

1 – Introdução

As relações podem estar presentes em nossa vida de diversas formas, como em relações familiares ou, num caso mais específico, no estudo da ciência da computação. Nesse trabalho, em especial, o problema é direcionado à relações binárias em conjuntos, que podem funcionar como abstrações para diversas situações, como em circuitos lógicos digitais.

Numa relação binária definida de A para A pode ser representada por um grafo dirigido que, por sua vez, pode ser representada numa matriz $n \times n$, sendo n a quantidade de elementos no conjunto A. Além disso, a relação pode, ou não, atender às seguintes propriedades: reflexividade, irreflexividade, simetria, anti-simetria, assimetria e transitividade, que deverão ser testadas pelo programa implementado.

Ademais, existem outros dois tipos de relações convenientes para o estudo: a de equivalência e a de ordem parcial. A primeira deve ser reflexiva, simétrica e transitiva, enquanto a segunda deve ser reflexiva, anti-simétrica e transitiva. Por fim, pode ser definido os fechos reflexivos, simétricos e transitivos da relação, que devem mostrar os pares necessários para que a mesma atenda à condição das respectivas propriedades.

2 – Implementação

2.1 – Estrutura de Dados

Para a representação do conjunto A, foi utilizado um vetor dinâmico, que é alocado no momento em que o programa recebe a quantidade de elementos do conjunto. Posteriormente, existe uma estrutura de repetição que preenche o vetor com os elementos lidos no arquivo. Todo esse processamento é executado dentro da função “leElementos”, que retorna o vetor preenchido ao “main”.

Já para a representação das relações entre os elementos de A para A, foi utilizada uma matriz $n \times n$, em que n é a quantidade de elementos do conjunto. Nesse caso, a cada relação lida pelo programa (até encontrar o fim do arquivo – EOF), a linha e a coluna referente aquela relação era preenchida com o número 1, representando que a mesma existe na relação. Por exemplo, se o programa lê o número 3 e o número 5 num conjunto que vai de 1 até 5, a posição equivalente a linha 3 e a coluna 5 da matriz é preenchida com o número 1.

Todo esse processamento é executado pela função “lePares”, que, além de tudo, atenta-se ao fato de que, não necessariamente a relação na primeira linha da entrada foi dada na ordem correta. Por fim, a função retorna a matriz preenchida com as relações passadas ao programa para o “main”.

2.2 – Funções e Procedimentos

Para a análise das diversas propriedades que pode ter uma relação binárias, foram implementados os seguintes métodos:

```
82 void reflexividade (int *reflexiva, int *irreflexiva, int **mat, int nElementos, int *vetElementos){  
83  
84     int *faltantesReflex = NULL, *faltantesIrreflex = NULL, i, tamanhoVetReflex = 0, tamanhoVetIrreflex = 0;
```

Figura 1: Assinatura do procedimento e criação de variáveis

Nesse método, são passados a partir do método main o ponteiro da variável reflexiva e da variável irreflexiva, além da matriz preenchida com os elementos da relação. Além disso, são passados o número de elementos no conjunto, além do vetor com os mesmos. As variáveis reflexiva e irreflexiva são criadas no método main e são, por default, verdadeiras. São também criados no método o vetor dinâmico para as relações faltantes para a reflexividade e a irreflexividade, necessários para serem mostrados durante a execução do programa, assim como seus respectivos tamanhos.

Para verificar a reflexividade ou a irreflexividade nesse método, a verificação necessária foi: se a diagonal principal é toda composta de 1, todo elemento se relaciona com ele mesmo e, portanto, a relação é reflexiva. Já se a diagonal principal é toda composta de 0, nenhum elemento se relaciona com ele mesmo e, portanto, a relação é irreflexiva. Por fim, o método imprime se a propriedade é verdadeira ou falsa e, caso seja falsa, mostra os pares faltantes (no caso da reflexiva) ou em excesso (no caso da irreflexiva) para que a relação atenda àquela propriedade.

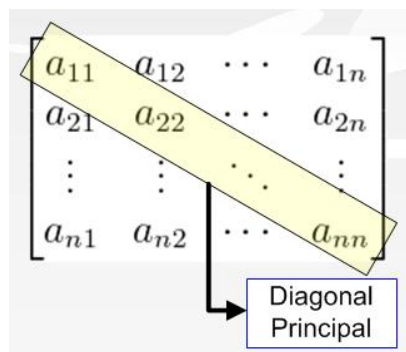


Figura 2: Diagonal principal analisada pelo procedimento

```
130 void simetria (int *simetrica, int *anti_simetrica, int **mat, int nElementos, int *vetElementos){
131
132     int *faltantesSimetrica = NULL, *faltantesAnti = NULL, i, j, tamanhoVetSim = 0, tamanhoVetAnti = 0;
```

Figura 3: Assinatura do procedimento e criação de variáveis

No método “simetria”, a criação de variáveis e passagem de parâmetros se dá da mesma maneira que no “reflexividade”, contudo, a verificação da propriedade acontece de maneira diferente. Existem duas estruturas de repetição, uma que vai de 0 até a última posição da matriz, e outra que vai de 0 até o valor do contador da primeira estrutura. Dessa forma, é passado apenas a parte superior da matriz.

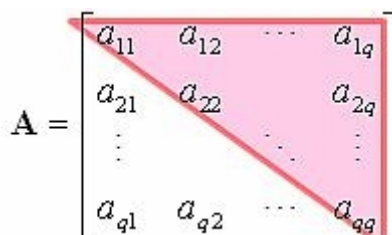


Figura 4: Matriz triangular superior

Sendo os contadores da estrutura de repetição “i” e “j”, para ser simétrica, em todo elemento da matriz a_{ij} que for igual a 1, o elemento a_{ji} também deve ser igual a 1. Da mesma forma, para ser anti-simétrica, todo elemento a_{ij} igual a 1 deve ter o elemento a_{ji} valendo 0. Por fim, o procedimento imprime se as propriedades são verdadeiras ou falsas, e, se não for simétrica, imprime os pares faltantes, assim como, se não for anti-simétrica, imprime os pares que impedem a relação de ser verdadeira nessa propriedade.

```
318     if (irreflexiva && anti_simetrica){
319         printf("5. Assimétrica: V\n");
320     }else{
321         printf("5. Assimétrica: F\n");
322     }
```

Figura 5: Verificação da assimetria

A verificação da propriedade assimétrica ocorre dentro do próprio “main”, já que é bastante simples e, portanto, não se mostrou necessária a criação de um procedimento específico para tal. Para ser assimétrica a relação deve ser irreflexiva e anti-simétrica.

Para a implementação da verificação da transitividade foram enfrentadas as maiores dificuldades. Foi escolhido o algoritmo de Floyd-Warshall, muito usado em aplicações de grafo para tal tarefa, embora existam soluções possíveis com multiplicação da matriz por ela mesma.

```
193 int** transitividade (int *transitivo, int **mat, int nElementos, int *vetElementos){
194
195     //Solução inspirada no algoritmo de Floyd-Warshall
196
197     int** matTrans = alocaMatriz(nElementos);
198
199     int i, j, k, *faltantesTrans = NULL, tamanhoVetTrans = 0;
200
201     //clonando matriz
202     for (i = 0; i < nElementos; ++i){
203         for (j = 0; j < nElementos; ++j){
204             matTrans[i][j] = mat[i][j];
205         }
206     }
207
208
209     for (i = 0; i < nElementos; ++i){
210         for (j = 0; j < nElementos; ++j){
211             for (k = 0; k < nElementos; ++k){
212                 if (matTrans[j][i] == 1 && matTrans[i][k] == 1 && matTrans[j][k] == 0){
213                     *transitivo = 0;
214                     matTrans[j][k] = 1;
215                 }
216             }
217         }
218     }
219 }
```

Figura 6: Implementação do algoritmo de Floyd-Warshall

Além da assinatura da função semelhante aos anteriores, dessa vez, é retornada ao main a matriz com o fecho transitivo, útil posteriormente para a impressão do mesmo. Tal matriz é alocada e clonada logo no começo da função.

Posteriormente, vemos o algoritmo de Floyd-Warshall em ação. O primeiro for verifica as relações n-árias que podem ser criadas para que a relação seja transitiva. O segundo for passa pelas linhas da matriz e, o terceiro, pelas colunas. O if verifica se a relação não é estabelecida (existe a para b, b para c mas não existe a para c) e, caso seja verdadeiro, a relação não é mais transitiva e o par é adicionado à matriz do fecho.

Por fim, assim como nos procedimentos anteriores, é impresso se a relação é verdadeira ou falsa e, caso seja falsa, é mostrado o par faltante para que seja transitiva.

```
329     if (reflexiva && simetrica && transitivo)
330         printf("Relacao de equivalencia: V\n");
331     else
332         printf("Relacao de equivalencia: F\n");
333
334
335     if (reflexiva && anti_simetrica && transitivo)
336         printf("Relacao de ordem parcial: V\n");
337     else
338         printf("Relacao de ordem parcial: F\n");
339 }
```

Figura 7: Implementação das relações de equivalência e ordem parcial

A implementação das relações de equivalência e ordem parcial se mostrou bem tranquila, atendendo apenas às seguintes condições: uma relação R é uma relação de equivalência se e somente se R é reflexiva, simétrica e transitiva; uma relação R é uma relação de ordem parcial se e somente se R é reflexiva, anti-simétrica e transitiva.

Por fim, era necessário a impressão dos fechos:

```
342     if(!reflexiva)
343         fechoReflexivo(nElementos, vet, matId);
344
345     if (!simetrica)
346         fechoSimetrico(nElementos, vet, matId);
347
348     if(!transitivo)
349         fechoTransitivo(nElementos, vet, matTrans);
```

Figura 8: Condições para impressão dos fechos

Os fechos são dados por todos os pares da relação acrescidos dos pares faltantes para aquela determinada propriedade, portanto, só são impressos quando a relação é falsa. Para o fecho reflexivo e simétrico, além da própria relação, são impressos, utilizando a mesma lógica da verificação nos procedimentos, os pares faltantes.

Já para o fecho transitivo, é utilizada a matriz transitiva criada pelo algoritmo de Floyd-Warshall e, com dois laços de repetição, todo o fecho transitivo é impresso.

2.2 – Ambiente de testes

O sistema operacional utilizado para o desenvolvimento do programa foi o Ubuntu 18.04 LTS, além do compilador GNU Compiler Collection (GCC).

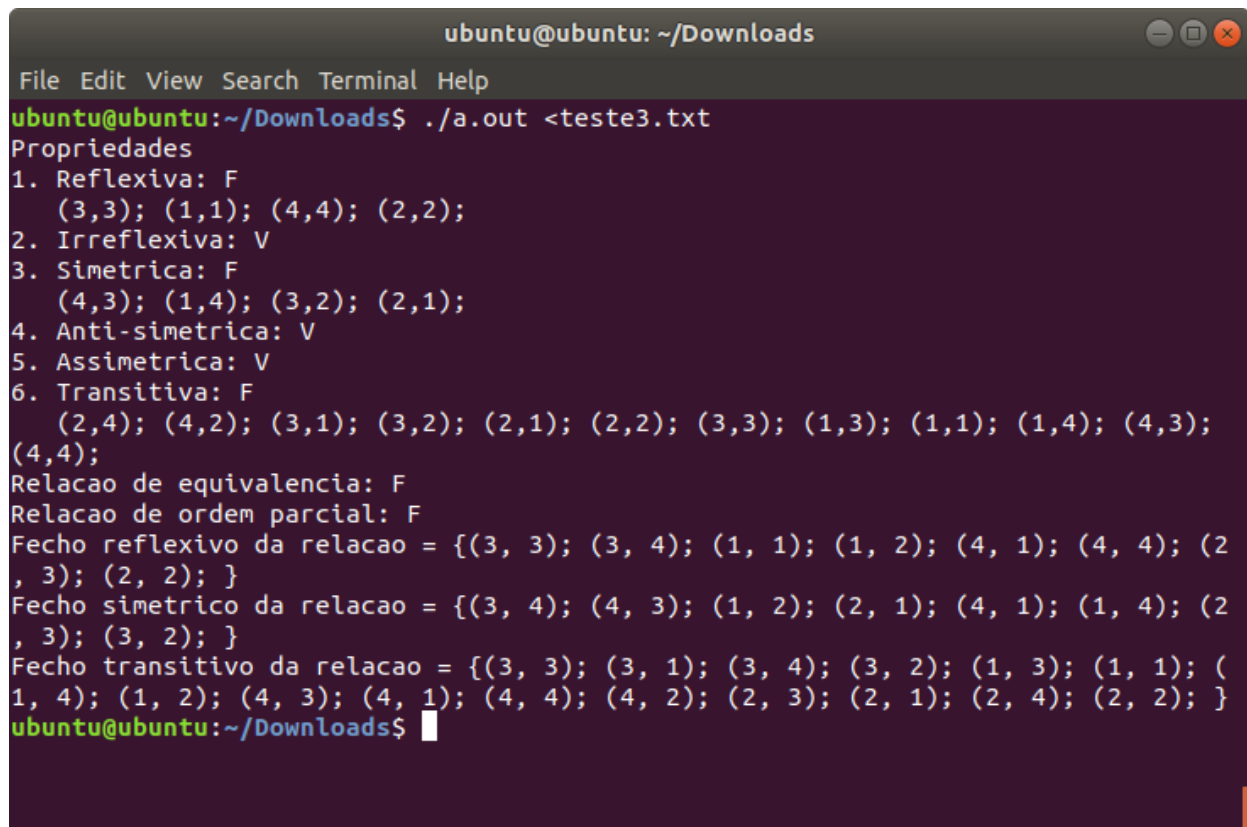
3 – Execução

A execução padrão do programa se dá pela passagem de um arquivo em que a primeira linha contém: o número de elementos da relação, seguido de seus elementos. No restante do arquivo, cada linha representa tem dois números, representando uma relação.

Para o arquivo da imagem abaixo, a saída esperada é:



Figura 9: Um dos arquivos usados nos casos de teste



```
ubuntu@ubuntu: ~/Downloads
File Edit View Search Terminal Help
ubuntu@ubuntu:~/Downloads$ ./a.out <teste3.txt
Propriedades
1. Reflexiva: F
   (3,3); (1,1); (4,4); (2,2);
2. Irreflexiva: V
3. Simetrica: F
   (4,3); (1,4); (3,2); (2,1);
4. Anti-simetrica: V
5. Assimetrica: V
6. Transitiva: F
   (2,4); (4,2); (3,1); (3,2); (2,1); (2,2); (3,3); (1,3); (1,1); (1,4); (4,3);
   (4,4);
Relacao de equivalencia: F
Relacao de ordem parcial: F
Fecho reflexivo da relacao = {(3, 3); (3, 4); (1, 1); (1, 2); (4, 1); (4, 4); (2,
3); (2, 2); }
Fecho simetrico da relacao = {(3, 4); (4, 3); (1, 2); (2, 1); (4, 1); (1, 4); (2,
3); (3, 2); }
Fecho transitivo da relacao = {(3, 3); (3, 1); (3, 4); (3, 2); (1, 3); (1, 1); (
1, 4); (1, 2); (4, 3); (4, 1); (4, 4); (4, 2); (2, 3); (2, 1); (2, 4); (2, 2); }
ubuntu@ubuntu:~/Downloads$
```

Figura 10: Saída esperada

4 – Conclusão

O trabalho foi bastante interessante pois fez com que fosse necessário um maior aprofundamento no estudo das relações, refletindo na facilidade de absorção do conteúdo para a avaliação. Além disso, para a implementação do programa, embora tenha sido necessário pensar para a verificação das demais propriedades, a principal dificuldade encontrada foi na transitividade da relação.

Foi bastante interessante conseguir utilizar o conhecimento que já havia adquirido a respeito de teoria dos grafos e poder aplica-lo nesse trabalho prático, sobretudo o algoritmo de Floyd-Warshall. Além disso, produzir uma saída padronizada mostrou a necessidade de se ter atenção nesse tipo de correção.

5 – Referências

- 1 - https://pt.wikipedia.org/wiki/Algoritmo_de_Floyd-Warshall
- 2 - https://en.wikipedia.org/wiki/Transitive_closure