

Relações Binárias

Lucas Paulo Martins Mariz

Matrícula: 2018055016

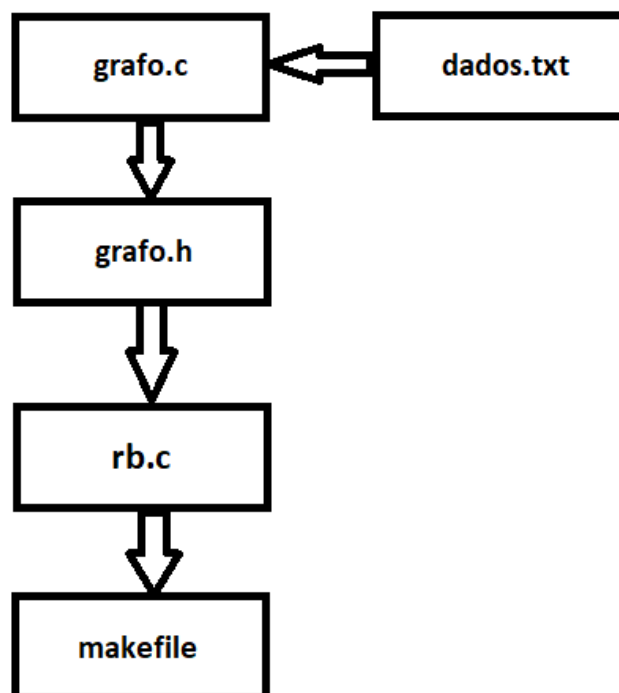
Ciência da Computação

“Com o advento de novas tecnologias bem como a constante evolução da mesma, é cada vez mais necessário que a programação esteja presente no dia a dia da população pois ela permite a solução de problemas de forma mais dinâmica e eficiente.”

Descrição Geral:

O desenvolvimento de uma estrutura na linguagem C na qual relações seriam armazenadas em um grafo se torna uma tarefa um tanto quanto complexa dado o fato de ser necessário o uso de ponteiros de ponteiros, arquivos, vetores dinâmicos, entre outros. Sendo assim, todo processo foi minuciosamente esquematizado para evitar que erros de lógica pudessem vir a surgir.

Primeiramente, a estrutura de arquivos seguiu o seguinte padrão:



A biblioteca “grafo.c” contém todas as funções do trabalho e recebe os dados contidos no arquivo “dados.txt”. Todo trabalho é compilado simplesmente digitando “make” no terminal devido ao arquivo “makefile”.

O arquivo “rb.c” possui uma estrutura bem simples, contendo apenas a inclusão de bibliotecas e a chamada da função “inicializar”, localizada na biblioteca de grafos.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Grafo.h"
int main(){
    inicializar();
    return 0;
}
```

O arquivo “grafo.h” contém a declaração de todas as funções contidas em “grafo.c” bem como a estrutura do grafo (foram desenvolvidas 22 funções no total).

```
//Declaração de funções:

typedef struct grafo Grafo;

//#1
Grafo* cria_Grafo();
//#2
void libera_Grafo(Grafo* gr);
//#3
int insereAresta(Grafo* gr, int orig, int dest);
//#4
void imprime_Grafo(Grafo *gr);
//#5
void eh_Reflexivo(Grafo *gr, int fecho);
//#6
void eh_Irreflexivo(Grafo *gr, int fecho);
//#7
int removeAresta(Grafo* gr, int orig, int dest);
//#8
int remove_Simetrico(Grafo* gr2);
```

E o arquivo “makefile” compila a biblioteca e o “rb.c”.

```
1 all:
2     gcc Grafo.c rb.c
```

Além disso, o trabalho é composto por um arquivo “leiam.txt” contendo as principais instruções para a execução do trabalho. Seu conteúdo está descrito abaixo:

//inicio

O projeto foi inteiramente desenvolvido em um ambiente linux, sendo assim, o melhor sistema para sua execução sem grandes problemas é o Linux (preferencialmente Ubuntu).

Para a execução é necessário instalar alguns pacotes relacionados a makefile, caso já não estejam instalados.

\$ sudo apt-get update

\$ sudo apt-get upgrade

\$ sudo apt-get install make

Após a instalação, basta digitar ir para o diretório do trabalho utilizando o comando 'cd' e digitar o comando "make" no terminal, o qual irá compilar o main e a biblioteca:

`$ make`

A biblioteca será automaticamente compilada e o programa irá gerar um arquivo executável "a.out", que deverá ser executado seguindo da seguinte forma:

`$./a.out`

A execução do projeto é independente e não necessita da interferência do usuário pois todos os dados são resgatados do arquivo "dados.txt".

Para modificar as informações do programa basta alterar o arquivo "dados.txt". Ele está formatado da seguinte forma:

- 1º Linha: 1º dígito-> número de elementos do grafo

Dígitos seguintes-> nome de cada elemento

- Linhas seguintes-> relacionamentos entre os elementos do grafo.

----> Arquivos presentes:

- rb.c : main do trabalho;

- Grafo.c : biblioteca com todas as funções para determinar as propriedades dos grafos;

- Grafo.h : linkagem das bibliotecas;

- makefile : arquivo make para compilar o rb.c e o Grafo.c;

- dados.txt : arquivo com a entrada do trabalho (número de elementos, elementos, relações);

- rb.pdf : documentação do trabalho.

//fim

- **Arquivo grafo.c e suas funções:**

De modo a deixar o trabalho mais organizado e, de certa forma, modularizado, optei por fazer todas as funções designadas a determinar propriedades das relações em um arquivo separado. Tal arquivo pode ser chamado como uma biblioteca, a qual é incluída através de seu arquivo ".h".

A estrutura principal do grafo consiste em alguns elementos e propriedades devidamente explicadas. As propriedades podem assumir os valores 1 ou 0 dependendo, respectivamente, da sua ocorrência ou não.

```

struct grafo{
    int nro_vertices;           //Número total de vertices que o grafo terá
    int n_ligacoes;           //Número total de ligações que o grafo fará
    int** arestas;             //Matriz de adjacências que armazena as ligações do grafo
    int* grau;                 //Vetor que representa a quantidade de ligações que cada vértice faz
    int elementos[50];         //Nomes dos vértices fornecidos pelo usuário
//elementos[50] : o número máximo de relacionamentos que o grafo pode ter é 50 (contando com ele mesmo)
/*
O programa funciona de forma transparente para o usuário. São inseridos os vértices 10, 13 e 15, por exemplo.
Internamente, se os vértices forem inseridos nessa ordem, eles serão armazenados no vetor elementos na posição
[0], [1] e [2] e serão tratados como elementos 0, 1 e 2 nos momentos de comparações. Contudo, na hora de printar
para o usuário, seus nomes são traduzidos pelos verdadeiros armazenados no vetor elementos.
*/
    int lig_aux1;
    int lig_aux2;
//Propriedades:
    int reflexiva;
    int irreflexiva;
    int simetrica;
    int anti_simetrica;
    int assimetrica;
    int transitiva;
};

```

Ao executar o programa, a função “inicializar” é chamada e sua estrutura consiste basicamente na chamada de outras funções que verificam determinadas propriedades do grafo. Obs: Alguns grafos foram criados nessa função por questão de ambiguidade dos dados, de forma a manter os valores constantes em um grafo principal enquanto os mesmos são alterados em outras estruturas.

```

void inicializar(){
    Grafo* gr = cria_Grafo();// inicialização do grafo
    Grafo* gr3 = cria_Grafo();
    Grafo* gr4 = cria_Grafo();
    system("clear");
    printf("\n\n");
    eh_Reflexivo(gr,0);
    eh_Irreflexivo(gr,0);
    eh_Simetrico(gr,0);
    eh_Anti_simetrico(gr,0);
    eh_Assimetrica(gr);
    eh_Transitivo(gr,0);
    relacao_Equivalencia(gr);
    relacao_Ordem_Parcial(gr);
    fecho_Reflexivo(gr);
    fecho_Simetrico(gr,gr3,gr4);
    fecho_Transitivo(gr);
    libera_Grafo(gr);
}

```

A seguir, será explicado o funcionamento das principais funções acima.

cria_Grafo: essa primeira função inicializa o grafo, fazendo as devidas alocações, e carrega os dados armazenados no arquivo para as variáveis do grafo, de forma a montar uma matriz de adjacências.

```
//-----Lendo dados do arquivo-----//
char url[]="dados.txt";
char ch[1];
FILE *arq;

arq = fopen(url, "r");
if(arq == NULL)
    printf("Erro, nao foi possivel abrir o arquivo\n");
else
    ch[0]=fgetc(arq);    //Pega o primeiro caractere do arquivo que representa o número de elementos

fclose(arq);
nro_vertices = atoi(ch);    //Transformação de char para inteiro
gr->nro_vertices = nro_vertices;
gr->grau = (int*) calloc(gr->nro_vertices,sizeof(int));

gr->arestas = (int**) malloc(gr->nro_vertices * sizeof(int*));
for(i=0; i<gr->nro_vertices; i++)
    gr->arestas[i] = (int*) malloc(50 * sizeof(int));
gr->n_ligacoes=0;
```

Caso esteja na primeira linha do arquivo, instancia-se o vetor de elementos, caso contrário, são inseridos os relacionamentos lidos até o final do arquivo:

```
while( (fgets(info, sizeof(info), arq))!=NULL ){
    if(controle){ //Se não estiver na primeira linha do arquivo temos as ligações
        //Lendo dados referentes aos relacionamentos feitos entre as arestas do grafo
        int n1,n2;
        char aux1[1], aux2[1];
        aux1[0] = info[0];
        aux2[0] = info[2];
        //Conversões de char para inteiro
        n1 = aux1[0] - '0';
        n2 = aux2[0] - '0';

        int arest = insereAresta(gr, n1, n2);
    }
    else{ //Se estivermos na primeira linha do arquivo temos os nomes dados as arestas
        int cont=0;
        for(i=2;i<strlen(info);i+=2){
            int nome;
            char aux[1];
            aux[0] = info[i];
            nome = aux[0] - '0';
            gr->elementos[cont] = nome;
            cont++;
        }
        controle++;
    }
}
```

O programa funciona de forma transparente para o usuário. São inseridos os elementos 10, 13 e 15, por exemplo. Internamente, se os vértices forem inseridos nessa ordem, eles serão armazenados no vetor “elementos” nas posições [0], [1] e [2] e serão tratados como elementos 0, 1 e 2 nos momentos de comparações e em todas as funcionalidades internas do programa. Contudo, na hora de transmitir para o usuário, seus nomes são “traduzidos” para aqueles que o usuário inseriu.

Tal “tradução” ocorre na função *imprime_Grafo()*, a qual imprime as relações inseridas no grafo de acordo com o arquivo de entrada de dados.

```

void imprime_Grafo(Grafo *gr){ //Função que imprime os relacionamentos realizados no grafo
    if(gr == NULL)
        return;

    int i, j;
    //Tradução anteriormente mencionada na qual imprimi-se o verdadeiro nome do vértice usando sua verdadeira posição no vetor
    for(i=0; i < gr->nro_vertices; i++){
        printf("%d: ", gr->elementos[i]);
        for(j=0; j < gr->grau[i]; j++){
            printf("%d, ", gr->elementos[gr->arestas[i][j]]);
        }
        printf("\n");
    }
}

```

eh_Reflexivo: essa função simplesmente verifica a propriedade reflexiva das relações, ou seja, se para todo elemento x existe um relacionamento do tipo (x,x) .

```

int i, j, verificado=0;
for(i=0; i < gr->nro_vertices; i++){
    for(j=0; j < gr->grau[i]; j++){
        if(i == gr->arestas[i][j]){ //Olhar se cada vértice se liga a ele mesmo
            verificado++;
        }
    }
}

if(verificado == gr->nro_vertices){
    printf("1. Reflexiva: V\n");
    gr->reflexiva = 1;
}

```

Foi criado um looping que passa por toda matriz de adjacência do grafo e fiscaliza se cada elemento se liga a ele próprio. Caso sim, é reflexivo, caso não, é necessário mostrar todos os elementos que não atendem a essa condição.

```

else{
    printf("1. Reflexiva: F\n");
    gr->reflexiva = 0;
    int entrou=0, cont=0;
    for(i=0; i < gr->nro_vertices; i++){
        for(j=0; j < gr->grau[i]; j++){
            if(i == gr->arestas[i][j]){
                //Caso não seja reflexivo, contar o número de vértices que não são reflexivos
                cont++;
            }
        }
        if(cont == gr->grau[i]){ //Caso nem todos os vértices sejam reflexivos, imprimir os que não são
            printf("(%d,%d); ", gr->elementos[i], gr->elementos[i]);
            entrou++;
        }
        cont=0;
    }
    if(entrou)
        printf("\n");
}

```

eh_Irreflexivo: esse função funciona de forma análoga a anterior, dado o fato que seu objetivo é basicamente o inverso de uma relação reflexiva. Sendo assim, a estrutura de código é basicamente a mesma, mudando apenas algumas condicionais.

removeAresta: função que, dado um conjunto de relações, remove o uma da matriz do grafo, diminuindo o número de relacionamentos e, conseqüentemente, o grau do vértice.

```
int removeAresta(Grafo* gr, int orig, int dest){ //Retirar um relacionamento
int i = 0;
while(i < gr->grau[orig] && gr->arestas[orig][i] != dest)
    i++;
if(i == gr->grau[orig]){//elemento nao encontrado
    return 0;
}
gr->grau[orig]--;
gr->arestas[orig][i] = gr->arestas[orig][gr->grau[orig]];
gr->n_ligacoes--;
return 1;
}
```

eh_Simetrico: função que, dado um conjunto de relações, verifica a propriedade de simetria do mesmo, ou seja, para todo relacionamento (x,y) deve existir um (y,x). Ocorre a criação de um grafo auxiliar de modo que caso dois elementos sejam simétricos os mesmos são removidos do grafo. Caso o grafo fique sem relacionamentos no final a propriedade simétrica pertence ao grafo, caso contrário, é necessário listar os elementos que não atendem a essa propriedade.

Todo esse processo de remoção de elementos é realizado pela função *remove_Simetrico()*.

```
int i, j, k, controle=0;
for(i=0; i < gr2->nro_vertices; i++){
    for(j=0; j < gr2->grau[i]; j++){
        for(k=0; k < gr2->grau[gr2->arestas[i][j]]; k++){
            if(i == gr2->arestas[gr2->arestas[i][j]][k]){
                if(gr2->arestas[i][j] != gr2->arestas[gr2->arestas[i][j]][k]){
                    int aux1=gr2->arestas[i][j];
                    int aux2=gr2->arestas[gr2->arestas[i][j]][k];
                    int y = removeAresta(gr2, gr2->arestas[i][j], gr2->arestas[gr2->arestas[i][j]][k]);
                    int x = removeAresta(gr2, aux2, aux1);
                    controle++;
                }
            }
            else{ //Caso seja um elemento reflexivo, não é necessario remover seu inverso
                int x = removeAresta(gr2, gr2->arestas[i][j], gr2->arestas[gr2->arestas[i][j]][k]);
                controle++;
            }
        }
    }
}
return controle;
```

A função *remove_Simetrico()* (trecho de código acima) faz um processo que deveria ser realizado de forma recursiva, entretanto, foi seguido um outro caminho mais simples, no qual a função é chamada em looping enquanto a variável “controle” não retornar com o valor 0 (quando não remover nenhuma ligação).

Seu funcionamento ocorre da seguinte forma: pega-se o primeiro elemento e é analisado a quem ele está relacionado e é buscado nos relacionamentos dos elementos ao qual ele está relacionado se os mesmos o possuem como relacionamento, e assim por diante. Exemplificando: $R = \{(1,1), (1,3), (1,2), (3,1)\}$

1: 1, 3, 2

2:

3: 1

O código pegaria os relacionamentos do elemento 1, por exemplo seu relacionamento com o elemento 3. A seguir, o programa verificaria os relacionamentos do elemento 3 para determinar se o mesmo possui um relacionamento com elemento 1, realizando o mesmo processo para todos elementos e relacionamentos.

Como explicado acima, caso o grafo contenha relacionamentos após a execução da função, é necessário listar os relacionamentos restantes, os quais não são simétricos.

```
else{ //Caso não seja simétrica, imprimir as ligações restantes no grafo
    printf("3. Simetrica: F\n");
    gr->simetrica = 0;
    int entrou=0;
    //imprime_Grafo(gr2);
    for(i=0; i < gr2->nro_vertices; i++){
        for(j=0; j < gr2->grau[i]; j++){
            printf("(%d,%d); ",gr2->elementos[gr2->arestas[i][j]],gr2->elementos[i]);
            entrou++;
        }
    }
    if(entrou)
        printf("\n");
}
libera_Grafo(gr2);
```

eh_Anti_simetrico: verifica se o conjunto de relações é anti-simetrico utilizando a lógica das funções *eh_simetrica()* e *eh_Reflexiva()*. Basicamente é verificado se para todo relacionamento (x,y) e (y,x), $x = y$. Caso a condição inicial não seja satisfeita, a propriedade é assumida por default.

eh_Assimetrica: verifica se o conjunto de relações é assimétrico, ou seja, é irreflexivo e anti-simétrico ao mesmo tempo.

```
void eh_Assimetrica(Grafo* gr){ //Verificar se o grafo é assimétrico
    if(gr->irreflexiva == 1 && gr->anti_simetrica == 1)
        printf("5. Assimetrica: V\n");
    else
        printf("5. Assimetrica: F\n");
}
```

eh_Transitivo: uma das funções mais complexas de ser implementada se refere a que determina a propriedade transitiva das relações, dado o fato que são realizadas verificações um nível mais profundo do que as realizadas na função *eh_Simetrico()*. De forma resumida, é necessário observar se um vértice se liga com todos os ligantes de suas ligações.

Exemplificando: $R = \{(1,2) , (1,3) , (2,3) , (3,2)\}$

1: 2, 3

2: 3

3: 2

O programa observa a existência do relacionamento (1,2) - (2,3) e (1,3) – (3,2), logo devem haver as ligações (1,3) e (1,2) para que seja transitivo. Um looping passa pela matriz de relacionamentos e confere se o elemento 1, por exemplo, se relaciona com os

relacionamentos dos elementos 2 e 3, previamente determinados nos processos anteriores, e isso ocorre para todos os elementos.

Para inserir todas as ligações faltantes é necessário chamar a função diversas vezes, assim como para determinar a propriedade reflexiva utilizando a função *remove_reflexiva()*.

Caso falte alguma ligação para a transitividade, a mesma é armazenada em um vetor de caracteres dinamicamente alocado, realizando as conversões de inteiro para caractere, e são posteriormente mostradas para o usuário. Obs: o motivo da conversão para caractere foi dado pelo fato que a função *realloc()* não estava funcionando para vetores inteiros.

```
if(ligacoes_faltantes==NULL){
    ligacoes_faltantes = (char*) malloc (tamanho * sizeof (char));
    ligacoes_faltantes[0] = gr->elementos[i]+'0';
    ligacoes_faltantes[1] = gr->elementos[gr->arestas[gr->arestas[temp][j]][k]]+'0';
}
else{
    tamanho+=2;
    ligacoes_faltantes = (char*) realloc (ligacoes_faltantes,tamanho*sizeof(char));
    ligacoes_faltantes[tamanho-2] = gr->elementos[i]+'0';
    ligacoes_faltantes[tamanho-1] = gr->elementos[gr->arestas[gr->arestas[temp][j]][k]]+'0';
}

else{ //Caso não seja transitivo, imprimir ligações que satisfazem essa afirmação.
    printf("6. Transitiva: F\n");
    for(i=0;i<tamanho;i+=2){
        printf("(%c,%c); ",ligacoes_faltantes[i],ligacoes_faltantes[i+1]);
    }
    printf("\n");
}
printf("\n");
```

relacao_Equivalencia: verifica se o grafo é reflexivo, simétrico e transitivo ao mesmo tempo.

```
void relacao_Equivalencia(Grafo* gr){ //Verificar a relação de equivalencia
//Reflexiva+Simétrica+Transitiva
if(gr->reflexiva==1 && gr->simetrica==1 && gr->transitiva==1)
    printf("Relacao de equivalencia: V\n");
else
    printf("Relacao de equivalencia: F\n");
}
```

relacao_Ordem_Parcial: verifica se o grafo é reflexivo, anti-simétrico e transitivo ao mesmo tempo.

```
void relacao_Ordem_Parcial(Grafo* gr){ //Verificar a relação de ordem parcial
//Reflexiva+Anti-Simétrica+Transitiva
if(gr->reflexiva==1 && gr->anti_simetrica==1 && gr->transitiva==1)
    printf("Relacao de ordem parcial: V\n\n");
else
    printf("Relacao de ordem parcial: F\n\n");
}
```

fecho_Reflexivo: caso o grafo já seja reflexivo, é necessário apenas imprimir os relacionamentos já existentes:

```
//Imprimir todos os vertices ja existentes e depois os faltantes
printf("Fecho reflexivo da relacao = {");
int entrou=0;
for(i=0; i < gr->nro_vertices; i++){
    for(j=0; j < gr->grau[i]; j++){
        if(entrou)
            printf(",");
        entrou=0;
        if(i != gr->arestas[i][j]){
            printf("(%d,%d)",gr->elementos[i], gr->elementos[gr->arestas[i][j]]);
            entrou=1;
        }
    }
}
entrou=0;
for(i=0; i < gr->nro_vertices; i++){
    for(j=0; j < gr->grau[i]; j++){
        if(entrou)
            printf(",");
        entrou=0;
        if(i == gr->arestas[i][j]){
            printf("(%d,%d)", gr->elementos[gr->arestas[i][j]],gr->elementos[gr->arestas[i][j]]);
            entrou=1;
        }
    }
}
printf("}\n");
```

Caso contrário, é necessário completar com as ligações faltantes e imprimir os relacionamentos resultantes do novo grafo completo:

```
Grafo* gr2 = cria_Grafo();
int cont=0;
for(i=0; i < gr2->nro_vertices; i++){
    for(j=0; j < gr2->grau[i]; j++){
        if(i == gr2->arestas[i][j]){
        }
        else
            cont++;
    }
    if(cont == gr2->grau[i]){ //Inserir relacionamentos não reflexivos
        insereAresta(gr2,gr->elementos[i],gr->elementos[i]);
    }
    cont=0;
}
eh_Irreflexivo(gr2,1);
libera_Grafo(gr2);
```

fecho_Simetrico: caso o grafo já seja simétrico, é necessário apenas imprimir os relacionamentos existentes.

```
else if(fecho == 1){
    int entrou=0;
    gr2 = cria_Grafo();
    printf("Fecho simetrico da relacao = {");
    for(i=0; i<gr2->nro_vertices; i++){
        for(j=0; j<gr2->grau[i]; j++){
            for(k=0; k<gr2->grau[gr2->arestas[i][j]]; k++){
                if(i == gr2->arestas[gr2->arestas[i][j]][k]){
                    if(entrou)
                        printf(",");
                    if(gr2->arestas[i][j] != gr2->arestas[gr2->arestas[i][j]][k]){
                        int aux1=gr2->arestas[i][j];
                        int aux2=gr2->arestas[gr2->arestas[i][j]][k];
                        printf("(%d,%d)", gr2->elementos[aux2], gr2->elementos[aux1]);
                    }
                    else{
                        printf("(%d,%d)", gr2->elementos[gr2->arestas[i][j]], gr2->elementos[gr2->arestas[gr2->arestas[i][j]][k]]);
                    }
                    entrou=1;
                }
            }
        }
    }
    printf("\n");
}
```

Caso contrário, é necessário completar o grafo com as ligações faltantes e imprimir os relacionamentos resultantes do novo grafo completo:

```
/*A criação de dois grafos auxiliares de forma que um passe pela função remove_Simetrico e sua saída gere
as ligações necessárias para imprimir o fecho de forma completa. As ligações faltantes são realizadas no
segundo grafo auxiliar*/
Grafo* gr3 = cria_Grafo();
Grafo* gr4 = cria_Grafo();
while(controle!=0){
    controle = remove_Simetrico(gr3);
}
for(i=0; i < gr3->nro_vertices; i++){
    for(j=0; j < gr3->grau[i]; j++){
        insereAresta(gr4, gr3->elementos[gr3->arestas[i][j]], gr3->elementos[i]);
    }
}
printf("Fecho simetrico da relacao = {");
for(i=0; i<gr4->nro_vertices; i++){
    for(j=0; j<gr4->grau[i]; j++){
        for(k=0; k<gr4->grau[gr4->arestas[i][j]]; k++){
            if(i == gr4->arestas[gr4->arestas[i][j]][k]){
                if(entrou)
                    printf(",");
                if(gr4->arestas[i][j] != gr4->arestas[gr4->arestas[i][j]][k]){
                    int aux1=gr4->arestas[i][j];
                    int aux2=gr4->arestas[gr4->arestas[i][j]][k];
                    printf("(%d,%d)", gr4->elementos[aux2], gr4->elementos[aux1]);
                }
                else{
                    printf("(%d,%d)", gr4->elementos[gr4->arestas[i][j]], gr4->elementos[gr4->arestas[gr4->arestas[i][j]][k]]);
                }
                entrou=1;
            }
        }
    }
}
printf("\n");
libera_Grafo(gr3);
libera_Grafo(gr4);
```

fecho_Transitivo: só é necessário mostrar o fecho transitivo caso a relação seja assimétrica, pois caso não seja os pares já foram listados anteriormente (requisito não seguido por ser opcional).

Se necessário, é preciso completar os relacionamentos faltantes para tornar o grafo transitivo. Esses relacionamentos são obtidos na função eh_Transitivo.

```
else if(entrou == 0 && fecho == 1){
    insereAresta(gr2, gr->elementos[i], gr->elementos[gr->arestas[gr->arestas[temp][j]][k]]);
}
```

Feito isso, basta mostrar todas as ligações que determinam essa propriedade:

```
printf("Fecho transitivo da relacao = {");
if(gr->transitiva==1){
    for(i=0; i < gr->nro_vertices; i++){
        for(j=0; j < gr->grau[i]; j++){
            if(entrou)
                printf(",");
            printf("(%d,%d)",gr->elementos[i], gr->elementos[gr->arestas[i][j]]);
            entrou=1;
        }
    }
}
else if(controle_pior_caso=1){
    for(i=0; i < gr2->nro_vertices; i++){
        for(j=0; j < gr2->grau[i]; j++){
            if(entrou)
                printf(",");
            printf("(%d,%d)",gr2->elementos[i], gr2->elementos[gr2->arestas[i][j]]);
            entrou=1;
        }
    }
}
else{
    for(i=0;i<gr2->nro_vertices;i++){
        for(j=0;j<gr2->grau[i];j++){
            for(k=0;k<gr2->grau[gr2->arestas[i][j]];k++){
                if(i == gr2->arestas[gr2->arestas[i][j]][k]){
                    if(entrou)
                        printf(",");
                    if(gr2->arestas[i][j] != gr2->arestas[gr2->arestas[i][j]][k]){
                        int aux1=gr2->arestas[i][j];
                        int aux2=gr2->arestas[gr2->arestas[i][j]][k];
                        printf("(%d,%d)", gr2->elementos[aux2],gr2->elementos[aux1]);
                    }
                }
                else{
                    printf("(%d,%d)",gr2->elementos[gr2->arestas[i][j]],gr2->elementos[gr2->arestas[gr2->arestas[i][j]]]);
                }
            }
        }
    }
}
```

Dificuldades:

Durante o desenvolvimento do trabalho foram encontradas diversas barreiras a serem quebradas com o intuito de concluir a tarefa. Uma delas se refere ao conhecimento matemático no que tange ao assunto grafos, sobre o qual tive que estudar a fim de aprender mais nesse contexto. Outro problema foi na verificação da simetria de forma recursiva, lógica que foi abandonada devido a erros excessivos. E por fim, dificuldades na criação de funções que verificassem a transitividade das relações, pois era necessário pensar em uma maneira de percorrer a matriz de adjacências de uma forma que funcionasse para todos os casos possíveis. Mesmo com todos esses problemas, o esforço e a dedicação diária contribuíram para que as soluções fossem alcançadas.

Conclusão:

Após a finalização desse trabalho, é necessário reconhecer uma considerável evolução nos meus conhecimentos tanto na matemática quanto na área da informática. Foram estabelecidos grandes desafios que só foram superados com bastante estudo e dedicação. Assim sendo, o seu desenvolvimento contribuiu de forma notória em minha formação acadêmica.