxygen* e os ficheiros de dados usados pelo programa, respetivamente.
* O diretório *include/* mantém o ficheiro principal de *header* que que contém o protótipo e documentação da função principal (*main*).
* Já o diretório *src/* contém o ficheiro-fonte da aplicação (*main.c*), que tem como função meramente utilizar as implementações da biblioteca.
* O diretório *lib/* possui os ficheiros relativos ao código da biblioteca utilizada. Esta é composta por *lib.c*, *lib\_aux.c e* *lib.h.*
* *obj/* contém os ficheiros de ‘*output’* (*main.o,* *lib.o e lib\_aux.o*) gerados pelo GCC através da compilação dos ficheiros que compõem o software.
* O ficheiro *Makefile* automatiza o processo de compilação do projeto, enquanto *README.md* fornece informações básicas sobre o projeto.
* *‘lib.dll’* é a solução final de todos os componentes relacionados com a biblioteca.
* Uma imagem com texto, captura de ecrã, software, Software de multimédia

  Descrição gerada automaticamente‘*PFase2.exe’* é o executável gerado após a compilação do projeto que também compila a biblioteca ao mesmo efeito.

## Instruções de compilação e execução

Para a compilação e execução do programa, é apenas necessário utilizar o comando *make* a partir do diretório raíz do projeto.

Após a compilação do projeto, é gerado um novo executável ‘PFase2.exe’ e lib.dll, na raíz do repositório.

Para executar esta aplicação, é possível de duas formas:

Ambiente gráfico: Duplo-clique no ficheiro PFase2.exe;

CLI / Consola: Navegar para o diretório raíz do projeto, e executar ‘./PFase2.exe’;

## Bibliotecas utilizadas

Para este projeto foram utilizadas algumas bibliotecas externas, sendo estas:

* stdio.h – “*Standard input-output header”.* Biblioteca padrão de projetos em linguagem C, que possui definições de subrotinas relativas a operações de entrada/saída, leitura de dados, exibição de informações, etc.
* stdbool.h – Utilizada para manipular variáveis lógicas, como verdadeiro e falso. Veio a substituir a utilização de números inteiros para esse efeito, e serve principalmente para facilitar a compreensão.
* stdlib.h – Permite utilizar funções de alocação de memória dinâmica, conversões de *string* para números e outras.
* string.h – Contém funções e macros para manipulação de *strings* e regiões de memória.
* limits.h – Define as constantes de limite de tipos de dados como ‘*INT\_MAX’*, ‘*INT\_MIN’*, etc.

## Estruturas de Dados

As estruturas de dados utilizadas neste software foram as listas ligadas e por consequência, apontadores.

As estruturas base do projeto são ‘*GR’*, ‘*Aresta’* e ‘Vertice’.

*GR* representa o grafo, esta é composta pelo número de vértices que compõe o grafo, e um apontador para a estrutura *Vertice*.

A estrutura Vertice representa um item de vértice do grafo, este é composto pelo seu valor, o número de arestas a que está conectado, um apontador para a estrutura *Aresta*

A estrutura *Aresta* representa uma lista ligada de conexão ao vértice, onde cada aresta tem um peso, um apontador para o vértice destino, e um apontador para a próxima aresta da lista.

A estrutura *Caminho* representa uma lista ligada, onde cada caminho tem um tamanho, um apontador para o próximo caminho, e um apontador para vértices.

typedef struct Vertice {

int valor;

struct Aresta \*arestas;

int num\_arestas;

} Vertice;

typedef struct Aresta {

struct Vertice \*destino;

int peso;

struct Aresta \*proxima;

} Aresta;

typedef struct GR {

Vertice \*vertices;

int num\_vertices;

} GR;

typedef struct Caminho {

int \*vertices;

int tamanho;

struct Caminho \*proximo;

} Caminho;

*Excerto 1 – Estruturas de dados principais.*

Funcionalidades

Pelas implementações necessárias no software, o mesmo propõe diferentes funcionalidades:

Leitura e inicialização do grafo através de um ficheiro

A funcionalidade de leitura de dados permite a importação de informações de um ficheiro externo (por exemplo, ./data/input.txt) para serem utilizadas como a matriz de entrada do programa. Este processo é realizado pela função **carrega\_fich,** que recebe como parâmetro o caminho para o ficheiro.

Primeiramente, a função abre o ficheiro no modo de leitura. Se ocorrer um erro ao abrir o ficheiro, a função imprime uma mensagem de erro e retorna NULL.

A seguir, a função percorre o ficheiro linha por linha para contar o número de linhas e determinar o número máximo de colunas, separando os valores pelo caractere “;”. Para cada linha lida, os tokens (valores) são extraídos e contados. Se um token não for vazio ou nulo, o contador de colunas é incrementado. O número total de linhas é também incrementado para cada linha lida. No final dessa fase, o número máximo de colunas é comparado e atualizado se necessário.

Depois de determinar as dimensões da matriz, a função utiliza rewind para voltar ao início do ficheiro. Com base no maior número entre linhas e colunas, é determinado o tamanho do grafo, para que se torne numa matriz quadrada.

A estrutura do grafo é então inicializada chamando a função **criar\_grafo**, que recebe o tamanho determinado. Novamente, o ficheiro é percorrido linha por linha. Para cada linha, os tokens são separados por “;” e convertidos para inteiros. Se um token não for vazio, nulo, ou zero, a função **add\_aresta** é chamada para adicionar uma aresta entre os vértices correspondentes, utilizando o valor do token como peso da aresta.

A leitura da matriz para um grafo dá-se por entender que cada coordenada representa uma conexão entre dois vértices. Cada índice de coluna e linha representa um vértice.

Para uma melhor visualização de como a matriz é traduzida em grafo:

Uma imagem com diagrama, file, círculo

Descrição gerada automaticamente

|  |  |  |  |
| --- | --- | --- | --- |
|  | V1 | V2 | V3 |
| V1 | 5 | 6 | 0 |
| V2 | 15 | 25 | 34 |
| V3 | 13 | 27 | 0 |

Ao final do processo, o ficheiro é fechado e a função retorna o grafo inicializado e preenchido com os valores lidos do ficheiro.

GR \*carrega\_fich(const char \*nomefich);

GR\* criar\_grafo(int num\_vertices);

7;53;183;439;863

497;383;563;79;973

287;63;343;169;583

627;343;773;959;943

767;473;103;699;303

*Figura 2 – Conteúdo exemplar do ficheiro ‘input.txt’*

Soma de pesos de um caminho

A função *‘****soma\_pesos\_caminho’***é responsável por calcular a soma dos pesos das arestas num caminho específico dentro de um grafo.

A função opera da seguinte maneira: Começa por verificar se o caminho fornecido é válido. Isto inclui verificar se o apontador do caminho não é nulo e se o tamanho do caminho é maior que zero. Se qualquer uma dessas condições não for satisfeita, a função imprime uma mensagem de erro e retorna 1.

Percorre então cada par de vértices consecutivos no caminho fornecido. Para cada par, ela verifica se os índices dos vértices estão dentro dos limites válidos do grafo. Se algum índice for inválido, a função imprime uma mensagem de erro e retorna 1.

Para cada par de vértices, a função busca a aresta que conecta o vértice atual ao próximo vértice. Isso é feito percorrendo a lista de arestas do vértice atual. Se a aresta que conecta os dois vértices for encontrada, o seu peso é adicionado à soma total dos pesos.

Se todas as arestas necessárias forem encontradas, a função retorna a soma total dos pesos das arestas no caminho.

Alterar peso de uma aresta

É possível alterar o peso de uma aresta, recorrendo à função ***‘alterar\_peso’***.

Esta tem como parâmetro a instância do grafo já criado, o número do índice do vértice de origem e destino, e o novo valor para substituir o peso existente.

Primeiramente, verifica que os valores passados por parâmetros são válidos. Estes não podem ser menor que 0, nem podem estar fora dos limites do grafo. Caso verifique um erro, mostra uma mensagem e retorna 1.

Após isso, são criados dois apontadores do tipo *Vertice*, que tomam como valor a posição na memória dos vértices de origem e destino.

Percorre então a lista de arestas do vértice de origem, e se encontrar a aresta que se conecta ao vértice de destino, atualiza o peso dado pelo parâmetro, e retorna 0.

Se não encontrar a aresta que os conecta, mostra uma mensagem de erro, e retorna 1.

Remoção de aresta

Para remover uma aresta do grafo, recorremos à função **‘*rem*\_*aresta’***. Esta função é responsável por eliminar uma aresta que conecta dois vértices do grafo.

A função toma como parâmetros a instância do grafo, o índice do vértice de origem, e o índice do vértice de destino.

A função começa por verificar se o grafo é nulo e se os índices indicados estão dentro dos limites válidos do grafo. Caso qualquer uma dessas verificações falhe, a função imprime uma mensagem de erro indicando que os vértices são inválidos e retorna 0.

São criados dois apontadores do tipo *Vertice*, *origem* e *destino*, que são inicializados com os endereços de memória dos vértices de origem e destino, conforme fornecidos pelos parâmetros. Isso permite que a função aceda diretamente às listas de arestas associadas a cada vértice.

A função então utiliza dois apontadores do tipo *Aresta*, *anterior* e *atual*, para percorrer a lista de arestas do vértice de origem. O apontador *anterior* é utilizado para manter o rastro da aresta anterior na lista, enquanto o apontador *atual* é utilizado para percorrer a lista de arestas.

Dentro do ciclo, a função verifica se a aresta atual conecta o vértice de origem ao vértice de destino. Se encontrar a aresta, a função a remove da lista da seguinte forma:

* Se a aresta a ser removida é a primeira na lista de arestas, o apontador *origem->arestas* é atualizado para apontar para a próxima aresta.
* Se a aresta a ser removida não é a primeira, o apontador *anterior->proxima* é atualizado para apontar para a próxima aresta, removendo assim a aresta atual da lista.

Após remover a aresta, a função liberta a memória alocada para a estrutura da aresta utilizando *free(atual)* e retorna 0 para indicar sucesso.

Se a função percorre toda a lista de arestas sem encontrar a aresta que conecta os vértices especificados, ela imprime uma mensagem de erro indicando que a aresta não foi encontrada e retorna 1.

Adição de aresta

A função ‘***add\_aresta***’ é responsável por adicionar uma aresta ao grafo, ligando dois vértices com um peso especificado.

A função toma como parâmetros dois vértices, *v1* e *v2*, e um valor inteiro *peso*, que representa o peso da aresta que conecta *v1* a *v2*.

A primeira operação da função é alocar memória para uma nova aresta usando *malloc*. Se a alocação de memória falhar, a função retorna 1 para indicar a falha. Caso contrário, a função prossegue configurando os atributos da nova aresta:

* *nova\_aresta->destino* é configurado para apontar para o vértice v2, estabelecendo assim a conexão da aresta de v1 para v2.
* *nova\_aresta->peso* é configurado com o valor do peso passado como parâmetro.
* *nova\_aresta->proxima* é configurado para apontar para a primeira aresta da lista de arestas de v1. Este passo serve para manter a integridade da lista ligada de arestas, colocando a nova aresta no início da lista.

Após configurar a nova aresta, a função atualiza o apontador v1->arestas para apontar para a nova aresta, fazendo com que a nova aresta seja a primeira da lista de arestas de v1. A função também incrementa o contador de arestas do vértice v1 (v1->num\_arestas) para refletir a adição da nova aresta.

Finalmente, a função retorna 0 para indicar que a aresta foi adicionada com sucesso.

Encontrar a maior soma de pesos possível

A função **‘*encontrar\_maior\_soma\_caminho’*** é responsável por encontrar o caminho de maior soma de pesos no grafo. Esta função explorar todos os possíveis caminhos no grafo e identifica aquele/s cuja soma dos pesos das arestas é a maior.

Ela funciona em conjunto com a função **‘dfs’** (*Depth-First Search*), que é responsável por realizar a busca em profundidade a partir de um vértice específico.

O processo começa com a inicialização de variáveis importantes: ‘*maior\_soma’*, que armazena a maior soma encontrada, é inicialmente definida como o menor valor possível para um inteiro (‘INT\_MIN’). Isso garante que qualquer soma válida encontrada será maior que este valor inicial. A variável do tipo Caminhos ‘*melhores\_caminhos’* é inicializada como NULL e será utilizada para armazenar os caminhos que resultam na maior soma encontrada.

A função então itera sobre todos os vértices do grafo. Para cada vértice, são alocados dois vetores: ‘*visitados’*, que marca quais vértices foram visitados durante a busca, e ‘*caminho\_atual*’, que armazena o caminho atual enquanto a busca é realizada. Estes arrays são essenciais para acompanhar o progresso da busca e garantir que cada caminho seja explorado corretamente.

Em seguida, a função chama ‘*dfs’* para realizar uma busca em profundidade a partir do vértice atual. Explora todos os caminhos possíveis a partir do vértice de origem, calculando a soma dos pesos das arestas para cada caminho. Se a soma do caminho atual for maior que ‘*maior\_soma’*, esta variável é atualizada e a lista de ‘*melhores\_caminhos*’ é limpa e preenchida com o novo caminho. Se a soma for igual a ‘*maior\_soma*’, o novo caminho é adicionado à lista de melhores caminhos, permitindo o armazenamento de múltiplos caminhos com a mesma soma máxima.

A ‘dfs’ continua a busca recursivamente para cada vértice adjacente não visitado, permitindo a exploração completa de todos os caminhos possíveis no grafo, aumentando os níveis de profundidade. Após explorar todos os vértices adjacentes, a função desmarca o vértice atual como visitado para permitir outras buscas.

Depois de todas as iterações e buscas, a função `*encontrar\_maior\_soma\_caminho*` exibe os resultados na consola, mostrando a maior soma encontrada e os caminhos correspondentes. Se nenhum caminho for encontrado, uma mensagem é exibida.

Finalmente, a função liberta a memória alocada para os caminhos, recorrendo a uma função auxiliar ‘*limpa\_caminhos’*, antes de retornar 0.

Funções Auxiliares

Estas são funções que estão criadas no ficheiro auxiliar da biblioteca ‘lib\_aux.c’. Utilizam o mesmo *header* que a biblioteca *(lib.h).*

São funcionalidades menores, que foram separadas por diferentes razões, mas principalmente por melhor leitura e análise de código.

## Exibição do grafo

Após a leitura e inicialização das estruturas de dados, o grafo é exibido pela função ***print\_grafo***. Esta função percorre todos os vértices do grafo e, para cada vértice, exibe as suas arestas e os pesos correspondentes.

A função funciona da seguinte maneira: Percorre todos os vértices do grafo utilizando um ciclo. Para cada vértice, a função exibe o seu valor. Os vértices são numerados de 1 a n (em vez de 0 a n) para facilitar a leitura.

A função printf é utilizada para formatar e imprimir o valor do vértice. Após exibir o valor do vértice, a função percorre a lista de arestas conectadas a este vértice. Para cada aresta, a função imprime o vértice de destino e o peso da aresta.

Após exibir todas as arestas de um vértice, a função insere uma nova linha antes de passar para o próximo vértice. Isto garante que cada vértice e as suas arestas sejam exibidos numa linha separada.

for (int i = 0; i < grafo->num\_vertices; i++) {

Vertice v = grafo->vertices[i];

printf("V%d: ", v.valor + 1); // Vértices numerados de 1 a n (em vez de 0 a n)

// Imprime informações sobre as arestas conectadas ao vértice

for (Aresta \*a = v.arestas; a != NULL; a = a->proxima) {

printf("-> V%d (Peso %d) ", a->destino->valor + 1, a->peso);

}

printf("\n");

}

V1: -> V5 (Peso 863) -> V4 (Peso 439) -> V3 (Peso 183) -> V2 (Peso 53) -> V1 (Peso 7)

V2: -> V5 (Peso 973) -> V4 (Peso 79) -> V3 (Peso 563) -> V2 (Peso 383) -> V1 (Peso 497)

V3: -> V5 (Peso 583) -> V4 (Peso 169) -> V3 (Peso 343) -> V2 (Peso 63) -> V1 (Peso 287)

V4: -> V5 (Peso 943) -> V4 (Peso 959) -> V3 (Peso 773) -> V2 (Peso 343) -> V1 (Peso 627)

V5: -> V5 (Peso 303) -> V4 (Peso 699) -> V3 (Peso 103) -> V2 (Peso 473) -> V1 (Peso 767)

## Criar Vértices

A função *criar\_vertice* serve para criar e inicializar um novo vértice no grafo. Esta função é fundamental para a construção da estrutura do grafo, garantindo que cada vértice seja corretamente configurado antes de ser utilizado.

Recebe um parâmetro inteiro que representa o valor ou identificação do vértice a ser criado. Este valor é usado para inicializar o campo valor do vértice.

A primeira operação é a alocação de memória para um novo vértice. Isso é feito utilizando a função *malloc*, que reserva um bloco de memória do tamanho da estrutura *Vertice*. Se a alocação for bem-sucedida, o ponteiro v passa a apontar para o novo bloco de memória alocado.

Após a alocação, a função inicializa os campos do vértice:

* v->valor é configurado com o valor passado como parâmetro, estabelecendo a identificação ou valor do vértice.
* v->arestas é inicializado como NULL, indicando que, inicialmente, o vértice não possui arestas conectadas a ele. Este campo será posteriormente utilizado para armazenar uma lista ligada de arestas.
* v->num\_arestas é inicializado com 0, indicando que o vértice não possui arestas conectadas na altura da criação.

Finalmente, a função retorna o ponteiro v, que agora aponta para o novo vértice inicializado. Este vértice pode então ser utilizado em outras operações no grafo, como na adição de arestas.

## Adicionar Caminhos

A função ***‘adicionar\_caminho’*** é responsável por adicionar um novo caminho a uma lista de caminhos representada pela estrutura *Caminho*. Esta função é crucial para gerir múltiplos caminhos possíveis dentro de um grafo.

A função recebe três parâmetros:

* *Caminho \*\*lista*: Um ponteiro duplo para a lista de caminhos. Isso permite que a função modifique a lista original, adicionando o novo caminho no início.
* *int \*caminho*: Um ponteiro para um array de inteiros que representa o caminho a ser adicionado.
* i*nt tamanho*: Um inteiro que especifica o número de vértices no caminho.

A função começa por alocar memória para um novo nó do tipo *Caminho* usando *malloc.* Em seguida, a função aloca memória para armazenar os vértices do novo caminho. O tamanho dessa memória é baseado no número de vértices (tamanho).

A função copia os vértices do array fornecido (caminho) para o array de vértices do novo caminho, usando a função *memcpy.* Configura então o campo *tamanho* do novo caminho para o valor fornecido pelo parâmetro tamanho.

Insere o novo caminho no início da lista de caminhos. O campo *proximo* do novo caminho é configurado para apontar para o início da lista atual. Em seguida, o ponteiro *\*lista* é atualizado para apontar para o novo caminho, tornando-o o primeiro elemento da lista.

No fim, retorna 0, para indicar que a função teve sucesso.

## Remover Caminhos

A função ***’limpar\_caminhos’*** é responsável por garantir que toda a memória alocada para a estrutura de dados da lista de caminhos seja libertada adequadamente.

É fundamental para evitar problemas de memória em programas que utilizam estruturas dinâmicas, como listas ligadas e apontadores.

A função utiliza um ciclo while para percorrer cada nó na lista de caminhos. Continua até que lista aponte para *NULL*, indicando que o final da lista foi alcançado.

Dentro do ciclo, a função segue os seguintes passos para liberar a memória de cada nó:

* A função armazena o ponteiro para o nó atual numa variável temporária (temp). Isso é necessário porque lista será atualizado para apontar para o próximo nó antes da libertação da memória do nó atual.
* O ponteiro *lista* é atualizado para apontar para o próximo nó na lista. Isso permite que o ciclo continue a percorrer a lista após a libertação do nó atual.
* A função liberta a memória alocada para o array de vértices do caminho atual, e no fim a memória alocada do próprio nó *temp*, usando *free*().

## Libertar Memória

texto.

Casos de teste

Foram executados alguns testes, com matrizes diferentes, para verificar a fiabilidade, eficiência e comportamento do software.

## Caso 1

Para este primeiro caso, foi utilizada a matriz dada pelo enunciado:

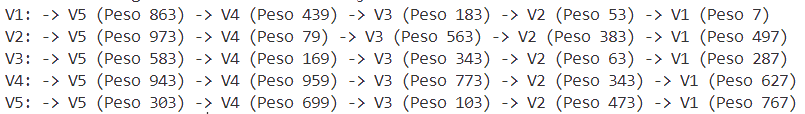
7;53;183;439;863

497;383;563;79;973

287;63;343;169;583

627;343;773;959;943

767;473;103;699;303

Após a leitura e inicialização do grafo, deu-se que a conexões entre os vértices seriam as seguintes:

# Bibliografia / Referências

Durante todas as etapas deste projeto, foram utilizadas várias fontes de informação, entre as quais:

* <https://www.tutorialspoint.com/c_standard_library/index.htm>
* <https://www.doxygen.nl/manual/docblocks.html>
* <https://stackoverflow.com>
* <https://code.visualstudio.com>
* <https://gnuwin32.sourceforge.net/packages/make.htm>
* <https://sourceforge.net/projects/mingw/files/>
* <https://elemarjr.com/clube-de-estudos/artigos/o-algoritmo-hungaro-e-a-solucao-de-problemas-de-atribuicao/>
* <https://chat.openai.com>
* <https://devdocs.io/c/>
* <https://en.wikipedia.org/wiki/Hungarian_algorithm>
* <https://www.geeksforgeeks.org/what-is-linked-list/>
* <https://pubs.opengroup.org/onlinepubs/009695399/basedefs/stdbool.h.html>
* <https://www.tutorialspoint.com/c_standard_library/ctype_h.htm>
* <https://www.tutorialspoint.com/c_standard_library/limits_h.htm>
* <https://www.simplilearn.com/tutorials/data-structure-tutorial/dfs-algorithm>
* <https://www.cs.yale.edu/homes/aspnes/pinewiki/C(2f)Graphs.html>