Licenciatura em Engenharia de Sistemas Informáticos

Unidade Curricular: Estruturas de Dados Avançadas
Docente: Luís Ferreira

Trabalho realizado por:
Gonçalo Gomes, a23039

Instituto Politécnico do Cávado do Ave
2º Semestre 2023/2024

Lista de abreviaturas e siglas

IPCA	Instituto Politécnico do Cávado e do Ave
EOF	*End Off File*
NULL	*Vazio*

Índice

1. Introdução 4

2. Funcionalidades a Implementar 5

3. Implementação das funcionalidades 6

3. Conclusão 12

1. Introdução

O presente trabalho surge no âmbito da unidade curricular de Estruturas de Dados Avançadas, que faz parte do plano curricular da Licenciatura em Engenharia de Sistemas Informáticos da Escola Superior de Tecnologia do Instituto Politécnico do Cávado e Ave (IPCA).
Com vista a demonstrar os conhecimentos e competências adquiridos ao longo da referida unidade curricular, foi solicitado um projeto de Avaliação em duas fases, na primeira fase – Listas Ligadas e na segunda fase – Grafos, neste relatório vou documentar a segunda fase.
Nesta segunda fase, pretende-se aplicar conceitos avançados de teoria dos grafos e programação em C para resolver um problema computacional com grau de complexidade maior, relacionando estruturas de dados, algoritmos de procura e técnicas de otimização. O objetivo é desenvolver uma solução capaz de calcular o somatório máximo possível de inteiros a partir de uma matriz de inteiros de dimensões arbitrárias, considerando regras específicas de conexão entre os inteiros.

2. Funcionalidades a Implementar

1. Definir uma estrutura de dados GR para representar um grafo. Esta estrutura deve ser capaz de representar grafos dirigidos e deve suportar um número variável de vértices. A implementação deve incluir funções básicas para criação do grafo, adição e remoção de vértices e arestas;

2. Após definir a estrutura de dados GR, pretende-se modelar o problema utilizando grafos. Cada elemento da matriz de inteiros será representado por um vértice no grafo. As arestas entre vértices devem representar a possibilidade de somar dois elementos adjacentes na matriz, sob uma regra de conexão específica que poderá ser configurada pelo utilizador (por exemplo, apenas elementos na mesma linha ou coluna, não permitindo diagonais, ou qualquer outra regra);

3. Carregamento para a estrutura de dados da alínea anterior GR dos dados de uma matriz de inteiros constante num ficheiro de texto. A operação deverá considerar matrizes de inteiros com qualquer dimensão, sendo os valores separados por vírgulas. A título de exemplo, o ficheiro de texto deverá respeitar o formato seguinte:

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

4. Implementar operações de manipulação de grafos, incluindo procura em profundidade ou em largura, para identificar todos os caminhos possíveis que atendem às regras de conexão definidas. Desenvolver também uma função para calcular a soma dos valores dos vértices num dado caminho;

5. Utilizar as estruturas e algoritmos desenvolvidos para encontrar o caminho que proporciona a maior soma possível dos inteiros na estrutura GR, seguindo a regra de conexão estabelecida. O programa deve fornecer tanto a soma máxima quanto o caminho (ou caminhos, se existirem múltiplos caminhos com a mesma soma máxima) que resulta nessa soma;

3. Implementação das funcionalidades

Por uma questão de organização e orientação dividi a questão um em várias questões.

1. **Criação de Bibliotecas**

   Foi realizado num projeto à parte, a implementação das funções essenciais para Criar o Grafo e Carregar o Grafo.

2. **Estrutura para a criação do Grafo**

   Para a criação do Grafo criamos a seguinte estrutura:

   Criação do vértice em memoria.

```
Vertice* createVertice(int cod, char* nome) {
    Vertice* novo = (Vertice*)malloc(sizeof(Vertice));
    if (novo == NULL) return NULL;
    novo->cod = cod;
    strcpy(novo->nome, nome);
    novo->visitado = false;
    novo->next = NULL;
    novo->adjacentes = NULL;
    return novo;
      }
```

Nesta função aceita dois parâmetros de entrada o código do vértice e o nome do vértice,,na primeira linha e criado espaço em memoria com o tamanho da struct vertice.
E verificado se o vertice em memoria foi criado.
Insere o código de entrada na variável da struct cod
Usa-se o strcpy para copiar os char todos para a variável nome da struct
Atribuiçao do NULL a visitado e next na struct
E returna o novo vertice
Funcao auxiliar na função LoadGraph.
Depois criamos a função para criar uma adjacência em memoria conf

```
Adj* createAdj(int cod, float weight) {
    Adj* novo = (Adj*)malloc(sizeof(Adj));
    if (novo == NULL) return NULL;
    novo->cod = cod;
    novo->dist = weight;
    novo->next = NULL;
    return novo;
      }
```

Fazemos o mesmo para Adjacencia.

Depois criamos a função para inserir vértice, conforme podemos observar

```
Vertice* InsereVertice(Vertice* h, Vertice* novo, bool* res) {
    if (h == NULL) {
        *res = true;
        return novo;
    }
    Vertice* current = h;
    while (current->next != NULL) {
```

```
        current = current->next;
    }
    current->next = novo;
    *res = true;
    return h;
      }
```

Nesta função aceita três parâmetros de entrada.

Ele verica se o vertice h é o primeiro em memoria e se for retorna um novo.

Se não for um vertice novo faz um loop até chegar ao next null e depois insere o vertice novo no next do vertice anterior e returna o novo vertice

3. Carregamento para a estrutura de dados da alínea anterior GR dos dados de uma matriz de inteiros constante num ficheiro de texto.

Para o carregamento do Grafo usamos uma função LoadGraph conforme podemos verificar.

```
Vertice* LoadGraph(Vertice* h, const char* fileName, bool* res) {
    *res = false;
    FILE* fp = fopen(fileName, "r");
    if (fp == NULL) {
        perror("Failed to open file");
        return NULL;
    }

    char line[256];
    while (fgets(line, sizeof(line), fp)) {
        char* token = strtok(line, ",");
        if (token == NULL) continue;

        int cod = atoi(token);

        token = strtok(NULL, ",");
        if (token == NULL) continue;

        char nome[MAX_NAME_LENGTH];
        strncpy(nome, token, MAX_NAME_LENGTH - 1);
        nome[MAX_NAME_LENGTH - 1] = '\0';

        Vertice* novo = createVertice(cod, nome);
        h = InsereVertice(h, novo, res);

        token = strtok(NULL, ",");
        while (token != NULL) {
            char* adjToken = strtok(token, ":");
            if (adjToken == NULL) break;

            int adjCod = atoi(adjToken);

            adjToken = strtok(NULL, ":");
```

```
            if (adjToken == NULL) break;

            float dist = atof(adjToken);

            Adj* adj = createAdj(adjCod, dist);
            addAdj(novo, adj);

            token = strtok(NULL, ",");
        }

    }
    fclose(fp);
    *res = true;
    return h;
    }
```

Esta função aceita três parâmetro de entrada que são o vértice, o nome do ficheiro e o resultado.

A função faz uma verificação para ver se o ficheiro existe.

Definimos a linha com 256 caracteres

Se o ficheiro existir ele entra num loop com as condições de ler a linha do ficheiro com o tamanho máximo de 256

Usamos o método strtok para ler a linha até a primeira linha e guarda no token char

Converte o token em int para guardar o cod na struct vértice

Faz o mesmo processo para ler ate a, para guardar o nome do vértice

Cria o vértice em memoria com as informações de cima

Depois insere o vértice criado na struct

Agora chega a vez de adicionar as adj, o algoritmo foi pensado da mesma forma.

Le o token até a primeira virgula

Entra em loop até o token for nulo:

Dentro do loop converte as informações em integer e guarda a variável adjcod

Lê o até o primeiro: para verificar o peso da adj

Adciona as adj nas funções auxiliares

4. Implementar operações de manipulação de grafos, incluindo procura em profundidade ou em largura, para identificar todos os caminhos possíveis

Para a manipulação de grafo usamos duas funções: a DepthFirstSearchRec BreadthFirstSearch.

```
bool DepthFirstSearchRec(Vertice* g, int origem, int dest)
{
    int j;
    if (origem < 1) return false;
    if (origem == dest) return true;

    Vertice* aux = ProcuraVerticeCod(g, origem);
    aux->visitado = true;

    Adj* adj = aux->adjacentes;
    while (adj) {
        Vertice* aux = ProcuraVerticeCod(g, adj->cod);
        if (aux->visitado == false)
        {
            bool existe = DepthFirstSearchRec(g, adj->cod, dest);
            return existe;
        }
        else
            aux = aux->next;
    }
    return true;
      }
```

A função DepthFirstSearchRec aceita três parâmetro de entrada que são o vértice a origem e o destino.
Primeiro verificamos se a origem, é menor que um se dor ele retorna falso.
Depois verificamos se a origem é igual ao destino se forem iguais ele retorna verdadeiro.
A seguir ele procura o vértice pelo código e marca o vértice como visitado.
No vértice guarda na variável adj as adjacências daquele vertice
Entra no loop ate acabar as adj do vértice, faz a procura do vértice e verifica se o vertice já foi visitado se não foi vistado entra na função novamente com o método da adj-cod na origem, e o if(origem== dest) e onde verifica se o vértice tem adj daquele dist ,porque na entrada origem e onde percorre as adj
Se não conseguir ficar true no if falta fora do loop porque já não existe adj para verificar e retorna false porque não existe caminhos entre os dois vértices de entrada.

A BFS é um algoritmo de pesquisa que explora todos os vértices de um nível antes de passar para o próximo, utilizando uma fila para gerenciar os vértices a serem visitados.

```
bool BreadthFirstSearch(Vertice* listaVertice, int verticeOrigem, int
destino) {
    int fila[300];

    if (verticeOrigem == destino) return true;
    bool res = false;

    int primeiro = 0;
    int ultimo = primeiro;
    int origem;

    Vertice* auxOrigem = listaVertice;
    Vertice* auxVertice = auxOrigem;

    while (auxOrigem != NULL && auxOrigem->cod != verticeOrigem) {
        auxOrigem = auxOrigem->next;
    }
    if (auxOrigem == NULL) return false;

    res = inserirFila(fila, &primeiro, auxOrigem->cod);

    if (res == false) return false;

    auxOrigem->visitado = true;

    while (verificaFilaVazia(&primeiro, &ultimo) == false)
    {
        res = removerFila(fila, &ultimo, &origem);
        if (res == false) return false;

        if (origem == destino) return true;

        Vertice* auxOrigem = listaVertice;

        while (auxOrigem != NULL && auxOrigem->cod != origem) {
            auxOrigem = auxOrigem->next;
        }
        if (auxOrigem == NULL) return false;

        Adj* auxAresta = auxOrigem->adjacentes;

        while (auxAresta != NULL)
        {
            Vertice* auxOrigem = listaVertice;

            while (auxOrigem != NULL && auxOrigem->cod != auxAresta->cod) {
                auxOrigem = auxOrigem->next;
            }

            if (auxOrigem == NULL) return false;

            if (auxOrigem->visitado == false) {
                res = inserirFila(fila, &primeiro, auxAresta->cod);
                if (res == false) return false;
                auxOrigem->visitado = true;
            }
```

```
            auxAresta = auxAresta->next;

        }
    }
    return false;
}
```

A fila fila é usada para armazenar os vértices que serão visitados.
Se o vértice de origem for igual ao de destino, o caminho existe e a função retorna true.
res: variável auxiliar para indicar sucesso ou falha em operações.
primeiro e ultimo: índices para controlar a fila.
origem: variável auxiliar para armazenar o vértice removido da fila.
A função procura o vértice de origem na lista de vértices.
Explorando os Vértices em Largura:

Utiliza um loop enquanto a fila não estiver vazia.
Remove o vértice da frente da fila e verifica se é o destino.
Percorre as adjacências do vértice atual, inserindo na fila os vértices adjacentes não visitados.
Marca os vértices adjacentes como visitados.
Se encontrar o destino durante a exploração, retorna verdadeiro.
Caso contrário, continua a exploração até esgotar todos os vértices na fila.
auxAresta = auxAresta->next; //avanca para a proxima arresta
Se a função chegar até aqui, significa que não foi encontrado um caminho do vértice de origem ao destino, então retorna false.

Diferença entre BFS e a Função BreadthFirstSearch

A principal diferença entre a BFS e a função BreadthFirstSearch é que a BFS é um algoritmo genérico para busca em largura em grafos, enquanto a função BreadthFirstSearch é uma implementação específica desse algoritmo para verificar se existe um caminho entre um vértice de origem e um vértice de destino em um grafo representado por uma lista de adjacências.

Enquanto a BFS pode ser aplicada em diferentes contextos e problemas, a função BreadthFirstSearch tem um propósito específico e é otimizada para verificar a existência de um caminho em um grafo.

Espero que esta explicação ajude a compreender tanto a função BreadthFirstSearch quanto o algoritmo de busca em largura (BFS) em geral.

3. Conclusão

Nesta segunda fase do projeto, aplicámos conceitos avançados da teoria de grafos e programação em C para resolver um problema computacional complexo. O objetivo era desenvolver uma solução que calculasse o somatório máximo possível de inteiros a partir de uma matriz de dimensões arbitrárias, respeitando regras específicas de conexão entre os inteiros.
Para alcançar esse objetivo, implementámos as seguintes funcionalidades:

1. **Definição da Estrutura de Dados GR:** Criámos uma estrutura de dados para representar grafos dirigidos com um número variável de vértices. Esta estrutura inclui funções básicas para criação do grafo, adição e remoção de vértices e arestas.
2. **Modelagem do Problema Utilizando Grafos:** Cada elemento da matriz de inteiros foi representado por um vértice no grafo. As arestas entre vértices representam a possibilidade de somar dois elementos adjacentes na matriz, conforme uma regra de conexão configurável pelo utilizador (por exemplo, somas permitidas apenas entre elementos na mesma linha ou coluna).
3. **Carregamento dos Dados:** Implementámos a funcionalidade de carregar dados de uma matriz de inteiros a partir de um ficheiro de texto para a estrutura de dados GR. A operação é capaz de lidar com matrizes de qualquer dimensão, com valores separados por vírgulas, como ilustrado no exemplo fornecido.
4. **Manipulação e Procura de Grafos:** Desenvolvemos operações de manipulação do grafo, incluindo busca em profundidade (DFS) ou em largura (BFS), para identificar todos os caminhos possíveis que atendem às regras de conexão. Também foi criada uma função para calcular a soma dos valores dos vértices num dado caminho.
5. **Otimização para Encontrar a Soma Máxima:** Utilizámos as estruturas e algoritmos desenvolvidos para encontrar o caminho que proporciona a maior soma possível dos inteiros na estrutura GR, seguindo a regra de conexão estabelecida. O programa fornece tanto a soma máxima quanto o caminho (ou caminhos) que resulta nessa soma.

Com esta abordagem, conseguimos não apenas resolver o problema proposto, mas também desenvolver uma solução robusta e flexível, capaz de lidar com diferentes configurações e dimensões de matrizes. Isto demonstra a aplicação prática de conceitos teóricos de grafos e técnicas avançadas de programação para resolver problemas computacionais complexos.