Trilhas de Aprendizagem com Grafos

Murielly Oliveira Nascimento

Universidade Federal de Uberlândia
Faculdade de Computação
Trabalho de Algoritmos e Estrutura de Dados 2.

Uberlândia
2022

Murielly Oliveira Nascimento

Trilhas de Aprendizagem com Grafos

Trabalho apresentado e desenvolvido durante a matéria de Algoritmos e Estrutura de Dados 2 no ano de 2022/1 sobre o tema Grafos.

Área de concentração: Sistemas de Informação

Uberlândia
2022

Resumo

Seja a trilha de aprendizagem de um aluno em uma disciplina no sistema Moodle representada em um grafo direcionado. A disciplina é Estrutura de Dados e cada vértice representa um recurso (URL, arquivo, tarefa, pasta etc) definido no âmbito da disciplina. As arestas contém um valor inteiro, que significa a quantidade de vezes que a ligação aconteceu.

O objetivo deste trabalho é projetar e desenvolver uma biblioteca para manipular grafos deste tipo (direcionado e ponderado), com funções para:

1. Criação do grafo, com inserção/remoção de vértices e arestas. Os vértices podem ser estruturas ou podem ser armazenados em vetor e referenciados a partir de um número no grafo. Ainda, pode-se usar matriz ou lista de adjacências.

2. Busca do vértice de maior grau, que, para a trilha, representa um recurso com peso importante no fluxo.

3. Dados dois recursos (vértices), verificar se existe caminho entre os mesmos.

4. A partir de um vértice, encontrar o menor caminho para os outros vértices a ele conectados.

5. Usando busca em profundidade, encontrar recursos fortemente conectados (Algoritmo).

6. Impressão do grafo.

Palavras-chave: Grafos.

Capítulo 1

Solução

Repositório: <https://replit.com/@Murielly/Trabalho-1#main.c>

Foi usada a linguagem C para a solução do problema. O código e é divido em arquivos para as TADs e para a função main. A estrutura APRENDIZADO é armazenada da seguinte forma no arquivo Aprendizado.h.

```
#ifndef __APRENDIZADO_H_INCLUDED__
#define __APRENDIZADO_H_INCLUDED__

#define MAX 100

typedef struct
{
    char nome[MAX];
    char tipo[MAX];
    char acao[MAX];
} APRENDIZADO;

#endif
```

Como definido no enunciado, os dados são recebidos do arquivo dados.txt e o TAD File é responsável por tratá-los e armazená-los num vetor da estrutura APRENDIZADO.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <math.h>

#include "TAD_File.h"

APRENDIZADO *open_txt(char *nome, int *tamanho)
{
    // O arquivo txt ja deve existir e portanto faremos apenas uma
   leitura
    FILE *dados = fopen(nome, "r");
```

```
13
14      // Descobrimos quantas linhas e colunas o arquivo tem
15      char caracter = '\0';
16      int i = 0, j = 0;
17
18      while (caracter != EOF)
19      {
20          caracter = fgetc(dados);
21          if (caracter == '\n')
22              i++;
23      }
24
25      // Voltamos o ponteiro para o inicio
26      rewind(dados);
27
28      // Linhas eh a quantidade de '\n' no arquivo
29      int linhas = i;
30      *tamanho = (linhas + 1) / 3;
31
32      // Criamos o vetor com os dados de aprendizagem
33      APRENDIZADO *vetor = (APRENDIZADO *)malloc((*tamanho) * sizeof(
    APRENDIZADO *));
34
35      if (vetor == NULL)
36          return NULL;
37
38      char aux[MAX] = {'\0'};
39
40      // Lemos os dados do arquivo e armazenamos no vetor
41      for (int i = 0; i < (*tamanho); i++)
42      {
43          fgets(aux, MAX, dados);
44          strcpy(vetor[i].nome, aux);
45
46          fgets(aux, MAX, dados);
47          strcpy(vetor[i].tipo, aux);
48
49          fgets(aux, MAX, dados);
50          strcpy(vetor[i].acao, aux);
51      }
52
53      // Fechamos o arquivo e liberamos a estrutura que lida com ele
54      fclose(dados);
55
56      // A funcao retorna o vetor preenchido com dados
57      return vetor;
58  }
```

A biblioteca de funções desse TAD foi especificada da seguinte forma.

```
#include "Aprendizado.h"

APRENDIZADO *open_txt(char *nome, int *tamanho);
```

O Grafo dessa estrutura segue os padrões definidos durantes as aulas do curso. Apenas duas mudanças são feitas no uso de Grafo como Matriz, a adição do atributo tamanho e o ponteiro para a estrutura Aprendizado. Note que assim não gastamos com a passagem das informações de uma estrutura para outra.

```
#include "Aprendizado.h"

typedef struct
{
    int peso;
    APRENDIZADO *aprendizado;
} VERTICE;

struct grafo
{
    int NumVertices;
    int NumArestas;
    int tamanho;
    VERTICE **Mat;
};
typedef struct grafo TipoGrafo;

TipoGrafo *CriarGrafo(int NVertices);

int inserirVertice(TipoGrafo *G, APRENDIZADO *dados);
int removeVertice(TipoGrafo *G, int posicao);

int inserirAresta(TipoGrafo *G, int v1, int v2, int peso);
int retirarAresta(TipoGrafo *Grafo, int v1, int v2);

void exibirGrafo(TipoGrafo *G);
void exibirMatriz(TipoGrafo *G);

TipoGrafo *liberarGrafo(TipoGrafo *G);

int *caminhoMaisCurto(TipoGrafo *G, int origem);
void caminho(TipoGrafo *G, int origem, int destino);
void exibeCaminho(TipoGrafo *G, int origem);

int grauVertice(TipoGrafo *G, int vertice);
int maiorGrau(TipoGrafo *G);

void BuscaEmProfundidade(TipoGrafo *G, int origem);
```

```
39 void conexoesFortes(TipoGrafo *G);
```

As função para criação do Grafo aloca dinamicamente a estrutura do Grafo. Em seguida a matriz para armazenamento dos dados também é alocada. O tamanho do Grafo é o valor definido nos parâmetros da função.

```
1  // CRIA GRAFO
2  TipoGrafo *CriarGrafo(int NVertices)
3  {
4      int i = 0, k = 0;
5      TipoGrafo *Grafo;
6
7      // Se o numero de vertices eh invalido
8      if (NVertices <= 0)
9          return NULL;
10
11     // Aloca Grafo
12     Grafo = (TipoGrafo *)malloc(sizeof(TipoGrafo *));
13     if (Grafo == NULL)
14         return NULL;
15
16     // Aloca matriz de vertices
17     Grafo->Mat = (VERTICE **)malloc(NVertices * sizeof(VERTICE *));
18     if (Grafo->Mat == NULL)
19     {
20         free(Grafo);
21         return NULL;
22     }
23
24     // Aloca arestas
25     for (i = 0; i < NVertices; i++)
26     {
27         Grafo->Mat[i] = (VERTICE *)calloc(NVertices, sizeof(VERTICE));
28         // Se a alocacao foi mal sucedida;
29         if (Grafo->Mat[i] == NULL)
30         {
31             for (k = 0; k < i; k++)
32                 free(Grafo->Mat[k]);
33             free(Grafo);
34             return NULL;
35         }
36     }
37     Grafo->NumVertices = 0;
38     Grafo->NumArestas = 0;
39     Grafo->tamanho = NVertices;
40     return Grafo;
41 }
```

Para inserir um novo vértice é preciso levar em conta se o número de vértices excedeu o tamanho alocado para o Grafo. Se sim será necessário realocá-lo. Caso não, basta fazer o último vértice da estrutura apontar para a variável dados, passada por parâmetro, e adicionar 1 ao número de vértices.

Se a primeira situação ocorrer precisamos primeiro salvar as informações da matriz em um vetor separado. A função realloc, funciona somente sobre estruturas que foram alocadas com malloc e calloc. Sendo assim, não podemos usá-la para realocar as linhas e colunas da matriz. Daí a necessidade de salvar os dados dos vértices numa estrutura a parte.

Ao final, determinamos o novo tamanho do Grafo. Por questões práticas o incremento no tamanho é de um em um. Isso pode ser alterado conforme a necessidade do programador. Perceba, também, que as informações de APRENDIZADO são inseridas na posição M(i,i), ou seja, coluna igual a linha. Isso é feito para evitar conflitos ao inserirmos arestas.

```
1  // INSERE VERTICE
2  int inserirVertice(TipoGrafo *G, APRENDIZADO *dados)
3  {
4      // Grafo nao existe
5      if (G == NULL)
6      {
7          printf("Grafo nao existe\n");
8          return -1;
9      }
10     // Se o grafo esta cheio
11     if (G->NumVertices == G->tamanho)
12     {
13         // Novo tamanho do Grafo
14         int novoTam = G->tamanho + 1;
15
16         // Salvo as informacoes antigas no vetor temp;
17         VERTICE temp[100][100] = {0};
18         for (int i = 0; i < G->NumVertices; i++)
19         {
20             for (int k = 0; k < G->NumVertices; k++)
21             {
22                 temp[i][k].aprendizado = G->Mat[i][k].aprendizado;
23                 temp[i][k].peso = G->Mat[i][k].peso;
24             }
25         }
26
27         VERTICE **aux;
28
29         // Realoca vertices;
30         aux = (VERTICE **)realloc(G->Mat, novoTam * sizeof(VERTICE *));
31         if (aux == NULL)
```

```
32          {
33              free(aux);
34              return -1;
35          }
36          // O grafo antigo aponta para o novo aux
37          G->Mat = aux;
38
39          // Realoca arestas
40          for (int i = 0; i < novoTam; i++)
41          {
42              aux[i] = (VERTICE *)calloc(novoTam, sizeof(VERTICE *));
43              // Se a aloca  o foi mal sucedida;
44              if (G->Mat[i] == NULL)
45              {
46                  for (int k = 0; k < i; k++)
47                      free(G->Mat[k]);
48                  free(G);
49                  return -1;
50              }
51              // Se foi bem sucedida
52              else
53              {
54                  // Atribuimos as arestas
55                  for (int k = 0; k < G->NumVertices; k++)
56                      aux[i][k] = temp[i][k];
57              }
58          }
59
60          // Novo tamanho do Grafo
61          G->tamanho = novoTam;
62      }
63
64      // Inserimos o novo vertice na ultima posicao usada da matriz
65      int posicao = G->NumVertices;
66      G->Mat[posicao][posicao].aprendizado = dados;
67      G->NumVertices++;
68
69      return 0;
70  }
```

A remoção do vértice é mais simplificada. Dada uma posição é acessado o VERTICE(i,i), o seu peso passa a ser zero e o ponteiro para aprendizado dessa posição aponta,agora, para NULL. Por fim, é necessário apagar qualquer ligação que outros vértice possam ter com ele e as ligações que ele venha a ter também. Para o primeiro caso percorremos a linha dessa posição atribuindo 0 ao seu peso, e para a segunda percorremos a coluna realizando o mesmo procedimento.

```c
int removeVertice(TipoGrafo *G, int posicao)
{
    // Grafo nao existe
    if (G == NULL)
    {
        printf("Grafo n o existe\n");
        return -1;
    }

    // Se o vertice nao existe
    if (posicao < 0 || posicao > G->NumVertices)
        return -1;

    G->Mat[posicao][posicao].aprendizado = NULL;
    G->Mat[posicao][posicao].peso = 0;

    for (int i = 0; i < G->NumVertices; i++)
    {
        G->Mat[i][posicao].peso = 0;
        G->Mat[posicao][i].peso = 0;
    }
    return 0;
}
```

Para inserirmos arestas, dado um local de origem e de chegada, verificamos se o intervalo é possível ou não, e se já não está preenchido. Feito isso atribuímos um peso a essa posição e incrementamos o número de arestas.

```c
// INSERE ARESTA
int inserirAresta(TipoGrafo *G, int v1, int v2, int peso)
{
    // Grafo n o existe
    if (G == NULL)
    {
        printf("Grafo n o existe\n");
        return -1;
    }

    // Intervalo    invalido
    if (v1 < 0 || v1 >= G->NumVertices || v2 < 0 || v2 >= G->NumVertices
   )
    {
        printf("Intervalo inv lido\n");
        return -1;
    }

    if (G->Mat[v1][v2].peso == 0)
    {
```

```
        G->Mat[v1][v2].peso = peso;
        G->NumArestas++;
    }
    return 1;
}
```

O mesmo procedimento na inserção é repetido na remoção de arestas, com a diferença que o peso dessa ligação passa a ser zero e decrementamos o número de arestas do Grafo.

```
// REMOVE ARESTA
int retirarAresta(TipoGrafo *G, int v1, int v2)
{
    // Grafo n o existe
    if (G == NULL)
    {
        printf("Grafo n o existe\n");
        return -1;
    }

    // Intervalo    invalido
    if (v1 < 0 || v1 >= G->NumVertices || v2 < 0 || v2 >= G->NumVertices
    )
    {
        printf("Intervalo inv lido\n");
        return -1;
    }

    // Aresta n o existe
    if (G->Mat[v1][v2].peso == 0)
    {
        printf("Aresta n o existe\n");
        return 0;
    }

    // Remove aresta
    G->Mat[v1][v2].peso = 0;
    G->NumArestas--;
    return 1;
}
```

Há duas formas de exibir o Grafo, na primeira apresentamos um resumo dele, com os dados da estrutura APRENDIZADO sendo impressos. Note que dentro do primeiro "for"verificamos se o VERTICE(i,i) aponta para algum dado APRENDIZADO, se não, isso significa que aquele vértice não existe.

```
// EXIBE GRAFO
void exibirGrafo(TipoGrafo *G)
{
    int v = 0, w = 0;
```

```
    printf("\nGrafo - Resumo:\n");
    for (v = 0; v < G->NumVertices; ++v)
    {
        if (G->Mat[v][v].aprendizado != NULL)
        {
            printf("%d - %s", v, G->Mat[v][v].aprendizado->nome);
            for (w = 0; w < G->NumVertices; ++w)
                if (G->Mat[v][w].peso != 0)
                    printf("   Ligado a %s", G->Mat[w][w].aprendizado->
   nome);
            printf("\n");
        }
    }
}
```

Na segunda exibimos a matriz com os pesos. Os dados de aprendizado são ignorados em favor de mostrar as conexões e seus respectivos pesos. Caso um vértice tenha sido removido a sua linha e coluna não aparecerão na matriz.

```
// EXIBE MATRIZ
void exibirMatriz(TipoGrafo *G)
{
    printf("\nGrafo - Matriz:\n");

    // Printamos as colunas
    printf("     ");
    for (int j = 0; j < G->NumVertices; j++)
    {
        if (G->Mat[j][j].aprendizado != NULL)
            printf(" %d ", j);
    }
    printf("\n");

    // Printamos as linhas e os pesos
    for (int i = 0; i < G->NumVertices; ++i)
    {
        if (G->Mat[i][i].aprendizado != NULL)
        {
            printf("%d:   ", i);
            for (int j = 0; j < G->NumVertices; ++j)
                if (G->Mat[j][j].aprendizado != NULL)
                    printf(" %d ", G->Mat[i][j].peso);
            printf("\n");
        }
    }
}
```

Para liberar o Grafo levamos em conta o seu tamanho no lugar do número de vértices.

Começamos liberando o espaço alocado para as colunas da matriz, depois liberamos as linhas, por fim o próprio Grafo é liberado.

```
// LIBERA GRAFO
TipoGrafo *liberarGrafo(TipoGrafo *G)
{
    int i;
    if (G == NULL)
        return NULL;

    for (i = 0; i < G->tamanho; i++)
    {
        free(G->Mat[i]);
    }
    free(G->Mat);
    free(G);
    G = NULL;
    return G;
}
```

O menor caminho partindo de um vértice é calculado usando o algoritmo disponibilizado durante as aulas, tendo como modificação a consideração do peso de cada ligação no seu cálculo. A função caminhoMaisCurto() faz os devidos cálculos e retorna o ponteiro para o vetor com os valores dos caminhos. A função exibeCaminho() recebe esse ponteiro e imprime a solução.

```
// Caminho Mais Curto
int *caminhoMaisCurto(TipoGrafo *G, int origem)
{
    int i, v, w, minimo, posmin, S;
    int *M, *L;

    M = (int *)malloc(G->NumVertices * sizeof(int));
    L = (int *)malloc(G->NumVertices * sizeof(int));

    // preenchimento preliminar dos vetores
    for (i = 0; i < G->NumVertices; i++)
    {
        M[i] = 0;       // falso, v rtice n o visitado
        L[i] = 999999; // valor inicial para os custos
    }

    M[origem] = 1;
    for (v = 0; v < G->NumVertices; v++)
    {
        if (G->Mat[origem][v].peso != 0)
        {
            L[v] = G->Mat[origem][v].peso;
```

```
        }
    }

    for (i = 0; i < G->NumVertices; i++)
    {
        w = 0;
        minimo = 999999;
        for (v = 0; v < G->NumVertices; v++)
        {
            if (L[v] < minimo && M[v] == 0)
            {
                minimo = L[v];
                posmin = v;
                w = 1;
            }
        }
        if (w == 0)
            break;
        M[posmin] = 1;
        for (v = 0; v < G->NumVertices; v++)
        {
            if (G->Mat[posmin][v].peso != 0 && M[v] == 0)
            {
                S = L[posmin] + G->Mat[posmin][v].peso;
                if (S < L[v])
                    L[v] = S;
            }
        }
    }
    return L;
}

// EXIBE CAMINHO
void exibeCaminho(TipoGrafo *G, int origem)
{
    if (G == NULL)
        return;

    int *L = caminhoMaisCurto(G, origem);

    printf("\nMenor caminho partindo de %d: \n", origem);
    for (int v = 0; v < G->NumVertices; v++)
    {
        if (L[v] == 999999)
            printf("%d: - \n", v);
        else
            printf("%d: %d\n", v, L[v]);
```

```
    }
}
```

Para descobrir se há um caminho entre dois vértices usamos a função caminhoMaisCurto() para receber o vetor com os valores das passagens. Se o ponto de destino aparece nesse vetor então há um caminho entre o ponto de origem e destino.

```
// CAMINHO
void caminho(TipoGrafo *G, int origem, int destino)
{
    if (G == NULL)
        return;

    int *vetor = caminhoMaisCurto(G, origem);

    if (vetor[destino] != 999999)
        printf("Ha um caminho entre %d e %d\n", origem, destino);
    else
        printf("Nao ha um caminho entre %d e %d\n", origem, destino);

    return;
}
```

O grau de um vértice é a soma dos pesos de suas colunas na horizontal e vertical. Ou seja, as arestas que saem e chegam nele.

```
// GRAU VERTICE
int grauVertice(TipoGrafo *G, int vertice)
{
    if (G == NULL)
        return 0;

    int grau = 0;
    for (int i = 0; i < G->NumVertices; i++)
    {
        grau += G->Mat[vertice][i].peso + G->Mat[i][vertice].peso;
    }

    return grau;
}
```

O vértice de maior grau é calculado a partir da função grauVertice().

```
// MAIOR GRAU
int maiorGrau(TipoGrafo *G)
{
    if (G == NULL)
        return 0;

    int maior = 0, cont = 0;
```

```
    for (int i = 0; i < G->NumVertices; i++)
    {
        int valor = grauVertice(G, i);
        if (valor > maior)
        {
            cont = i;
            maior = valor;
        }
    }
    printf("O vertice de maior grau eh %d com peso: %d\n", cont, maior);

    return cont;
}
```

O algoritmo de busca em profundidade é recursivo e se vale de uma implementação simplificada.

```
// BUSCA EM PROFUNDIDADE
void dfs(int origem, bool *visitados, TipoGrafo *G)
{
    printf("%d ", origem);
    visitados[origem] = true;

    for (int i = 0; i < G->NumVertices; i++)
    {
        if (G->Mat[origem][i].peso != 0 && (!visitados[i]))
        {
            dfs(i, visitados, G);
        }
    }
}

void BuscaEmProfundidade(TipoGrafo *G, int origem)
{
    bool *visitados;
    visitados = (bool *)malloc(G->NumVertices * sizeof(bool));
    if (visitados == NULL)
        return;

    int cont = 0;
    dfs(origem, visitados, G);
}
```

Para calcular os componentes fortemente conectados fazemos uso do TAD Pilha.

```
#include <stdio.h>
#include <stdlib.h>
#include "TAD_Pilha.h"

```

```
5  //INICIA PILHA
6  void IniciaPilha(TipoPilha *pilha)
7  {
8      pilha->topo = NULL;
9      pilha->tamanho = 0;
10 }
11
12 //PILHA VAZIA
13 int VaziaPilha(TipoPilha *pilha)
14 {
15     //Se a pilha est  vazia
16     return (pilha->topo == NULL);
17 }
18
19 //TOPO
20 int Topo(TipoPilha *pilha)
21 {
22     TipoNo *aux;
23     if (pilha == NULL)
24         return -1;
25
26     //O valor no topo da pilha
27     return pilha->topo->valor;
28 }
29
30 //EMPILHA
31 void Empilha(int x, TipoPilha *pilha)
32 {
33     TipoNo *aux;
34     if (pilha == NULL)
35         return;
36
37     aux = (TipoNo *)malloc(sizeof(TipoNo));
38     //Se a aloca  o deu errado
39     if (aux == NULL)
40         return;
41
42     //O novo N  aponta para o primeriro N  da pilha
43     aux->valor = x;
44     aux->prox = pilha->topo;
45
46     //O topo da pilha agora   o primeiro N
47     pilha->topo = aux;
48     pilha->tamanho++;
49 }
50
51 //DESEMPILHA
```

```
52 int Desempilha(TipoPilha *pilha)
53 {
54     int v = 0;
55     TipoNo *no;
56 
57     if (pilha == NULL)
58         return -1;
59 
60     no = pilha->topo;
61     v = no->valor;
62 
63     //O topo passa a ser o pr ximo  N
64     pilha->topo = no->prox;
65     pilha->tamanho--;
66     free(no);
67 
68     //Retorno o valor desempilhado
69     return v;
70 }
```

Cuja biblioteca é como segue.

```
1 #include "No.h"
2 
3 typedef struct Pilha
4 {
5     TipoNo *topo;
6     int tamanho;
7 } TipoPilha;
8 
9 void IniciaPilha(TipoPilha *pilha);
10 int VaziaPilha(TipoPilha *pilha);
11 void Empilha(int x, TipoPilha *pilha);
12 int Desempilha(TipoPilha *pilha);
13 int Topo(TipoPilha *pilha);
```

E faz uso da estrutura No

```
1 #ifndef __NO_H_INCLUDED__
2 #define __NO_H_INCLUDED__
3 
4 typedef struct No
5 {
6     int valor;
7     struct No *prox;
8 } TipoNo;
9 
10 #endif
```

Por fim o cálculo dos componentes fortemente conectados se dá da seguinte forma. Primeiro os vértices são inseridos numa pilha.

```
void inserePilha(TipoGrafo *G, bool *visitados, TipoPilha *pilha, int
    vertice)
{
    visitados[vertice] = true;
    int i = 0;

    for (int i = 0; i < G->NumVertices; i++)
    {
        int posicao = G->Mat[vertice][i].peso;
        if ((!visitados[i]) && (posicao != 0))
        {
            inserePilha(G, visitados, pilha, posicao);
        }
    }
    Empilha(vertice, pilha);
}
```

Para o cálculo da transposta fazemos uso de um vetor auxiliar que guarda o peso das arestas para depois passá-las da forma correta ao Grafo. Para desfazer esse procedimento basta chamar a função transposta, o que fazemos depois de calcularmos as conexões fortes.

```
{
    int tam = G->NumVertices;

    int aux[tam][tam];

    for (int i = 0; i < tam; i++)
    {
        for (int j = 0; j < tam; j++)
        {
            aux[j][i] = G->Mat[i][j].peso;
        }
    }
    for (int i = 0; i < tam; i++)
    {
        for (int j = 0; j < tam; j++)
        {
            G->Mat[i][j].peso = aux[i][j];
        }
    }
}
```

Por fim chamamos a função dfs para calcular os componentes fortemente conectados. Note que ao final dela chamamos a função transposta novamente e liberamos a pilha.

```
void conexoesFortes(TipoGrafo *G)
```

```
{
    int tam = G->NumVertices;
    bool visitados[tam];

    TipoPilha *pilha;
    // Iniciar a pilha e inserir o v rtice de origem
    pilha = (TipoPilha *)malloc(sizeof(TipoPilha));
    if (pilha == NULL)
        return;
    IniciaPilha(pilha);

    for (int i = 0; i < tam; i++)
        visitados[i] = false;

    for (int i = 0; i < tam; i++)
    {
        if (visitados[i] == false)
        {
            inserePilha(G, visitados, pilha, i);
        }
    }
    int count = 1;
    for (int i = 0; i < tam; i++)
        visitados[i] = false;

    transposta(G);

    while (!VaziaPilha(pilha))
    {
        int v = Desempilha(pilha);

        if (visitados[v] == false)
        {
            printf("Componentes fortemente conectados %d: \n", count++);
            dfs(v, visitados, G);
            printf("\n");
        }
    }
    transposta(G);

    free(pilha);
}
```

No arquivo main.c temos as funções criarMenu() e criarSubMenu() que exibem as opções ao usuário.

```
void criarSubMenu()
{
```

```
    printf("\nMENU\n");
    printf("1 - Exibe Grafo\n");
    printf("2 - Remove Vertice\n");
    printf("3 - Insere Aresta\n");
    printf("4 - Remove Aresta\n");
    printf("5 - Exibe Vertice de maior Grau\n");
    printf("6 - Existe caminho\n");
    printf("7 - Menor Caminho a partir de um vertice\n");
    printf("8 - Componentes Fortemente Conectados\n");
    printf("9 - Encerrar\n");
    printf("Digite uma opcao: ");
}

void criarMenu()
{
    printf("\nBEM VINDO AO GRAFO APRENDIZADO\n");
    printf("1 - Iniciar\n");
    printf("2 - Sair\n");
    printf("Digite uma opcao: ");
}
```

A função opcoes() cria um loop sobre o qual o usuário pode manipular o Grafo, chamando cada uma de suas funções.

```
void opcoes(TipoGrafo *g)
{
    int opcao = 0;
    scanf("%d", &opcao);

    while (opcao != 9)
    {
        if (opcao == 1)
        {
            int aux = 0;
            printf("1 - Exibir Grafo\n");
            printf("2 - Exibir Matriz\n");
            printf("Digite uma op   o: ");
            scanf("%d", &opcao);

            if (opcao == 1)
                exibirGrafo(g);
            else
            {
                exibirMatriz(g);
            }
        }
        else if (opcao == 2)
        {
```

```
25              int pos = 0;
26              printf("Digite a posi   o: ");
27              scanf("%d", &pos);
28
29              removeVertice(g, pos);
30          }
31          else if (opcao == 3)
32          {
33              int origem = 0, destino = 0, peso = 0;
34
35              printf("Digite a origem: ");
36              scanf("%d", &origem);
37
38              printf("Digite o destino: ");
39              scanf("%d", &destino);
40
41              printf("Digite o peso da liga   o: ");
42              scanf("%d", &peso);
43
44              inserirAresta(g, origem, destino, peso);
45          }
46          else if (opcao == 4)
47          {
48
49              int origem = 0, destino = 0;
50
51              printf("Digite a origem: ");
52              scanf("%d", &origem);
53
54              printf("Digite o destino: ");
55              scanf("%d", &destino);
56
57              retirarAresta(g, origem, destino);
58          }
59          else if (opcao == 5)
60          {
61              maiorGrau(g);
62          }
63          else if (opcao == 6)
64          {
65              int origem = 0, destino = 0;
66
67              printf("Digite a origem: ");
68              scanf("%d", &origem);
69
70              printf("Digite o destino: ");
71              scanf("%d", &destino);
```

```
72
73              caminho(g, origem, destino);
74          }
75          else if (opcao == 7)
76          {
77              int origem = 0;
78              printf("Digite o v rtice de origem: ");
79              scanf("%d", &origem);
80
81              exibeCaminho(g, origem);
82          }
83          else if (opcao == 8)
84          {
85              conexoesFortes(g);
86          }
87          else if (opcao == 9)
88          {
89
90              break;
91          }
92          else
93              printf("Op   o inv lida\n");
94
95          criarSubMenu();
96          scanf("%d", &opcao);
97      }
98  }
```

A função iniciar abre o arquivo dados.txt, onde estão as informações da estrutura APRENDIZADO, e as armazena num vetor. Em seguida o Grafo do tamanho desse vetor é criado e os vértices são inseridos. Quando o usuário digita a opção 9 a função opcao() é encerrada e o Grafo, juntamente com os dados, é liberado.

```
1  void iniciar()
2  {
3      // Dados s o passados para a estrutura APRENDIZADO
4      char nome[] = "dados.txt"; // nome do arquivo
5      int tamanho = 0;           // tamanho do arquivo
6
7      APRENDIZADO *dados = open_txt(nome, &tamanho);
8      APRENDIZADO *iterator = dados;
9
10     /* Criamos o grafo
11        Afim de poupar processamento optei por guardar um ponteiro
12        para a estrutura APRENDIZADO em cada v rtice do grafo.*/
13     TipoGrafo *g = CriarGrafo(tamanho);
14
15     // Insiro v rtices no grafo. Cada um deles aponta para um dado
```

```
    for (int i = 0; i < tamanho; i++)
    {
        inserirVertice(g, iterator);
        iterator++;
    }

    /*Chamo a funcao com as opcoes e o Menu*/
    opcoes(g);

    // Liberamos o grafo e a estrutura APRENDIZADO
    free(dados);
    liberarGrafo(g);
}
```

A função main somente exibe o Menu Principal, se o usuário escolher a opção 1 as demais funções são disparadas a partir da chamada da função iniciar().

```
int main(void)
{

    int opcao = 0;
    criarMenu();
    scanf("%d", &opcao);
    if (opcao == 1)
    {
        criarSubMenu();
        iniciar();
    }

    return 0;
}
```

Com isso finalizamos o programa para lidar com Trilhas de Aprendizagem usando Grafos.