Relações Binárias em Conjuntos (Documentação)

Victor Hugo Silva Moura
Ciência da Computação/ 2018/ 1º semestre
Matrícula: 2018054958

O Problema:

Criar um programa na linguagem C capaz de analisar e identificar relações binárias em conjuntos e categorizar essas relações por meio de suas propriedades. As relações que devem ser tratadas são do tipo AxA, ou seja, de um conjunto A para o próprio conjunto A. Sendo assim, as relações poderiam ser representadas por um grafo dirigido. O programa deve ser capaz de armazenar uma relação, fornecida pelo usuário, em um grafo e, após isso, identificar as propriedades e mostrá-las na tela, bem como categorizar a relação por meio dessas propriedades.

A Solução:

A resolução desse problema se deu por meio de três passos fundamentais, que foram:

- Desenvolvimento do armazenamento de dados e teste das primeiras propriedades para um caso específico
- Desenvolvimento das propriedades mais trabalhosas e categorização das relações, ainda para um caso específico
- Generalização do problema para todos os casos

1ª Etapa:

Era necessário decidir qual o melhor modo de armazenar os vértices e arestas do grafo com o objetivo de facilitar a leitura e utilização dos dados. Esse armazenamento foi feito através de uma matriz, onde as linhas indicam o vértice de origem e as colunas o vértice de destino. Sendo assim, o elemento a_{ij} (sendo *i* a i-ésima linha e *j* a j-ésima coluna da matriz) possuí um inteiro (1 ou 0) que indica se a aresta que liga o vértice *i* ao vértice *j* existe. Se o elemento for 1, então a aresta existe. Caso contrário, a aresta não existe.

Após essa abstração inicial, era hora de começar a parte de leitura dos dados e armazenamento na matriz, seguindo a lógica acima. O primeiro método de leitura não identificava os vértices (numeração dos vértices sempre de 1 a *n*, sendo *n* o número de vértices informado pelo usuário) e necessitava de uma variável que indicava o número de relações. (Figura 1)

```c
int main(){
  int qteElem, qteRel, i, j, count, n1, n2;
  int **grafo;

  scanf("%d", &qteElem);
  grafo = (int **) calloc(qteElem, sizeof(int *));
  for(i = 0; i < qteElem; i++)
    grafo[i] = (int *) calloc(qteElem, sizeof(int));

  scanf("%d", &qteRel);
  for(count = 0; count < qteRel; count++){
    scanf("%d %d", &n1, &n2);
    grafo[n1-1][n2-1] = 1;
  }
```

Figura 1: Código responsável pelo armazenamento inicial

Sendo assim, a entrada padrão modificada e a matriz resultante seriam:

Entrada

```
6 18
1 3
3 5
5 1
3 1
5 3
1 5
2 4
4 6
6 2
4 2
6 4
2 6
1 1
2 2
3 3
4 4
5 5
6 6
```

Matriz

ID	1	2	3	4	5	6
1	1	0	1	0	1	0
2	0	1	0	1	0	1
3	1	0	1	0	1	0
4	0	1	0	1	0	1
5	1	0	1	0	1	0
6	0	1	0	1	0	1

Com essa parte pronta, o próximo objetivo era identificar as primeiras propriedades da relação.

– Reflexividade e Irreflexividade:

Essas duas propriedades estão agrupadas em um mesmo tópico pois, foi desenvolvida apenas uma função que confere as duas propriedades. A função recebe 4 parâmetros, sendo eles: a matriz principal, o número de linhas/colunas da matriz, um valor a ser testado (será explicado em seguida) e uma matriz secundária para armazenar os pares ausentes ou presentes que tornavam a propriedade falsa, caso ela fosse falsa. (Figura 2)

Essa função percorre a diagonal principal da matriz e confere se existe algum elemento igual ao valor fornecido de parâmetro. Esse valor pode ser 1 ou 0 e, indica o número que, caso apareça na diagonal principal, torna a propriedade falsa. Por exemplo, na propriedade de Reflexividade (sendo *A* o conjunto, *R* a relação):

Se $\exists x$ em A tal que $(x,x) \notin R$, então a propriedade é falsa.

Passando isso para o código, se existe algum elemento da diagonal principal igual a 0, então a propriedade é falsa, pois isso significa que, para algum x não existe uma aresta dele para ele mesmo. Nesse caso, o valor a ser testado deve ser 0. Já para a propriedade de Irreflexividade, temos o caso contrário:

Se $\exists x$ em A tal que $(x,x) \in R$, então a propriedade é falsa.

Isso, em código, significa que caso exista algum elemento da diagonal principal igual a 1, então a propriedade é falsa, e nesse caso, o valor a ser testado é 1.

Em paralelo a isso, para cada elemento encontrado que torna a propriedade falsa, é feita uma cópia desse elemento para a matriz secundária fornecida. Assim, é possível saber todos os elementos que tornam a propriedade falsa.

A função retorna 1 caso a propriedade seja verdadeira e 0 caso seja falsa.

```c
//funçao que checa se o grafo é reflexivo ou irreflexivo
int reflexividade(int **grafo, int tam, int valor, int **grafo2){
  int i, validador = 1;
  limpaGrafo(grafo2, tam);
  for(i = 0; i < tam; i++){
    //se existir alguma linha com o valor diferente, então retorna falso
    if(grafo[i][i] == valor){
      validador = 0;
      grafo2[i][i] = 1;
    }
  }
  return validador;
}
```

Figura 2: Função que confere Reflexividade e Irreflexividade

– Simetria e Assimetria:

Duas propriedades que também estão agrupadas pois foram solucionadas utilizando uma única função. Essa função recebe também 4 parâmetros: a matriz principal, o número de linhas/colunas da matriz, um valor a ser testado e uma matriz secundária. (Figura 3)

A função percorre toda a matriz procurando por um caso que torne a propriedade falsa. No caso da Simetria, esse seria (sendo A o conjunto, R a relação):

Se $\exists x$ e y em A tal que $(x,y) \in R$ e $(y,x) \notin R$, então a propriedade é falsa.

Nesse caso, se um elemento a_{ij} for igual a 1 e o elemento a_{ji} for igual a 0, a propriedade é falsa (para i e j índices da matriz). Sendo assim, o valor de teste deve ser 0. Deve-se percorrer toda a matriz procurando por algum elemento igual a 1. Caso encontrar, basta inverter os índices e ver se o elemento com índices invertidos é igual ao valor de teste. Se for, a propriedade é falsa e o elemento que torna falso é copiado para a matriz secundária.

Semelhante à resolução da Irreflexividade, basta inverter o valor de teste (passar ele para 1) para testar a Assimetria. Isso ocorre pois,

Se $\exists x$ e y em A tal que $(x,y) \in R$ e $(y,x) \in R$, então a propriedade é falsa.

E esse é basicamente o caso “contrário” do anterior.

Essa função também retorna 1 se a propriedade é verdadeira e 0 se é falsa.

```c
//funcao que checa se o grafo é simétrico ou assimétrico
int simetria(int **grafo, int tam, int valor, int **grafo2){
  int i, j, validador = 1;
  limpaGrafo(grafo2, tam);
  for(i = 0; i < tam; i++){
    for(j = 0; j < tam; j++){
      if(grafo[i][j] == 1 && grafo[j][i] == valor){
        validador = 0;
        grafo2[j][i] = 1;
      }
    }
  }
  return validador;
}
```

Figura 3: Função que confere Simetria e Assimetria

– Anti-Simetria:

Primeira propriedade que teve uma função exclusiva para conferí-la. A função recebe 3 parâmetros, sendo eles: a matriz principal, o número de linhas/colunas da matriz e uma matriz secundária. (Figura 4)

A função também percorre toda a matriz procurando por um caso que torne a propriedade falsa. Nesse caso seria (sendo *A* o conjunto, *R* a relação):

Se $\exists x$ e y em A tal que $(x,y) \in R$, $(y,x) \in R$ e $x \neq y$, então a propriedade é falsa.

Nesse caso, se um elemento a_{ij} for igual a 1 e o elemento a_{ji} for igual a 1 porém, o índice *i* for diferente de *j*, a propriedade é falsa (para *i* e *j* índices da matriz). Isso, em código, significa que deve-se percorrer a matriz procurando por pares simétricos. Ao encontrar um, basta verificar se os índices são diferentes. Se forem, a propriedade é falsa e se o índice *i* é menor que o *j*, o elemento é copiado para a matriz secundária (evita a repetição ao imprimir pares que tornam falsa).

A função retorna 1 caso a propriedade seja verdadeira e 0 caso seja falsa.

```c
//funcao que checa se o grafo é antisimétrico
int antiSimetria(int **grafo, int tam, int **grafo2){
  int i, j, validador = 1;
  limpaGrafo(grafo2, tam);
  for(i = 0; i < tam; i++){
    for(j = 0; j < tam; j++){
      if(grafo[i][j] == 1 && grafo[j][i] == 1)
        if(i != j && i < j){
          validador = 0;
          grafo2[i][j] = 1;
        }
    }
  }
  return validador;
}
```

Figura 4: Função que confere Anti-Simetria

2ª Etapa:

A única propriedade faltante nesse momento é a Transitividade, que é uma propriedade mais difícil de ser conferida e, somente com ela é possível categorizar as relações.

– Transitividade:

Seguindo o mesmo pensamento das outras funções, basta encontrar um caso que torne a propriedade falsa. A função recebe 3 parâmetros sendo eles: a matriz principal, o número de linhas/colunas da matriz e uma matriz secundária. (Figura 5)

A função percorre toda a matriz procurando por um caso que torne a propriedade falsa. Nesse caso seria (sendo *A* o conjunto, *R* a relação):

Se $\exists x, y$ e z em A tal que $(x,y) \in R$ e $(y,z) \in R$ mas, $(x,z) \notin R$, então a propriedade é falsa.

Nesse caso, se um elemento a_{ij} for igual a 1 e o elemento a_{jk} for igual a 1 porém, o elemento a_{ik} for igual a 0, a propriedade é falsa (para *i*, *j* e *k* índices da matriz). Isso, em código, significa que deve-

se percorrer a matriz procurando por índices *i,j* que tenham elemento a_{ij} igual a 1. Após isso, deve-se encontrar índices *k* que tenham elemento a_{jk} igual a 1. A partir disso, basta pegar o elemento com índices *i,k* e verificar se ele é igual a 0. Se for, então a propriedade é falsa e esse elemento é copiado para a matriz secundária.

A função retorna 1 caso a propriedade seja verdadeira e 0 caso seja falsa.

```
//funcao que checa se o grafo é transitivo
int transitividade(int **grafo, int tam, int **grafo2){
  int i, j, k, validador = 1;
  limpaGrafo(grafo2, tam);
  for(i = 0; i < tam; i++){
    for(j = 0; j < tam; j++){
      if(grafo[i][j]){
        for(k = 0; k < tam; k++){
          if(grafo[j][k] == 1 && grafo[i][k] == 0){
            validador = 0;
            //só marca auxiliar se não houver marcação já
            if(grafo2[i][k] == 0){
              grafo2[i][k] = 1;
            }
          }
        }
      }
    }
  }
  return validador;
}
```

Figura 5: Função que confere Transitividade

– Funções Auxiliares:

Funções utilizadas para inicializar corretamente as funções de propriedades, imprimir na tela os resultados e retornar quais propriedades eram verdadeiras ou falsas. Divididas em três categorias principais, de acordo com as propriedades: Reflexividade, Simetria e Transitividade. A principal delas, e a que será mais abordada é a função responsável pela Transitividade, pois esta, além de inicializar a função e mostrar os resultados, corrige também uma “falha” da função para alguns casos específicos.

– checaTransitividade:

Função mais complexa do código, responsável por aplicar corretamente a função de Transitividade, de modo que todos os pares faltantes sejam listados.

A função de Transitividade confere inicialmente os elementos faltantes mas, existem casos em que apenas os primeiros elementos, quando adicionados ao grafo, não tornam ele ainda transitivo. Por exemplo:

- Seja A = {0, 1, 2, 3} e uma relação R = {(0, 1), (1, 2), (2, 3)}.
- Após a primeira passagem da função, teríamos os seguintes pares faltantes:
  - (0, 2) e (1, 3)
- Esses pares, se adicionados à relação inicial, ainda não tornam ela transitiva. Falta o par (0, 3) para que essa relação seja realmente transitiva.

Com isso, é necessário adicionar os pares faltantes ao grafo/matriz e reaplicar a função até que o grafo seja completamente transitivo. Como a matriz original não pode ser alterada para não prejudicar outra utilização futura, os dados dela são copiados para uma matriz auxiliar e essa matriz recebe os pares faltantes, caso a original não seja transitiva. Essa matriz auxiliar está presente nas funções checaReflex e checaSimetria também, pois elas serão muito úteis para imprimir os fechos.

Para adicionar os novos pares à matriz auxiliar, foi criada uma função somaGrafos (Figura 6), responsável por adicionar os pares faltantes a essa matriz. Esses pares são adquiridos por meio da matriz secundária que é passada como parâmetro para cada função de propriedade.

```
void somaGrafos(int **grafo1, int **grafo2, int **grafo3, int tam){
  int i, j;
  for(i = 0; i < tam; i++){
    for(j = 0; j < tam; j++){
      if(grafo1[i][j] + grafo2[i][j] == 2)
        grafo3[i][j] = 1;
      else
        grafo3[i][j] = grafo1[i][j] + grafo2[i][j];
    }
  }
}
```

Figura 6: Função somaGrafos

Como a matriz auxiliar é responsável por guardar todos os pares da matriz original e os pares novos, foi criada também uma função diferencaGrafos (Figura 7), responsável por fazer a diferença entre dois grafos, ou seja, mostrar quais pares não estão presentes em ambas as matrizes passadas como parâmetro. Uma terceira matriz é passada para armazenar os dados.

```
void diferencaGrafos(int **grafo1, int **grafo2, int **grafo3, int tam){
  int i, j;
  for(i = 0; i < tam; i++){
    for(j = 0; j < tam; j++){
      //diferença armazenada no grafo2 devido à ordem de passagem de parâmetros
      grafo2[i][j] = grafo3[i][j] - grafo1[i][j];
    }
  }
}
```

Figura 7: Função somaGrafos

Utilizando tudo isso, foi possível construir a função checaTransitividade (Figura 8), bem como as outras que já foram citadas. Todas as funções auxiliares utilizam somaGrafos porém, só a função checaTransitividade utiliza o método diferencaGrafos.

```c
int checaTransitividade(int **grafo, int tam, int **grafoResultante, int *elementos){
  int i, j, **grafo2, retorno = 0;
  grafo2 = alocaGrafo(tam);
  printf("6.\tTransitiva: ");
  if(transitividade(grafo, tam, grafo2)){
    printf("V\n");
    retorno = 1;
  }
  else{
    printf("F\n\t");
    //é necessário passar os valores do grafo para o grafoResultante. Reaproveitamento de função
    somaGrafos(grafo, grafo, grafoResultante, tam);
    //enquanto não for transitivo, aplica a função de transitividade e adiciona os novos vértices ao grafo
    do{
      somaGrafos(grafoResultante, grafo2, grafoResultante, tam);
   }while(!transitividade(grafoResultante, tam, grafo2));
   //faz a diferença entre grafos para descobrir quais vértices foram adicionados
   diferencaGrafos(grafo, grafo2, grafoResultante, tam);
    for(i = 0; i < tam; i++){
      for(j = 0; j < tam; j++){
        if(grafo2[i][j])
          printf("(%d,%d); ", elementos[i], elementos[j]);
      }
    }
    printf("\n");
  }
  liberaGrafo(grafo2, tam);
  return retorno;
}
```

Figura 8: Função checaTransitividade

A função recebe 4 parâmetros: a matriz principal, o número de linhas/colunas da matriz, uma matriz secundária (que serve como auxiliar nesse momento mas, será utilizada futuramente nos fechos) e um vetor de elementos (implementado na 3ª parte apenas). Ela retorna 1 caso a relação seja transitiva e 0 caso não seja.

– Outras Funções Auxiliares:

As outras funções realizam um procedimento parecido com a função checaTransitividade, porém de forma menos complexa. Elas imprimem as propriedades correspondentes e retornam se elas são verdadeiras ou não. A função checaReflex retorna 1 se a relação é reflexiva e 0 caso contrário. A função checaSimetria tem um retorno mais complexo pois, retorna 0 se a relação não é simétrica nem anti-simétrica, 1 se a relação é apenas simétrica, 2 se é apenas anti-simétrica e 3 se é simétrica e anti-simétrica. Ambas as funções recebem 4 parâmetros, sendo eles: a matriz principal, o número de linhas/colunas da matriz, uma matriz secundária e um vetor de elementos.

– Categorizando as Relações:

Feita a checagem das propriedades, agora é possível categorizar as relações. Existem dois tipos de relações bem importantes para esse trabalho: equivalência e ordem parcial. 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 meio do retorno das funções auxiliares pode-se conferir se a relação encaixa em algum dos tipos. Foram feitas duas funções (checaEquivalencia e checaOrdemParcial) para categorizar a relação. (Figura 9)

```c
void checaEquivalencia(int r, int s, int t){
  printf("Relacao de equivalencia: ");
  if(r == 1 && (s == 1 || s == 3) && t == 1)
    printf("V\n");
  else
    printf("F\n");
}

void checaOrdemParcial(int r, int s, int t){
  printf("Relacao de ordem parcial: ");
  if(r == 1 && (s == 2 || s == 3) && t == 1)
    printf("V\n");
  else
    printf("F\n");
}
```

Figura 9: Funções checaEquivalencia e checaOrdemParcial

3ª Etapa:

Hora de generalizar o programa para outros casos, pois ele ainda funciona apenas para vértices de 1 a *n* e necessita de ter a quantidade de relações binárias informada. Mas antes, falta mostrar os fechos reflexivo, simétrico e transitivo.

– Fechos:

Pode-se definir informalmente o fecho reflexivo, simétrico e transitivo de uma relação *R* como o menor número de pares ordenados necessários para transformar *R* em reflexiva, simétrica e transitiva, respectivamente.

Todos estão agrupados em um único tópico pois, foi criada apenas uma função para mostrar os fechos. A função recebe 4 parâmetros, sendo eles: a matriz do fecho, o número de linhas/colunas da matriz, uma string (que contém o nome do fecho) e um vetor de elementos. (Figura 10)

A matriz do fecho não é nada mais que a matriz auxiliar presente nas funções auxiliares. Para cada função, foi criada uma matriz correspondente no main (grReflex, grSimet, grTrans) que é responsável por guardar a relação inicial e os pares que tornam cada uma delas verdadeiras de acordo com a propriedade. Por isso essa matriz sempre recebia a matriz principal somada à matriz secundária, que possuía os pares faltantes da relação. Dessa forma, é possível obter o fecho sem que haja uma nova checagem de pares faltantes.

A função imprime o nome do fecho e logo em seguida os elementos, seguindo o padrão pedido na descrição do trabalho. Exemplo:

Fecho transitivo da relação = {(3,3),(3,5),(3,7),(4,4),(4,6),(4,8),(5,3),(5,5),(5,7),(6,4),(6,6),(6,8),(7,3),(7,5),(7,7),(8,4),(8,6),(8,8)}

Para imprimir dessa forma, foi criado uma variável de validação. Essa variável começa como 1 e ao encontrar o primeiro elemento igual a 1 da matriz, ela se torna 0. Dessa forma, toda vez antes de imprimir um novo elemento, é impresso uma vírgula. Ao final de todos os elementos, basta imprimir as chaves que indicam o final do fecho.

```c
void fecho(int **grafo, int tam, char *nome, int *elementos){
  int i, j, validador = 1;
  printf("Fecho %s da relacao = {", nome);
  for(i = 0; i < tam; i++){
    for(j = 0; j < tam; j++){
      if(grafo[i][j]){
        if(!validador)
          printf(",");
        printf("(%d,%d)", elementos[i], elementos[j]);
        validador = 0;
      }
    }
  }
  printf("}\n");
}
```

Figura 10: Função que imprime os fechos

– Generalização da Solução:

Como dito anteriormente, o programa até este ponto funciona apenas para vértices indo de 1 a *n* e precisa que seja informado o número de total de pares ordenados. É necessário criar um modo de identificar a finalização da entrada sem pedir o usuário para informar o número de pares e de codificar a entrada para que o usuário possa escolher os vértices que ele queira.

Como a entrada é feita por redirecionamento de arquivo e é requisito ler pela entrada padrão da linguagem C (ou seja, o teclado), foi feito um scanf que para apenas quando ele identifica o final do arquivo, no caso ‘EOF’. (Figura 11)

Já a codificação da entrada foi feita em duas etapas: leitura dos vértices informados pelo usuário, conversão dos pares para índices correspondentes na matriz. Para ler os vértices, foi criado um vetor de elementos (citado anteriormente na parte de Funções Auxiliares, na função de Transitividade). Esse vetor recebe armazena o número de cada vértice informado pelo usuário. Com isso, a conversão de elementos ficou mais fácil. Essa conversão foi feita através da função converteIndice (Figura 12). Essa função pega um valor digitado pelo usuário no par ordenado e confere se esse valor está no vetor de elementos. Se sim, ele retorna o índice em que ele se encontra no vetor, que será o índice correspondente na matriz. Isso se deve ao fato da matriz ter tamanho *n* x *n* e o vetor ter tamanho *n*, sendo *n* o primeiro valor digitado pelo usuário, que corresponde ao número de elementos do conjunto. Caso o valor não se encontre no vetor, a função retorna -1 e o par ordenado não é armazenado na matriz.

```c
void leGrafo(int **grafo, int tam, int *elementos){
  int i, n1, n2;
  while (scanf("%d %d", &n1, &n2) ≠ EOF){
    n1 = converteIndice(n1, elementos, tam);
    n2 = converteIndice(n2, elementos, tam);
    if(n1 ⩾ 0 && n2 ⩾ 0)
      grafo[n1][n2] = 1;
  }
}
```

Figura 11: Função que lê os pares ordenados e armazena no grafo

```c
int converteIndice(int indice, int *elementos, int tam){
  int i;
  for(i = 0; i < tam; i++){
    if(indice == elementos[i])
      return i;
  }
  return -1;
}
```

Figura 12: Função que converte índice digitado pelo usuário

Dessa forma, o usuário pode gerar as relações que precisar sem nenhum esforço. Exemplos:

Entrada

```
6 3 4 5 6 7 8
3 5
5 7
7 3
5 3
7 5
3 7
4 6
6 8
8 4
6 4
8 6
4 8
3 3
4 4
5 5
6 6
7 7
8 8
```

Matriz

ID	3	4	5	6	7	8
3	1	0	1	0	1	0
4	0	1	0	1	0	1
5	1	0	1	0	1	0
6	0	1	0	1	0	1
7	1	0	1	0	1	0
8	0	1	0	1	0	1

Entrada

```
6 2 3 4 6 7 9
4 7
7 4
3 6
3 9
6 9
9 3
6 3
9 6
2 2
3 3
4 4
6 6
7 7
9 9
```

Matriz

ID	2	3	4	6	7	9
2	1	0	0	0	0	0
3	0	1	0	1	0	1
4	0	0	1	0	1	0
6	0	1	0	1	0	1
7	0	0	1	0	1	0
9	0	1	0	1	0	1

Com essa codificação, é possível aplicar as funções para diversos tipos de relações em vez de apenas uma. E além disso, o vetor de elementos permite a impressão correta pois, o índice que se acessa na matriz corresponde ao índice do vetor, então basta imprimir o elemento do vetor ao invés do índice. Isso é o que foi feito nas funções auxiliares e de fecho.

Considerações Finais:

A realização deste trabalho foi um grande desafio. Escolher as formas adequadas de armazenamento e tratamento de dados foi uma tarefa que exigiu certo tempo e abstração mas, creio que fazer e concluir esse trabalho contribuiu bastante para meus conhecimentos em programação e em Matemática Discreta também.

Optei por trazer essa documentação em formato de relato pois, considerei essa a melhor forma de mostrar o que foi feito acompanhado do raciocínio envolvido por trás de cada etapa. Dessa forma, é possível seguir perfeitamente os passos da resolução de cada trecho do problema, bem como as subdivisões que fiz para resolver. Tentei explicar da forma mais resumida e clara possível tentando facilitar a compreensão do raciocínio.