Relações Binárias

Matemática Discreta - ICEx/UFMG

Documentação - Trabalho Prático

— LaTeX—

Eduardo Capanema
eduardocapanema@ufmg.br
Matrícula: 2020041515

Contents

1 Introdução

A Matemática e a Computação estão interligadas em diversas aplicações e áreas do conhecimento. Em nosso trabalho prático, proposto pela disciplina de Matemática Discreta da Universidade Federal de Minas Gerais, abordamos o tema de Relações Binárias através da modelagem de um grafo direcionado, que é representado utilizando uma Matriz de Adjacência.

2 Estratégia

Grafos e matrizes de representação para seus vértices e arestas, são extremamente úteis e corriqueiros no estudo de problemas variados dentro da Ciência da Computação em geral.

Em nosso trabalho, criamos um código na linguagem c capaz de receber uma entrada padronizada que descreveremos a seguir e distribui esses dados, relativos a um grafo de relações binárias dirigido, em uma matriz bidimensional, ou seja, composta por um conjunto de vetores $v_1, v_2, ..., v_n$ que compõe uma matriz M_{nxn}.

2.1 Matriz de Adjacência

Uma matriz de adjacência é uma forma bastante útil de se representar um grafo.

"Dado um grafo G com n vértices, podemos representá-lo em uma matriz n x n $A(G) = [aij]$ (ou simplesmente A). A definição precisa das entradas da matriz varia de acordo com as propriedades do grafo que se deseja representar, porém de forma geral o valor a_{ij} guarda informações sobre como os vértices vi e vj estão relacionados (isto é, informações sobre a adjacência de vi e vj)." [Fonte: wikipedia https://pt.wikipedia.org/wiki/Matriz_de_adjacencia]

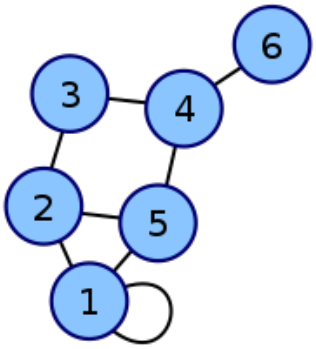


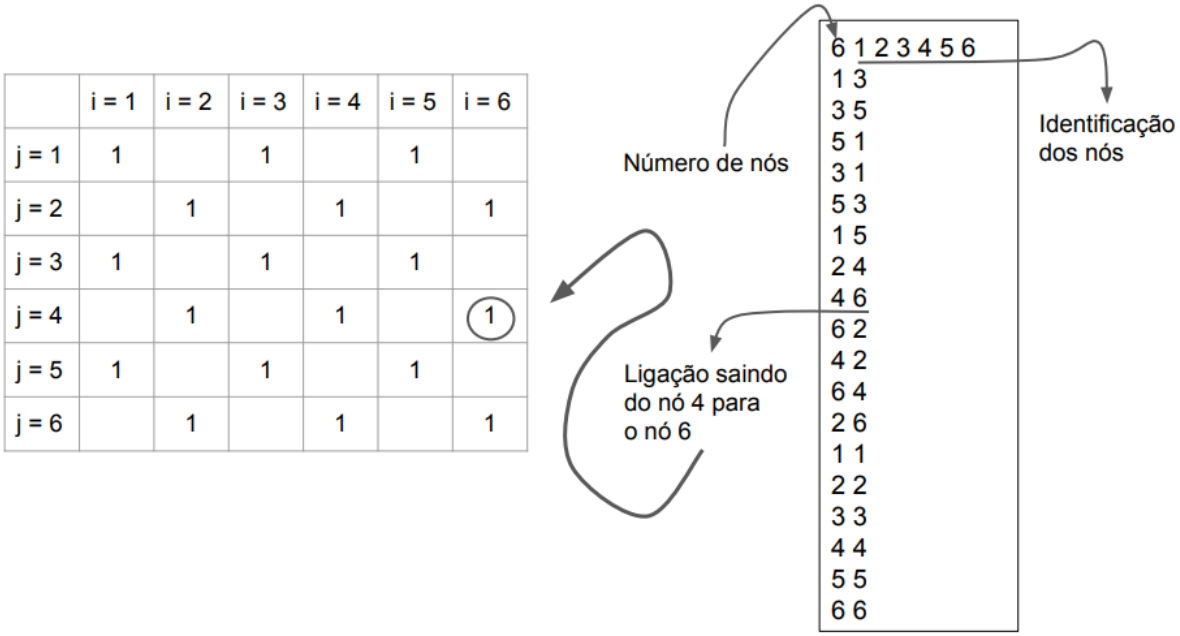
Fig. 1 - Um grafo genérico que pode ser representado por uma Matriz de Adjacência

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Fig. 2 - Um exemplo de uma Matriz de Adjacência

2.2 Matriz nas Relações Binárias

Desta forma, a partir de uma entrada padronizada como ilustrado abaixo, através do código relacionado ao fim deste documento na seção Apêndice, que exploraremos mais detalhadamente ao longo deste relatório, fomos capazes de relacionar, de maneira discreta, os nós (ou vértices) de nossas instâncias de dados.



	i = 1	i = 2	i = 3	i = 4	i = 5	i = 6
j = 1	1		1		1	
j = 2		1		1		1
j = 3	1		1		1	
j = 4		1		1		1
j = 5	1		1		1	
j = 6		1		1		1

Fig. 3 - Relação entre entradas e Matriz de Adjacência.

2.3 Entrada de Dados

A entrada dos dados consiste em um inteiro positivo inicial n, seguido de n inteiros, separados por espaços até o final da primeira linha. Realizamos a leitura desses dados, dentro de nossa função main da seguinte forma:

```
  int n;
  scanf( "%d", &n);

  int f[n];
  for ( int k=0; k<n; k++ ) {
    scanf( "%d", &f[k] );
  }
```

Em seguida, inicializamos uma matriz dinamicamente, tomando o cuidado de acrescentar uma linha e uma coluna a mais do que o número n, de forma a permitir o uso de um cabeçalho nas linhas e nas colunas, útil para a atribuição dos valores relativos dos nós, e capturamos os pares ordenados fornecidos, conforme abaixo:

```
int** matriz;
  matriz = ( int** ) malloc( sizeof( int* ) * (n+1) );
  for( int m=0; m<=n; m++ ) {
    matriz[m] = ( int* ) malloc( sizeof( int ) * (n+1) );
  }

  //

 int a, b;
  int rc;
  while ( (rc = scanf( "%d", &a ) ) == 1 ) {
    scanf( "%d", &b );
    int lb = 0;
    int cb = 0;
    // percorrer linha 0 e coluna 0
    for( lb=0; lb<=n; ++lb ) {
      for( cb=0; cb<=n; ++cb ) {
```

```
        if( matriz[lb][0] == a && matriz[0][cb] == b ) {
          matriz[lb][cb] = 1;
        }
      }
    }
  }
```

Essa estratégia se mostrou extremamente útil para que pudéssemos acessar diretamente os *labels* ou valores relativos dos nós, ao longo de nossa implementação.

3 Propriedades

Em seguida, prosseguimos com a descrição no código de nossas funções, sob um paradigma de programação funcional não-estruturada (i.e. sem estruturas struct), uma vez que avaliamos ser tal abordagem suficientemente clara e concisa para a realização da tarefa requerida.

3.1 Reflexividade

Nossa função (is_reflexiva) busca, dentro da matriz, passada por referência como parâmetro à função - veremos na seção de análise de complexidade mais à frente neste documento que essa abordagem é importante para economia na alocação de memória - avaliar a diagonal principal da matriz, retornando um valor verdadeiro quando todos seus valores são não-nulos.

```
/*****************/
/*** REFLEXIVA ***/
int is_reflexiva( int** matriz, int size ) {
  int is_it = 1;
  for( int i = 1; i<=size; i++ ) {
    for( int j = 1; j<=size; j++ ) {
       if( i == j ) {
         if( matriz[i][j] == 0 ) {
           is_it = 0;
         }
       }
    }
  }
  printf( "Reflexiva: %s \n", ( is_it == 0 ) ? "F" : "V" );
  if( is_it == 0 ) {
    for( int i = 1; i<=size; i++ ) {
      for( int j = 1; j<=size; j++ ) {
        if( i == j ) {
          if( matriz[i][j] == 0 ) {
            printf( "(%d,%d); ", matriz[i][0], matriz[0][j] );
          }
        }
      }
    }
    printf( "\n" );
  }
  return is_it;
}
/*** REFLEXIVA ***/
/*****************/
```

3.2 Irreflexividade

Novamente, e como todas as demais funções verificadoras de propriedades da matriz que descrevemos nessa seção, a matriz é avaliada estaticamente pela função descrita abaixo, e fornece a saída booleana pedida.

```c
/*******************/
/*** IRREFLEXIVA ***/
int is_irreflexiva( int** matriz, int size ) {
  int is_it = 0;
  for( int i = 1; i<=size; i++ ) {
    for( int j = 1; j<=size; j++ ) {
       if( i == j ) {
         if( matriz[i][j] == 1 ) {
           is_it = 1;
         }
       }
    }
  }
  printf( "Irreflexiva: %s \n", ( is_it == 1 ) ? "F" : "V" );
  if( is_it == 1 ) {
    for( int i = 1; i<=size; i++ ) {
      for( int j = 1; j<=size; j++ ) {
        if( i == j ) {
          if( matriz[i][j] == 1 ) {
            printf( "(%d,%d); ", matriz[i][0], matriz[0][j] );
          }
        }
      }
    }
    printf( "\n" );
  }
  return is_it;
}
/*******************/
/*** IRREFLEXIVA ***/
```

3.3 Simetria

A análise de simetria é feita ao compararmos os valores M_{ixj} com M_{jxi}. Havendo, discrepância entre esses valores, a matriz não é simétrica, conforme ilustrado abaixo:

```c
/*****************/
/*** SIMÉTRICA ***/
int is_simetrica( int** matriz, int size ) {
  int is_it = 1;
  for( int i=1; i<=size; i++ ) {
    for( int j=1; j<=size; j++ ) {
      if( matriz[i][j] && !matriz[j][i] ) {
        is_it = 0;
      }
    }
  }
  printf( "Simétrica: %s \n", ( is_it == 1 ) ? "V" : "F" );
  if( is_it == 0 ) {
    for( int i=1; i<=size; i++ ) {
      for( int j=1; j<=size; j++ ) {
        if( matriz[i][j] && !matriz[j][i] ) {
          printf( "(%d,%d); ", matriz[0][j], matriz[i][0] );
        }
      }
    }
    printf( "\n" );
  }
  return is_it;
}
/*** SIMÉTRICA ***/
/*****************/
```

3.4 Anti-simetria

A anti-simetria consiste em reconhecer os nós simétricos, desconsiderando a diagonal principal. Isso é importante pois, como veremos abaixo, a assimetria é uma propriedade correlata que leva em conta os valores da diagonal principal.

```c
/**********************/
/*** ANTI-SIMÉTRICA ***/
int is_antisimetrica( int** matriz, int size ) {
  int is_it = 0;
  for( int i = 1; i<=size; i++ ) {
    for( int j = 1; j<=size; j++ ) {
      if( ( matriz[i][j] && matriz[j][i] ) && i != j ) {
        is_it = 1;
      }
    }
  }
  printf( "Anti-simétrica: %s \n", ( is_it == 1 ) ? "F" : "V" );
  if( is_it == 1 ) {
    for( int i = 1; i<=size; i++ ) {
      for( int j = 1; j<=size; j++ ) {
        if( i<j ) {
          if( ( matriz[i][j] && matriz[j][i] ) && i != j ) {
              printf( "(%d,%d); ", matriz[i][0], matriz[0][j] );
              printf( "(%d,%d); ", matriz[0][j], matriz[i][0] );
          }
        }
      }
    }
    printf( "\n" );
  }
  return is_it;
}
/*** ANTI-SIMÉTRICA ***/
/**********************/
```

3.5 Assimetria

Como explicado acima, a assimetria é avaliada, levando-se em conta os valores da diagonal principal.

```c
/*******************/
/*** ASSIMÉTRICA ***/
int is_assimetrica( int** matriz, int size ) {
  int is_it = 1;
  for( int i = 1; i<=size; i++ ) {
    for( int j = 1; j<=size; j++ ) {
      if( matriz[i][j] && matriz[j][i] ) {
        is_it = 0;
      }
    }
  }
  printf( "Assimétrica: %s \n", ( is_it == 1 ) ? "V" : "F" );
  return is_it;
}
/*** ASSIMÉTRICA ***/
/*******************/
```

3.6 Transitividade

Por fim, avaliamos a transitividade, função reguladora, podemos já destacar, do limite superior de complexidade assintótica em nosso cógido. A transitividade consiste em reconhecer M_{ixj} e M_{jxz} como *true* e avaliar, da mesma forma, o valor M_{ixz}, que também deverá ser não-nulo.

```
/******************/
/*** TRANSITIVA ***/
int is_transitiva( int** matriz, int size ) {
  int is_it = 1;
  for( int i=1; i<=size; i++ ) {
    for( int j=1; j<=size; j++ ) {
      if( matriz[i][j] && i != j ) {
        for( int k=1; k<=size; k++ ) {
          if( matriz[j][k] && !matriz[i][k] ) {
            is_it = 0;
          }
        }
      }
    }
  }
  printf( "Transitiva: %s \n", ( is_it == 1 ) ? "V" : "F" );
  if( is_it == 0 ) {
    for( int i = 1; i<=size; i++ ) {
      for( int j = 1; j<=size; j++ ) {
        if( matriz[i][j] && i != j ) {
          for( int k=1; k<=size; k++ ) {
            if( matriz[j][k] && !matriz[i][k] ) {
              printf( "(%d,%d); ", matriz[i][0], matriz[0][k] );
            }
          }
        }
      }
    }
    printf( "\n" );
  }
  return is_it;
}
/*** TRANSITIVA ***/
/******************/
```

4 Complexidade

A análise de complexidade é muito importante em Ciência da Computação pois ela permite, ao profissional e usuários dos diversos códigos implementados, compreender os limites de funcionamento de tal código, seja em termos de passos para sua conclusão (custo temporal), como de alocação de memória necessária (custo espacial) para seu devido funcionamento.

4.1 Complexidade Assintótica

A complexidade assintótica de nosso programa está limitada superiormente, como mencionado anteriormente, na função que analisa a transitividade, uma vez que percorre três vezes o conjunto n de dados de entrada. Inferiormente, o limite se mantêm, por se tratar de uma abordagem linear de programação.

Desta maneira, podemos afirmar que possuímos limites assintóticos firmes (pior e melhor caso) que permitem determinarmos que nosso código é $\theta(n^3)$.

4.2 Complexidade de Recursos

A alocação de memória em nosso código é perfeitamente linear. O código não faz uso de funções recursivas e trata seu principal elemento de alocação, qual seja, a matriz de adjacência, de forma estática.

Portanto, podemos dizer que o custo de espaço requerido é linearmente proporcional ao tamanho da entrada de dados fornecida.

5 Conclusão

Através da implementação do código que desenvolvemos, fomos capazes de produzir, a partir de uma entrada padronizada de dados referentes a uma representação de uma relação binária, por meio de grafos diridos, uma matriz de adjacência que fornece um meio bastante útil para a análise das propriedades cabíveis à modelagem dos dados solicitadas pelo problema.

Uma saída, portanto, é fornecida no terminal, composta pelas avaliações necessárias para a análise adequada do problema.

6 Apêndice: Programa Completo para consulta

main.cpp

```
/** Author: Eduardo Capanema ********/
/** Email: eduardocapanema@ufmg.br **/
/** Mat: 2020041515 *****************/

#include <stdlib.h>
#include <stdio.h>

// consts e globais
#define MAX_VAL 50

// cabeçalhos de funções
void relacao_de_fecho( int** matriz, int size );
int is_transitiva( int** matriz, int size );
int is_assimetrica( int** matriz, int size );
int is_antisimetrica( int** matriz, int size );
int is_simetrica( int** matriz, int size );
int is_irreflexiva( int** matriz, int size );
int is_reflexiva( int** matriz, int size );


int main( int argc, char* argv[] ) {

  int n;
  scanf( "%d", &n);
  // printf("n is %d \n", n);

  if( n > MAX_VAL ) {
    printf( "Numero de entradas maior que o limite especificado!" );
    return 0;
  }

  int f[n];
  for ( int k=0; k<n; k++ ) {
    scanf( "%d", &f[k] );
  }

  int** matriz;
  matriz = ( int** ) malloc( sizeof( int* ) * (n+1) );
  for( int m=0; m<=n; m++ ) {
    matriz[m] = ( int* ) malloc( sizeof( int ) * (n+1) );
  }

  // preencher base usando posições 0,0 para delimitar submatriz com ref dos nos
  for( int z=0; z<=n; z++) {
    for( int x=0; x<=n; x++) {
      if( z == 0 && x == 0) {
        matriz[z][x] = 0;
      }
      else if( z == 0 ) {
        matriz[z][x] = f[x-1];
      } else if( x == 0 ) {
        matriz[z][x] = f[z-1];
      } else {
        matriz[z][x] = 0;
      }
    }
  }

  int a, b;
```

```
60    int rc;
61    while ( ( rc = scanf( "%d", &a ) ) == 1 ) {
62      scanf( "%d", &b );
63      int lb = 0;
64      int cb = 0;
65      // percorrer linha 0 e coluna 0
66      for( lb=0; lb<=n; ++lb ) {
67        for( cb=0; cb<=n; ++cb ) {
68          if( matriz[lb][0] == a && matriz[0][cb] == b ) {
69            matriz[lb][cb] = 1;
70          }
71        }
72      }
73    }
74
75    // vendo a matriz - codigo de teste - "descomente" para verificar
76    // printf( "Mostrando a matriz\n" );
77    // for( int v=0; v<=n; v++ ) {
78    //   for( int w=0; w<=n; w++) {
79    //     printf( "%d ", matriz[v][w] );
80    //   }
81    //   printf( "\n" );
82    // }
83
84    int reflexiva = is_reflexiva( matriz, n );
85    int irreflexiva = is_irreflexiva( matriz, n );
86    int simetrica = is_simetrica( matriz, n );
87    int antisimetrica = is_antisimetrica( matriz, n );
88    int assimetrica = is_assimetrica( matriz, n );
89    int transitiva = is_transitiva( matriz, n );
90    printf( "Relação de equivalência: %s \n", ( reflexiva && simetrica && transitiva ) ? "V" : "F" );
91    printf( "Relação de ordem parcial: %s \n", ( reflexiva && transitiva && !antisimetrica ) ? "V" : "F" );
92    relacao_de_fecho( matriz, n );
93
94    // liberar memoria da matriz
95    free( matriz );
96
97    return 0;
98
99  }
100
101
102 /************************
103 Implementações de funções
104 _________________________*/
105
106
107 /************************/
108 /*** RELACAO DE FECHO ***/
109 void relacao_de_fecho( int** matriz, int size ) {
110   printf( "Fecho transitivo da relação: " );
111   for( int i=1; i<=size; i++ ) {
112     for( int j=1; j<=size; j++ ) {
113       if( matriz[j][i] ) {
114         if( i>=j ) {
115         /* OBS: para nossa análise
116         de complexidade assintótica,
117         este é nosso limite superior - O(n³) */
118           for( int k=1; k<=size; k++ ) {
119             if( matriz[i][k] ) {
120               printf( "(%d,%d); ", matriz[i][0], matriz[0][k] );
121             }
122           }
```

```
123           }
124         }
125       }
126     }
127     printf( "\n" );
128     return;
129   }
130   /*** RELACAO DE FECHO ***/
131   /************************/
132
133   /******************/
134   /*** TRANSITIVA ***/
135   int is_transitiva( int** matriz, int size ) {
136     int is_it = 1;
137     for( int i=1; i<=size; i++ ) {
138       for( int j=1; j<=size; j++ ) {
139         if( matriz[i][j] && i != j ) {
140           for( int k=1; k<=size; k++ ) {
141             if( matriz[j][k] && !matriz[i][k] ) {
142               is_it = 0;
143             }
144           }
145         }
146       }
147     }
148     printf( "Transitiva: %s \n", ( is_it == 1 ) ? "V" : "F" );
149     if( is_it == 0 ) {
150       for( int i = 1; i<=size; i++ ) {
151         for( int j = 1; j<=size; j++ ) {
152           if( matriz[i][j] && i != j ) {
153             for( int k=1; k<=size; k++ ) {
154               if( matriz[j][k] && !matriz[i][k] ) {
155                 printf( "(%d,%d); ", matriz[i][0], matriz[0][k] );
156               }
157             }
158           }
159         }
160       }
161       printf( "\n" );
162     }
163     return is_it;
164   }
165   /*** TRANSITIVA ***/
166   /******************/
167
168   /********************/
169   /*** ASSIMÉTRICA ***/
170   int is_assimetrica( int** matriz, int size ) {
171     int is_it = 1;
172     for( int i = 1; i<=size; i++ ) {
173       for( int j = 1; j<=size; j++ ) {
174         if( matriz[i][j] && matriz[j][i] ) {
175           is_it = 0;
176         }
177       }
178     }
179     printf( "Assimétrica: %s \n", ( is_it == 1 ) ? "V" : "F" );
180     return is_it;
181   }
182   /*** ASSIMÉTRICA ***/
183   /********************/
184
185   /**********************/
```

```
186  /*** ANTI-SIMÉTRICA ***/
187  int is_antisimetrica( int** matriz, int size ) {
188    int is_it = 0;
189    for( int i = 1; i<=size; i++ ) {
190      for( int j = 1; j<=size; j++ ) {
191        if( ( matriz[i][j] && matriz[j][i] ) && i != j ) {
192          is_it = 1;
193        }
194      }
195    }
196    printf( "Anti-simétrica: %s \n", ( is_it == 1 ) ? "F" : "V" );
197    if( is_it == 1 ) {
198      for( int i = 1; i<=size; i++ ) {
199        for( int j = 1; j<=size; j++ ) {
200          if( i<j ) {
201            if( ( matriz[i][j] && matriz[j][i] ) && i != j ) {
202                printf( "(%d,%d); ", matriz[i][0], matriz[0][j] );
203                printf( "(%d,%d); ", matriz[0][j], matriz[i][0] );
204            }
205          }
206        }
207      }
208      printf( "\n" );
209    }
210    return is_it;
211  }
212  /*** ANTI-SIMÉTRICA ***/
213  /**********************/
214
215  /*****************/
216  /*** SIMÉTRICA ***/
217  int is_simetrica( int** matriz, int size ) {
218    int is_it = 1;
219    for( int i=1; i<=size; i++ ) {
220      for( int j=1; j<=size; j++ ) {
221        if( matriz[i][j] && !matriz[j][i] ) {
222          is_it = 0;
223        }
224      }
225    }
226    printf( "Simétrica: %s \n", ( is_it == 1 ) ? "V" : "F" );
227    if( is_it == 0 ) {
228      for( int i=1; i<=size; i++ ) {
229        for( int j=1; j<=size; j++ ) {
230          if( matriz[i][j] && !matriz[j][i] ) {
231            printf( "(%d,%d); ", matriz[0][j], matriz[i][0] );
232          }
233        }
234      }
235      printf( "\n" );
236    }
237    return is_it;
238  }
239  /*** SIMÉTRICA ***/
240  /*****************/
241
242  /*******************/
243  /*** IRREFLEXIVA ***/
244  int is_irreflexiva( int** matriz, int size ) {
245    int is_it = 0;
246    for( int i = 1; i<=size; i++ ) {
247      for( int j = 1; j<=size; j++ ) {
248         if( i == j ) {
```

```
249           if( matriz[i][j] == 1 ) {
250             is_it = 1;
251           }
252         }
253       }
254     }
255     printf( "Irreflexiva: %s \n", ( is_it == 1 ) ? "F" : "V" );
256     if( is_it == 1 ) {
257       for( int i = 1; i<=size; i++ ) {
258         for( int j = 1; j<=size; j++ ) {
259           if( i == j ) {
260             if( matriz[i][j] == 1 ) {
261               printf( "(%d,%d); ", matriz[i][0], matriz[0][j] );
262             }
263           }
264         }
265       }
266       printf( "\n" );
267     }
268     return is_it;
269   }
270   /*******************/
271   /*** IRREFLEXIVA ***/
272
273   /*****************/
274   /*** REFLEXIVA ***/
275   int is_reflexiva( int** matriz, int size ) {
276     int is_it = 1;
277     for( int i = 1; i<=size; i++ ) {
278       for( int j = 1; j<=size; j++ ) {
279          if( i == j ) {
280            if( matriz[i][j] == 0 ) {
281              is_it = 0;
282            }
283          }
284       }
285     }
286     printf( "Reflexiva: %s \n", ( is_it == 0 ) ? "F" : "V" );
287     if( is_it == 0 ) {
288       for( int i = 1; i<=size; i++ ) {
289         for( int j = 1; j<=size; j++ ) {
290           if( i == j ) {
291             if( matriz[i][j] == 0 ) {
292               printf( "(%d,%d); ", matriz[i][0], matriz[0][j] );
293             }
294           }
295         }
296       }
297       printf( "\n" );
298     }
299     return is_it;
300   }
301   /*** REFLEXIVA ***/
302   /*****************/
```