TRABALHO PRÁTICO DE MATEMÁTICA DISCRETA

O código do arquivo “relacao.c” tem como objetivo ler informações sobre dois conjuntos, um de algarismos e outro de pares ordenados, e analisar as relações entre eles de forma automatizada. A saída do código é composta por três partes: propriedades, relações e fechos. Neste pdf, dividirei a explicação em 6 partes: parte inicial, análise de propriedades, análise de relações, cálculo dos fechos, impressão e leitura das entradas.

1. PARTE INICIAL

Inicialmente, além de declarar as bibliotecas utilizadas no código, declarei uma struct “par_ordenado”, estrutura de dados necessária para armazenar os pares analisados futuramente. Além disso, para facilitar a impressão, criei uma função “imprime_par”, que recebe como parâmetro um array de pares ordenados.

```
struct par_ordenado {
    int x;
    int y;
};
```

Outro ponto importante é a ordenação dos arrays. Para isso, declarei duas funções: “compara” e “ordena”. A função “compara”, recebe dois pares e, se x for diferente, retorna que a comparação deve ser feita aí, caso contrário, retorna que a comparação é feita em y. Essa função é utilizada em “ordena” como critério ao usar qsort.

Também é necessário a criação de uma função para remover inversos, que é importante para imprimir as exceções da propriedade anti-simétrica. Essa função percorre o array duas vezes e compara todos os pares, se encontra dois inversos ((x, y) e (y, x)), exclui o segundo e reajusta o índice dos próximos elementos.

```
void remove_inversos(struct par_ordenado pares[], int *tamanho) {
    for (int i = 0; i < *tamanho; i++)
    {
        for (int j = i + 1; j < *tamanho; j++)          {
            if (pares[i].x == pares[j].y && pares[i].y == pares[j].x)
            {
                for (int k = j; k < *tamanho - 1; k++)
                {
                    pares[k] = pares[k + 1];
                }
                (*tamanho)--;
                j--;
            }
        }
    }
}
```

De forma semelhante, criei uma função que remove duplicatas, uma vez que em alguns casos o mesmo elemento era adicionado duas vezes a um array. O funcionamento é muito semelhante à função anterior, muda apenas que o if compara se os elementos são idênticos.

2. ANÁLISE DAS PROPRIEDADES

Nessa parte do código, declarei 6 funções (uma para cada propriedade). Cada uma delas é composta por duas partes principais: uma para de fato checar a veracidade da propriedade e, caso falsa, descobrir as exceções à regra, e a segunda que imprime os resultados conforme pedido.

Segue um exemplo da primeira parte:

```
void is_reflexiva(int a[], int tamanho_a, struct par_ordenado pares[], int tamanho_pares) {

    bool reflexiva = true;

    struct par_ordenado excecoes[2450];
    int n_excecoes = 0;

    for (int i = 0; i < tamanho_a; i++)
    {
        bool found = false;
        for (int j = 0; j < tamanho_pares; j++)
        {
            if (pares[j].x == a[i] && pares[j].x == pares[j].y)
            {
                found = true;
                break;
            }
        }
        if (!found)
        {
            excecoes[n_excecoes].x = a[i];
            excecoes[n_excecoes].y = a[i];
            n_excecoes ++;
            reflexiva = false;
        }
    }

    ordena(excecoes, n_excecoes);
```

Aqui, declarei um array para armazenar as exceções e uma booleana que inicia como verdadeira. O código então percorre o conjunto e compara os pares e, caso não encontre um determinado par necessário, muda o valor da bool e acresce esse par às exceções. Ao final, ordena esse array para garantir a saída correta.

Agora, segue um exemplo da segunda parte:

```
    if (reflexiva)
    {
        printf("1. Reflexiva: V\n");
    } else
    {
        printf("1. Reflexiva: F\n");

        for (int i = 0; i < n_excecoes; i++)
        {
            imprime_par(excecoes[i]);

            if (i < n_excecoes - 1)
            {
                printf(", ");
            }
        }

        printf("\n");
    }
}
```

Essa parte, caso a propriedade seja verdadeira, imprime simplesmente “Propriedade: V”, caso contrário, imprime F e, na linha seguinte, as exceções separadas por “, “.

3. ANÁLISE DAS RELAÇÕES

Para verificar as relações do conjunto, criei duas funções, uma para cada tipo de relação. Como cada relação depende das propriedades, reutilizei a estrutura da primeira parte das funções das propriedades para verificar cada uma delas, e ao final, fiz um if para verificar se os critérios estavam de acordo e imprimi o resultado.

Aqui está um exemplo da parte final da função:

```
    if (reflexiva && anti_simetrica && transitiva)
    {
        printf("Relacao de ordem parcial: V\n");
    } else
    {
```

```
        printf("Relacao de ordem parcial: F\n");
    }
}
```

4. CÁLCULO DOS FECHOS

Para calcular os fechos, declarei 4 funções, vamos primeiramente falar das duas primeiras, que calculam os dois primeiros fechos. Elas são compostas por duas partes: a primeira checa a propriedade referente (mesma estrutura das funções de propriedades) e a segunda, se necessário, calcula os fechos e imprime os resultados.

Exemplo da segunda parte:

```
    printf("Fecho reflexivo da relacao:");

    if (reflexiva)
    {
        printf(" R\n");
    } else
    {
        struct par_ordenado fecho[2450];
        int tamanho_fecho = tamanho_pares + n_excecoes;

        for (int i = 0; i < tamanho_fecho; i++)
        {
            if (i < tamanho_pares)
            {
                fecho[i].x = pares[i].x;
                fecho[i].y = pares[i].y;
            } else
            {
                fecho[i].x = excecoes[i - tamanho_pares].x;
                fecho[i].y = excecoes[i - tamanho_pares].y;
            }
        }

        ordena(fecho, tamanho_fecho);

        printf("\n");

        for (int i = 0; i < tamanho_fecho; i++)
        {
            imprime_par(fecho[i]);
```

```
            if (i< tamanho_fecho - 1)
            {
                printf(", ");
            }
        }

        printf("\n");
    }
}
```

Nesse exemplo, podemos ver que o fecho só é calculado caso o conjunto não seja reflexivo. Para calculá-lo, como o fecho reflexivo é composto por conjunto + pares necessários para torná-lo reflexivo, adicionamos ao array "fecho", todos os elementos do conjunto e depois todas as exceções, e depois imprimimos.

As duas outras funções são um pouco diferentes. Acontece que, no caso da transitividade, não necessariamente o fecho será composto por conjunto + exceções. Por isso, declarei duas funções: uma para calcular o fecho e outra para imprimir o resultado. A função de calcular é chamada dentro da de imprimir caso o conjunto não seja transitivo.

A função de calcular funciona assim:

```
void calcula_fecho_transitivo(struct par_ordenado pares[], int
*tamanho) {

    struct par_ordenado fecho[2450];
    int n_fecho = *tamanho;

    for (int i = 0; i < *tamanho; i++)
    {
        fecho[i] = pares[i];
    }

    bool adicionado;
    do
    {
        adicionado = false;
        for (int i = 0; i < n_fecho; i++)
        {
            for (int j = 0; j < n_fecho; j++)
            {
                if (fecho[i].y == fecho[j].x)
                {
                    bool found = false;
                    for (int k = 0; k < n_fecho; k++)
```

```
                    {
                        if (fecho[k].x == fecho[i].x && fecho[k].y ==
fecho[j].y)
                        {
                            found = true;
                            break;
                        }
                    }
                    if (!found)
                    {
                        fecho[n_fecho].x = fecho[i].x;
                        fecho[n_fecho].y = fecho[j].y;
                        n_fecho++;
                        adicionado = true;
                    }
                }
            }
        }
    } while (adicionado);

    ordena(fecho, n_fecho);
    remove_duplicatas(fecho, &n_fecho);

    *tamanho = n_fecho;
    for (int i = 0; i < n_fecho; i++) {
        pares[i] = fecho[i];
    }
}
```

Essa função faz com que, enquanto o fecho final não for transitivo, os pares continuem sendo adicionados.

5. IMPRESSÃO:

Essa parte do código serve para facilitar a impressão. Aqui, montamos três funções: cada uma para imprimir uma das categorias. Ela funciona chamando todas as funções acima de uma só vez. Exemplo:

```
void imprime_relacoes(int a[], int tamanho_a, struct par_ordenado
pares[], int tamanho_pares) {

    relacao_equivalencia(a, tamanho_a, pares, tamanho_pares);

    relacao_ordem_parcial(a, tamanho_a, pares, tamanho_pares);
```

```
    printf("\n");
}
```

6. LEITURA DAS ENTRADAS

A leitura é realizada dentro da função main. Primeiramente, declarei e li o tamanho e o conjunto de números inteiros, usando um scanf e um for para percorrer o vetor enquanto o preenche. Depois, declarei o array de structs de pares ordenados e li. Para sua leitura, como não temos um tamanho fixo nem uma flag para finalizar a leitura, utilizei um while(scanf() =! EOF).

```
while (scanf("%d %d", &pares[tamanho_pares].x, &pares[tamanho_pares].y)
!= EOF) //checa se a leitura retorna um valor válido
    {
        tamanho_pares++; //acresce o tamanho do array
    }
```

Para finalizar, chamei as funções de impressão e dei um return 0 para oficialmente terminar o código.