Technical Report CS82013

An Algorithm for Generating the Set of Fundamental Cycles in a Graph

December, 1982

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061
An Algorithm for Generating the Set of Fundamental Cycles in a Graph

DESCRIPTION

The set of PASCAL procedures described here represent an extension of heuristic algorithms [1] for generating fundamental cycles with minimum total length:

\[ L(T) = \sum_{i=1}^{u} l_i \]

where \( l_i \) is the length of the \( i^{th} \) fundamental cycle and is the nullity of the graph. The nullity of a connected graph \( G \) with \( n \) vertices and \( e \) edges is given by \( \nu = e - u + 1 \).

The algorithm [2], called MBFSLOOK, is of hybrid nature. It consists of a multiple breadth first search, where we always explore from the vertex of highest degree with respect to "unexplored" edges, and a set of rules for selecting, in case two or more vertices have the same highest degree, the vertex to explore from.

MBFSLOOK uses an \( n \times n \) adjacency matrix to represent the graph, where \( n \) is the number of vertices and the matrix entry


\( (V_i, V_j) \) equals one if and only if there is an edge between vertex \( v_i \) and vertex \( v_j \). Once the structure has been defined, the algorithm determines the vertex of highest degree. Unless there is a tie, the algorithm generates a tree in a straight-forward breadth first fashion [3] and decreases the degree of the vertices visited by one. MBFSLOOK then considers a new vertex to explore from, thus building a forest as it goes along. The individual trees become eventually connected, extending the partial spanning tree or forming a fundamental cycle. The algorithm terminates when all vertices in \( G \) have been visited. So far we have assumed that only one vertex of highest degree was found everytime we determined the new vertex to explore from. MBFSLOOK calls the procedure LOOK everytime we encounter the situation where two or more edges are of the same highest degree. This procedure computes the combined degree of all the vertices that would be visited next if we were to start exploring from each vertex of equal highest degree. The so obtained cumulative degrees are then compared and ranked in an increased order. If the resultant turned out to be a single vertex, it becomes the selected one. If a tie remains after this step, the algorithm proceeds to another level, namely computing the combined degree of the vertices adjacent to each of the vertices in the previous level. Procedure LOOK terminates when either a tie is broken or all vertices have been visited (in which case it arbitrarily selects the new vertex to explore from).
One of the features of the MBFSLOOK algorithm is its performance in relation to other algorithms for generating fundamental cycles [2] [4]. In addition, the internal logic of the improved algorithm is based exclusively on an attribute of G, namely the degree of the vertices and not on the arbitrary labels used in identifying vertices in the graph. BCG [5], for instance, has an internal logic whose outcome is contingent upon the initial labeling of the vertices in the graph. This means we could have as many different sets of fundamental cycles as labeling combinations are possible. This high degree of arbitrariness is removed by MBFSLOOK through an internal logic based solely on the degree of the vertices.

The MBFSLOOK package is comprised of the following procedures:

MAIN PROGRAM: This procedure initializes the adjacency matrix and the various counters used in the package. It determines the vertex with the highest degree, and ascertains how many there are. The edges so created form a new tree, becoming part of the spanning tree or forming a fundamental cycle.

ADD_EDGE_TO_TREE: Is the procedure called to add an edge to the spanning tree of the graph. The spanning tree is represented by one way adjacency lists.
BUILD_E_TREE_ARRAY: Is the procedure that is called once a fundamental cycle has been found and its path in the spanning tree traced, from the vertex of the back edge to the second vertex of the back edge.

DEFINE_VERTICES_TO_FOCUS_ON: This procedure defines those vertices which should be examined when trying to break a tie between high degree vertices.

LOOK-AHEAD: Is the procedure that, when there is more than one high degree vertex, breaks the tie by determining the high degree vertex with the degrees of the adjacent vertices. If after the first pass no such vertex exists, i.e., the tie has not been broken, the LOOK-A-HEAD procedure is called again, until the tie is broken or no more vertices remain unvisited.

These five procedures must be used together, where the interface program is made up by procedures ADD_EDGE_TO_TREE and BUILD_E_TREE_ARRAY. All the procedures have been tested extensively on the IBM 370/158 under VM/SP Conversational Monitor System and on the VAX 11-780 under VMS at Virginia Polytechnic Institute and State University.
REFERENCES


APPENDIX

Complete listing of the MBFSLOOK Algorithm Implementation.
PRLOGUE

TITLE: MBSLOHA - IDENTIFIES FUNDAMENTAL CYCLES BY USING THE MULTIPONT BREADTH FIRST SEARCH (MBS) ALGORITHM ON A RANDOM GRAPH REPRESENTED BY AN ADJACENCY MATRIX. WHEN THERE IS A MORE THAN ONE HIGH DEGREE VERTEX, A LOOK AHEAD PROCEDURE IS CALLED WHICH TRIES TO BREAK THE TIE BY DETERMINING THE HIGH DEGREE VERTEX WHOSE ADJACENT VERTECIES ARE OF HIGHER DEGREE.

DESCRIPTION: THE MULTIPONT BREADTH FIRST SEARCH (MBS) ALGORITHM CHOOSES THE HIGHEST DEGREE VERTEX TO VISIT. IF THERE IS MORE THAN ONE VERTEX OF THE HIGHEST DEGREE THEN THE VERTEX TO VISIT IS CHOSEN ARBITRARILY FROM AMONG THESE HIGHEST DEGREE VERTECIES.

THE GRAPH REPRESENTATION IS AN N X N ADJACENCY MATRIX WHERE N IS THE NUMBER OF VERTECIES IN THE GRAPH. THE ADJACENCY MATRIX IS ALLOCATED IN A STATIC MANNER.

INPUT: THE ADJACENCY MATRIX WHICH REPRESENTS THE GRAPH IS INPUT TO THIS PROGRAM FROM A DISK FILE. THE NAME OF THE DISK FILE IS ALGOMATR.

OUTPUT: THIS PROGRAM OUTPUTS THE COMPONENTS OF EACH FUNDAMENTAL CYCLE; THE NUMBER OF FUNDAMENTAL CYCLES FOUND, AND THE AVERAGE LENGTH OF THE SET OF FUNDAMENTAL CYCLES.

INTERFACE: THIS PROGRAM CALLS TWO PROCEDURES.

THE FIRST PROCEDURE CALLED IS ADD_EDGE TO TREE. THE SECOND PROCEDURE CALLED IS BUILD_TREE ARRAY. THIS PROCEDURE IS CALLED TO ADD AN EDGE TO THE SPANNING TREE OF THE GRAPH. THE SPANNING TREE IS REPRESENTED BY ONE WAY ADJACENCY LISTS.

THE SECOND PROCEDURE CALLED IS BUILD_TREE ARRAY. THE SPANNING TREE FROM THE FIRST VERTEX OF THE BACK EDGE TO THE SECOND VERTEX OF THE BACK EDGE.

A THIRD PROCEDURE IS DEFINE_VERTENCIES TO FOCUS ON. THIS PROCEDURE DEFINES THOSE VERTENCIES WHICH SHOULD BE EXAMINED WHEN TRYING TO BREAK A TIE BETWEEN HIGH DEGREE VERTENCIES.
*  

LABEL

10, 99:

CONST

MAX_N = 45;
MAX_SUBTREES = 20;
N_PTYS.ONE = 46;
N_MINUS_ONE = 44;
MAX_CYCLES = 21;
PRECISION = 4;  (* NUMBER OF DECIMAL DIGITS *)

TYPE

VERTEX = 0, MAX_N;
SUB_TREE_COUNT = 0, MAX_SUBTREES;
CYCLE_RANGE = 0, MAX_CYCLES;
CYCLE_SUMS = ARRAY[1..MAX_CYCLES] OF VERTEX;
VERTEX_ARRAY = ARRAY[1..MAX_SUBTREES, 1..MAX_N] OF VERTEX;
VERTEX_ALL_ARRAY = ARRAY[1..MAX_SUBTREES, 1..MAX_N] OF VERTEX;
ADJ_MATRIX = ARRAY[1..MAX_N, 1..MAX_N] OF VERTEX;
VERTEX_VISITED_ARRAY = ARRAY[1..MAX_N] OF BOOLEAN;
VERTEX_PLUS.ONE = 1, N_PLUS.ONE;

NODE_POINTER = ^VERTEX_NODE;

VERTEX_NODE = RECORD
  V_INFO : VERTEX;  (* ADJACENCY LISTS FOR *)
  V_LINK : NODE_POINTER;  (* THE SPANNING TREE *)
END;

FIRST_POINTER = ARRAY[1..MAX_N] OF NODE_POINTER;
ARRAY_CYCLE_SWITCH = ARRAY[1..MAX_N] OF BOOLEAN;
INTEGER_VERTEX_ARRAY = ARRAY[1..MAX_N] OF INTEGER;

VAR

V_TREE : VERTEX_ARRAY;
TRE_TREE : VERTEX_ALL_ARRAY;
GLOBAL_TREE : VERTEX_ALL_ARRAY;
SUM_ARRAY : VERTEX_ALL_ARRAY;
TOP_VI = VERTEX.PLUS.ONE;
N, V, W, Z, V, HIGH, V, TEMP, V, VERTEX;
MATRIX_A : ADJ_MATRIX;
INDEX_E3, DEG, MAX_NI, DEG, V, VERTEX;
SAME_HIGH, DEG, V, VERTEX_ALL_ARRAY;
TOP_BAS, B, T, BASIC_COUNT, CYCLE_RANGE;
CYCLES_LENGTH, CYCLE_RANGE;
SAVE_CYCLE_LENGTH, CYCLE_SUMS;
TOTAL_CYCLE_LENGTH, INTEGER;
AVE_CYCLE_LENGTH, REAL;
ROOT_INSPECT : BOOLEAN;  (* INDICATES YOU HAVE FOUND VERTEX V.Z *)
FOUND_V_Z : BOOLEAN;  (* IN A PREVIOUS SUBTREE *)
FOUND_V_W : BOOLEAN;  (* INDICATES YOU HAVE FOUND VERTEX V.W *)
EDGE_COMPLETE : BOOLEAN;  (* IN A PREVIOUS SUBTREE *)
FOUND_CYCLE : BOOLEAN;

-8-
PROCEDURE ADD_EDGE_TO_TREE (LOC1_V_Z, LOC1_V_W : VERTEX;
VAR GL01_TOP_ADJ : FIRST_POINTER);

VAR
ADJ_PTR, LOC1_TEMP_PTR : NODE_POINTER;
BEGIN
(* ALLOCATE A NEW NODE FOR THE SPANNING TREE ADJACENCY
LIST FOR VERTEX LOC1_V_Z *)
NEW (ADJ_PTR);
ADJ_PTR^V_INFO := LOC1_V_W;
IF (GL01_TOP_ADJ[LOC1_V_Z] = NIL)
THEN
BEGIN
GL01_TOP_ADJ[LOC1_V_Z] := ADJ_PTR;
ADJ_PTR^V_LINK := NIL
END
ELSE
BEGIN
(* SEARCH THE ADJACENCY LIST FOR THE LAST NODE *)
(* SPECIFICALLY, SEARCH FOR THE NODE IN THE LIST *)
(* WITH A NIL VALUE IN THE LINK FIELD OF THE NODE *)
LOC1_TEMP_PTR := GL01_TOP_ADJ[LOC1_V_Z];
WHILE (LOC1_TEMP_PTR^V_LINK <> NIL) DO
LOC1_TEMP_PTR := LOC1_TEMP_PTR^V_LINK;
(* LOC1_TEMP_PTR NOW POINTS TO THE LAST NODE IN THE
ADJACENCY LIST *)
LOC1_TEMP_PTR^V_LINK := ADJ_PTR;
ADJ_PTR^V_LINK := NIL
END;
BEGIN
(* ALLOCATE A NEW NODE FOR THE SPANNING TREE ADJACENCY
LIST FOR VERTEX LOC1_V_W *)
NEW (ADJ_PTR);
ADJ_PTR^V_INFO := LOC1_V_Z;
IF (GL01_TOP_ADJ[LOC1_V_W] = NIL)
THEN
BEGIN
GL01_TOP_ADJ[LOC1_V_W] := ADJ_PTR;
ADJ_PTR^V_LINK := NIL
END
ELSE
BEGIN
(* SEARCH THE ADJACENCY LIST FOR THE LAST NODE *)
(* SPECIFICALLY, SEARCH FOR THE NODE IN THE LIST *)
(* WITH A NIL VALUE IN THE LINK FIELD OF THE NODE *)
LOC1_TEMP_PTR := GL01_TOP_ADJ[LOC1_V_W];
WHILE (LOC1_TEMP_PTR^V_LINK <> NIL) DO
LOC1_TEMP_PTR := LOC1_TEMP_PTR^V_LINK;
(* LOC1_TEMP_PTR NOW POINTS TO THE LAST NODE IN THE
ADJACENCY LIST *)
LOC1_TEMP_PTR^V_LINK := ADJ_PTR;
ADJ_PTR^V_LINK := NIL
END
END (* PROCEDURE ADD_EDGE_TO_TREE *)
PROCEDURE BUILD_E_TREE_ARRAY ( LOC2_V_Z, LOC2_V_W : VERTEX;
   VR2_E_T VERTEX_STACK: FIRST_POINTER;
   VR2_E_TREE_ARRAY : VERTEX_ALL_ARRAY )

VAR
   INITIAL_V_Z, LOC1_VERTEX, LOC2_V_TEMP : VERTEX;
   VR2_E_T VERTEX_STACK : VERTEX_ALL_ARRAY;
   VR3_TEMP_VERTEX : NODE_POINTER;

BEGIN
   (* SAVE THE INITIAL V.Z VALUE *)
   INITIAL_V_Z := LOC2_V_Z;
   (* PUSH THE VALUE 0 ONTO THE VR2_E_T VERTEX_STACK *)
   LOC1_VERTEX := 0;
   VR2_E_T VERTEX_STACK[LOC1_VERTEX] := LOC2_V_Z;
   (* PUSH THE STARTING VERTEX INTO THE VR2_E_T VERTEX_STACK *)
   LOC1_VERTEX := LOC1_VERTEX + 1;
   VR2_E_T VERTEX_STACK[LOC1_VERTEX] := LOC2_V_Z;
   (* WHILE THE UNEXAMINED VERTEX STACK IS NOT EMPTY DO *)
   WHILE ((VR2_E_T VERTEX_STACK[LOC1_VERTEX] <> 0 )
      AND (VR2_E_TREE_ARRAY[LOC2_V_W] = 0))
   DO
      BEGIN
      (* FOR THE VR2_E_T VERTEX STACK *)
      LOC1_VERTEX := VR2_E_T VERTEX_STACK[LOC1_VERTEX];
      LOC1_VERTEX := LOC1_VERTEX - 1;
      (* START WITH LOC2_V_Z, CHECK WHAT IS CONNECTED TO IT *)
      VR3_TEMP_VERTEX := VR2_E_TREE_ARRAY[LOC2_V_W];
      WHILE ((VR3_TEMP_VERTEX <> NIL )
      AND (VR2_E_TREE_ARRAY[LOC2_V_W] = 0))
      DO
         BEGIN
         (* ASCERTAIN THE FIRST EDGE IN THE SPANNING TREE *)
         VR3_TEMP := VR3_TEMP_VERTEX^V_INFO;
      IF (VR2_E_TREE_ARRAY[LOC2_V_Z] <> LOC3_TEMP)
      THEN
         BEGIN
         VR2_E_TREE_ARRAY[LOC3_TEMP] := LOC2_V_Z;
         (* PUSH LOC2_V_Z ON TO THE VR2_E_T VERTEX STACK *)
         LOC1_VERTEX := LOC1_VERTEX + 1;
         VR2_E_T VERTEX_STACK[LOC1_VERTEX] := LOC3_TEMP;
         END
         (* RESET THE TOP_POINTER FOR THE ADJACENCY LIST *)
         VR3_TEMP_VERTEX := VR3_TEMP_VERTEX^V_LINK
      END
      END
   END; (* PROCEDURE BUILD_E_TREE_ARRAY *)

PROCEDURE LOOK_AHEAD ( VR2_VERTEX ;
   VR2_EQ_HIGH : VERTEX_ALL_ARRAY )

VAR
   TEMP, JUNK, JUNKY, ASDF, JKL : VERTEX;
   VR2_EQ_DEG, VR2_MAX_DEG, VR2_CON : VERTEX;
   VR3_CUM_SUP_DEG : INTEGER_VERTEX_ARRAY;
PROCEDURE DEFINE_VERTICES_TO_FOCUS_ON
  (VAR GLO1_COUNT U V: VERTEX;
   VAR GLO1_MAX_DEX, V_CON: VERTEX);

VAR TEMP, JUNK, ASDF, COL_INDEX : VERTEX;
TEST_V_UNIQUE, DEX_EQ: VERTEX_ALL_ARRAY;
FOUND : BOOLEAN;

BEGIN (* DEFINE_VERTICES_TO_FOCUS_ON *)

  (* INITIALIZE AN ARRAY THAT TESTS THE UNIQUENESS OF VERTICES *)
  (* DETERMINE THE NON COMMON VERTICES *)
  (* INITIALIZE COUNTERS FOR THE COLUMN INDEX OF THE 
     V_CONSIDER ARRAY *)
  MAX_DEX, V_CON := 0;
  FOR TEMP := 1 TO N DO 
    DEX_EQ[TMP] := 0;
  (* TEST FOR UNCOMMON VERTICES *)
  FOR TEMP := 1 TO N DO
    BEGIN
      FOR ASDF := 1 TO INDEX_EQ_DEG DO
        (* FOR EACH HIGH DEGREE VERTEX IF THE 
           VERTEX IS IN V_VISITED ARRAY FOR THE HIGH DEGREE 
           VERTEX THEN PLACE THE VERTEX IN THE 
           GLO1 V_CONSIDER ARRAY FOR 
           THE SPECIFIC HIGH DEGREE VERTEX *)
        BEGIN
          FOUND := FALSE;
          IF (HAVE_CONSIDERED( ASDF, TEMP ) = FALSE) THEN
            BEGIN
              JUNK := 1;
              WHILE ((JUNK := MAX_VISIT( ASDF ) )
                AND (FOUND = FALSE)) DO 
                BEGIN
                  IF (V_VISITED( ASDF, JUNK ) = TEMP) THEN
                    BEGIN
                      FOUND := TRUE;
                      (* ADD THIS UNIQUE VERTEX TO THE 
                         V_CONSIDER ARRAY FOR HIGH DEGREE 
                         VERTEX ASDF *)
                      DEX_EQ[ ASDF ] := DEX_EQ[ ASDF ] + 1;
                      GLO1_V_CONSIDER[ASDF, DEX_EQ[ASDF] ] := TEMP;
                      (* SET THE MAXIMUM COLUMN INDEX FOR THE 
                         V_CONSIDER ARRAY *)
                      IF (DEX_EQ[ ASDF ] > GLO1_MAX_DEX, V_CON)
                      END
                      END
                    END
                  END
                END
            END
          END
        END
      END
    END
  END
END
THEN
  GLOBAL_MAX_DEG := DEX_EG[ASDF] 1
END (* IF *)
JUNK := JUNK + 1
END (* WHILE *)
END (* IF (HAVE_CONSIDERED *)
END (* FOR ASDF := 1 TO INDEX_EQ_DEG *)
IF (FOUND)
  THEN
    GLOBAL_COUNT_U_V := GLOBAL_COUNT_U_V + 1
END (* FOR TEMP := 1 TO N *)
(* AT THIS POINT THE VARIABLE GLOBAL_COUNT_U_V CONTAINS THE *
NUMBER OF VERTICES THAT ARE TO BE CONSIDERED *)
END (* PROCEDE D VERTICES_TO_FOCUS_ON *)
BEGIN (* PROCEDURE LOCK_AHEAD *)
(* INITIALIZE THE SUM DEG ARRAY *
FOR THE HIGH DEGREE VERTICES *)
FOR JUNK := 1 TO INDEX_EQ_DEG DO
  SUM_EQ_DEG[JUNK] := 0;
(* FOR EACH HIGH DEGREE VERTEX *
SET AN ARRAY COMPONENT TO THE MAX NUMBER OF ENTRIES IN *
THE CORRESPONDING ENTRY IN THE V_VISITED ARRAY *)
(* INITIALLY THIS NUMBER IS THE DEGREE OF THE TIED *
VERTICES PLUS ONE *)
FOR C_EQ_DEG := 1 TO INDEX_EQ_DEG DO
  MAX_VISIT_C_EQ_DEG := SUM_EQ_DEG[C_EQ_DEG] + 1;
(* DEFINE FOR EACH VERTEX WHICH VERTICES HAVE ALREADY *
BEEN VISITED. THESE VERTICES ARE STORED IN THE *
V_VISITED ARRAY *)
(* FOR ALL HIGH DEGREE VERTICES EXECUTE THE FOLLOWING *
CODE *)
FOR C_EQ_DEG := 1 TO INDEX_EQ_DEG DO
BEGIN
  JUNK := 1;
  (* PLACE THE ORIGINAL HIGH DEGREE VERTEX ON THE *
LIST OF VERTICES ALREADY CONSIDERED *)
  V_VISITED[C_EQ_DEG, JUNK] :=
  EQ_HIGH_DEG[C_EQ_DEG] 1;
  (* INITIALIZE THE REST OF THE V_VISITED ARRAY *)
  FOR ASDF := 2 TO MAX_VISIT_C_EQ_DEG DO
    V_VISITED[C_EQ_DEG, ASDF, JUNK] := 0;
  (* INITIALIZE THE V VISITED ARRAY COMPONENT FOR *
THE HIGH DEGREE VERTEX EQ_HIGH_DEG[C_EQ_DEG] *)
  (* PUT IN THE V_VISITED ARRAY THOSE VERTICES *
THAT ARE ADJACENT TO THE HIGH DEGREE VERTEX *)
  FOR TEMP := 1 TO N DO
    IF (MATRIX_AL [TEMP, EQ_HIGH_DEG[C_EQ_DEG] 1] = 1)
    THEN
      BEGIN
      (* ADD THE DEGREE OF THIS ADJACENT VERTEX TO *
THE DEGREE OF THE CURRENT VERTEX *)
      SUM_EQ_DEG[C_EQ_DEG] :=
      SUM_EQ_DEG[C_EQ_DEG] +
      SUM_ARR [ROW [TEMP, TEMP]];
      (* PLACE THE ADJACENT VERTEX ON THE LIST OF

-12-
VERTICES ALREADY CONSIDERED *)
JUNK := JUNK + 1;
V_VISITED[C_EQ_DEG] := TEMP
END;
(* FOR C_EQ_DEG := 1 TO INDEX_EQ_DEG *)

(* COMPARE THE RESULTING DEGREE SUMS FOR THE HIGHEST DEGREE VERTICES AND REDEFINE THE SET OF HIGHEST DEGREE VERTICES *)
C_EQ_DEG := 1;
FOR TEMP := 1 TO (INDEX_EQ_DEG - 1) DO
IF (C_EQ_DEG [C_EQ_DEG] < C_EQ_DEG [TEMP + 1])
THEN
BEGIN
(* RESET THE COUNT OF HIGHEST DEGREE VERTICES TO ONE *)
C_EQ_DEG := 1;
(* RESET THE VALUE OF C_EQ_DEG ARRAY INDEX *)
C_EQ_DEG [C_EQ_DEG] := C_EQ_DEG [TEMP + 1];
(* RESET THE HIGHEST DEGREE VERTEX *)
EQ_HIGH_DEG [C_EQ_DEG] := EQ_HIGH_DEG [TEMP + 1];
(* RESET THE VERTEX VISIT ARRAY ROW *)
FOR JUNK := 1 TO MAX_VISIT (TEMP + 1) DO
V_VISITED [C_EQ_DEG] := V_VISITED [TEMP + 1];
END
ELSE
IF (C_EQ_DEG [C_EQ_DEG] = C_EQ_DEG [TEMP + 1])
THEN
BEGIN
C_EQ_DEG := C_EQ_DEG + 1;
(* RESET THE HIGHEST DEGREE VERTEX *)
IF (C_EQ_DEG <> (TEMP + 1))
THEN
BEGIN
EQ_HIGH_DEG [C_EQ_DEG] := EQ_HIGH_DEG [TEMP + 1];
(* RESET VALUE OF C_EQ_DEG ARRAY INDEX *)
C_EQ_DEG [C_EQ_DEG] := C_EQ_DEG [TEMP + 1];
(* RESET VERTEX VISIT ARRAY ROW *)
FOR JUNK := 1 TO MAX_VISIT (TEMP + 1) DO
V_VISITED [C_EQ_DEG] := V_VISITED [TEMP + 1];
END
END (* IF C_EQ_DEG <> (TEMP + 1) ) *)

END (* C_EQ_DEG, VERTICES OF THE HIGHEST DEGREE EXIST *)
INDEX_EQ_DEG := C_EQ_DEG;

(* INITIALIZE THE COUNT OF UNIQUE VERTICES *)
COUNT_UNIQUE := 0;
(* IF THERE IS MORE THAN ONE HIGH DEGREE VERTEX THEN CALL THE DEFINE VERTICES TO FOCUS ON PROCEDURE WHICH ASCERTAINS THE UNIQUE VERTICES CONNECTED TO EACH HIGH DEGREE VERTEX *)
IF (INDEX_EQ_DEG > 1) AND (INDEX_EQ_DEG < MAX_NZ_DEG_V)
THEN
BEGIN
(* INITIALIZE THE HAVE CONSIDERED ARRAY *)
FOR ASDF := 1 TO INDEX_EQ_DEG DO
FOR TEMP := 1 TO N DO
IF (TEMP = EQ_HIGH_DEG [ASDF])
THEN
HAVE_CONSIDERED ASDF; TEMP J := TRUE
ELSE
    HAVE_CONSIDERED ASDF; TEMP J := FALSE;

(* CALL THE DEFINE_VERTICES_TO_FOCUS_ON PROCEDURE *)
DEFINE_VERTICES_TO_FOCUS_ON(COUNT_UNIQUE_V, V_CONSIDER,
    MAX_DEX_V_CON)
END; (* IF *)

DEGREE_SUM_CHANGE := TRUE;
WHILE (INDEX_EQ_DEG > 1) AND (INDEX_EQ_DEG < MAX_EQ_DEG_V)
AND (COUNT_UNIQUE_V > 0) AND (DEGREE_SUM_CHANGE) DO
    BEGIN
        (* INITIALIZE DEGREE_SUM_CHANGE TO FALSE *)
        DEGREE_SUM_CHANGE := FALSE;
        (* FOR ALL HIGH DEGREE VERTICES EXECUTE THE FOLLOWING CODE *)
        FOR C_EQ_DEG := 1 TO INDEX_EQ_DEG DO
            (* FOR ALL VERTICES IN V_CONSIDER ARRAY FOR THE CURRENT *
            HIGH DEGREE VERTEX DO THE FOLLOWING *)
            FOR JUNKY := 1 TO MAX_DEX_V_CON DO
                BEGIN
                    (* FOCUS ON VERTEX X V_CONSIDER(C_EQ_DEG, JUNKY) *
                    IN RELATION TO VERTEX X HIGH DEG C_EQ_DEG J *)
                    (* SET THE FLAG ON FOR HAVING CONSIDERED VERTEX JUNKY *
                    FOR HIGH DEGREE VERTEX X HIGH DEG C_EQ_DEG J *)
                    HAVE_CONSIDERED(C_EQ_DEG, V_CONSIDER(C_EQ_DEG, JUNKY J) :=
                    TRUE);

                    (* AFTER THE IMMEDIATE PROCESSING BELOW YOU SHOULD NEVER *
                    AGAIN CONSIDER VERTEX X V_CONSIDER(C_EQ_DEG, JUNKY) *
                    FOR EQUAL HIGH DEGREE VERTEX *
                    EQ_HIGH_DEG(C_EQ_DEG) *)
                    FOR TEMP := 1 TO N DO
                        BEGIN
                            (* TEST IF VERTEX TEMP IS ADJACENT TO VERTEX *
                            V_CONSIDER(C_EQ_DEG, JUNKY) *)
                            IF MATRIX_AT TEMP, V_CONSIDER(C_EQ_DEG, JUNKY) := 1
                                THEN
                                    (* CHECK IF THE ADJACENT VERTEX HAS ALREADY BEEN *
                                    VISITED *)
                                    BEGIN
                                        PROCEED := TRUE;
                                        FOR ASDF := 1 TO MAX_VISITED(C_EQ_DEG) DO
                                            IF (TEMP = V_VISITED(C_EQ_DEG, ASDF) )
                                                THEN
                                                    (* VERTEX TEMP IS ALREADY IN V_VISITED *
                                                    ARRAY FOR HI DEGREE VERTEX *
                                                    EQ_HIGH_DEG(C_EQ_DEG) *)
                                                    PROCEED := FALSE;
                                            IF (PROCEED)
                                                THEN
                                                    (* VERTEX TEMP IS NOT IN V_VISITED ARRAY FOR *
                                                    HIGH DEGREE VERTEX *
                                                    EQ_HIGH_DEG(C_EQ_DEG) *)
                                                    CUM_SUM_DEG(C_EQ_DEG) :=
                                                    CUM_SUM_DEG(C_EQ_DEG) +
                        END;
                    END;
                END;
            END;
        END;
    END;
END.
SUM_ARR_ROW[ TEMP ] :=
(* THE VALUE SUM_ARR_ROW[ TEMP ] WAS ADDED TO THE SUM DEGREE OF VERTEX EG_HIGHEST_DEG[C_EQ_DEG] *)
IF (DEGREE_SUM_CHANGE = FALSE)
THEN
DEGREE_SUM_CHANGE := TRUE;
(* PLACE THE ADJACENT VERTEX OF THE LIST OF VERTICES ALREADY CONSIDERED BUT YOU WANT TO CONSIDER THE ADJACENT VERTICES TO *)
MAX_VISIT[C_EQ_DEG] := MAX_VISIT[C_EQ_DEG] + 1;
V_VISITED[C_EQ_DEG][MAX_VISIT[C_EQ_DEG]] := TEMP
END (* IF (PROCEED) *)
END (* FOR TEMP *)
END (* FOR JUNK *)
(* DEGREE SUM FOR VERTEX EG_HIGHEST_DEG[C_EQ_DEG] IS CUM_SUM_DEG[C_EQ_DEG] *)
IF (DEGREE_SUM_CHANGE = TRUE)
THEN
BEGIN
(* COMPARE THE RESULTING DEGREE SUMS FOR THE HIGHEST DEGREE VERTICES AND REDEFINE THE SET OF HIGHEST DEGREE VERTICES *)
C_EQ_DEG := 1;
FOR TEMP := 1 TO (INDEX_EQ_DEG - 1) DO
IF (CUM_SUM_DEG[C_EQ_DEG] < CUM_SUM_DEG[TEMP + 1])
THEN
BEGIN
(* RESET THE COUNT OF HIGHEST DEGREE VERTICES TO ONE *)
C_EQ_DEG := 1;
(* RESET THE VALUE OF CUM_SUM_DEG ARRAY INDEX *)
CUM_SUM_DEG[C_EQ_DEG] := CUM_SUM_DEG[TEMP + 1];
(* RESET THE HIGHEST DEGREE VERTEX *)
EG_HIGHEST_DEG[C_EQ_DEG] := EG_HIGHEST_DEG[TEMP + 1];
(* RESET THE VERTEX VISIT ARRAY ROW *)
FOR JUNK := 1 TO N DO
V_VISITED[C_EQ_DEG, JUNK] := V_VISITED[TEMP + 1, JUNK];
(* RESET THE V_CONSIDER ARRAY ROW *)
FOR JUNK := 1 TO N DO
V_CONSIDER[C_EQ_DEG, JUNK] := V_CONSIDER[TEMP + 1, JUNK];
(* RESET THE HAVE CONSIDERED ARRAY ROW *)
FOR JUNK := 1 TO N DO
HAVE_CONSIDERED[C_EQ_DEG, JUNK] := HAVE_CONSIDERED[TEMP + 1, JUNK];
(* RESET THE MAX VISIT ARRAY COMPONENT *)
MAX_VISIT[C_EQ_DEG] := MAX_VISIT[TEMP + 1, JUNK];
(* A HIGHEST DEGREE VERTEX WAS FOUND, IT IS VERTEX EG_HIGHEST_DEG[C_EQ_DEG] *)
END
ELSE
IF (CUM_SUM_DEG[C_EQ_DEG] = CUM_SUM_DEG[TEMP + 1])
THEN

BEGIN
  C_EQ_DEG := C_EQ_DEG + 1;
  (* RESET THE HIGHEST DEGREE VERTEX *)
  IF (C_EQ_DEG >= (TEMP + 1)) THEN
    BEGIN
      EQ_HIGH_DEG[C_EQ_DEG] :=
      EQ_HIGH_DEG[TEMP + 1];
      (* RESET THE VALUE OF CUM SUM DEG ARRAY INDEX *)
      CUM_SUM_DEG[C_EQ_DEG] :=
      CUM_SUM_DEG[TEMP + 1];
      (* RESET VERTEX VISIT ARRAY ROW *)
      FOR JUNK := 1 TO N DO
        V_VISITED[C_EQ_DEG, JUNK] :=
        V_VISITED[TEMP + 1, JUNK];
      (* RESET THE V_CONSIDER ARRAY ROW *)
      FOR JUNK := 1 TO N DO
        V_CONSIDER[C_EQ_DEG, JUNK] :=
        V_CONSIDER[TEMP + 1, JUNK];
      (* RESET THE HAVE CONSIDERED
        ARRAY ROW *)
      FOR JUNK := 1 TO N DO
        HAVE_CONSIDER[C_EQ_DEG, JUNK] :=
        HAVE_CONSIDER[TEMP + 1, JUNK];
      (* SWITCH MAX VISIT ARRAY ITEMS *)
      MAX_VISIT[C_EQ_DEG] :=
      MAX_VISIT[TEMP + 1];
    END;
    (* A TIED HI DEGREE VERTEX WAS FOUND, IT IS VERTEX EQ_HIGH_DEGC_EQ_DEG匠 *)
  END;
  (* C_EQ_DEG VERTICES OF THE HIGHEST DEGREE EXIST *)
  (* THESE VERTICES ARE STORED IN EQ_HIGH_DEG ARRAY *)
  (* SAVE THE POSSIBLY NEW NUMBER OF TIED HI DEGREE VERTICES *)
  INDEX_EQ_DEG := C_EQ_DEG;
  COUNT.UNIQUE V := 1;
  (* IF THERE IS MORE THAN ONE HIGH DEGREE VERTEX THEN CALL THE DEFINE VERTICES TO FOCUS ON PROCEDURE *)
  IF (INDEX_EQ_DEG > 1) AND (INDEX_EQ_DEG < MAX_NI_DEG V)
  AND (DEGREE_SUM_CHANGE = TRUE) THEN
    (* CALL THE DEFINE VERTICES TO FOCUS ON PROCEDURE *)
    DEFINE_VERTICES_TO_FOCUS ON (COUNT.UNIQUE V,
    V_CONSIDER, MAX_DEX_V_CON)
  END (* IF DEGREE SUM CHANGE = TRUE *)
  V_H := EQ_HIGH_DEG[1]
END; (* PROCEDURE LOOK_AHEAD *)

BEGIN (* MAIN PROGRAM *)
  (* READ THE NUMBER OF VERTICES IN THE GRAPH *)
  READLN (N);
  (* INITIALIZE YOUR STRUCTURES *)
  (* SET THE POINTERS TO NIL FOR THE REPRESENTATION OF THE
  EDGES IN THE SPANNING TREE *)
  FOR TEMP := 1 TO N DO
    TOP ADJ TEMP := NIL;
    (* INITIALIZE THE COUNT OF THE BASIC CYCLES *)

-16-
BASIC COUNT := 0;
(* INITIALIZE SUM_ARR_ROW TO *)
FOR TEMP := 1 TO N DO
  SUM_ARR_ROW[TEMP] := 0;
(* INITIALIZE THE ADJACENCY MATRIX *)
FOR V Z := 1 TO N DO
  BEGIN
    FOR V W := 1 TO N DO
      BEGIN
        READ MATRIX_A[ V, Z, V, W ];
      END;
    READLN
  END;
(* INITIALIZE THE TOP OF THE V_TREE *)
TOP VT := 1;
(* SET THE NUMBER OF SUBTREES TO 1 *)
SUB T := 1;
SUB T MAX := 1;
(* INITIALIZE THE GLOBAL SET OF VERTICES *)
FOR TEMP := 1 TO N DO
  GLOBAL V TREE := [ TEMP ] := 0;
(* SET THE GRAPH ROOT INSPECTION FLAG ON *)
ROOT INSPECT := TRUE;
(* SET THE COUNT OF NON ZERO DEGREE VERTICES *)
MAX NZ DEG := N;
(* END OF INITIALIZATION SECTION *)
1
(* DETERMINE THE VERTEX WITH THE HIGHEST DEGREE AND ASCERTAIN HOW MANY VERTICES ARE OF THE HIGHEST DEGREE. PERFORM THIS BY DETERMINING THE ROW OF THE ADJACENCY WITH THE GREATEST SUM *)
INDEX EQ DEG := 1;
SAME HIGH DEG INDEX EQ DEG := 1;
FOR TEMP := 1 TO (N-1) DO
  IF SUM ARR ROW [ SAME HIGH DEG INDEX EQ DEG ] > SUM ARR ROW [ TEMP + 1 ] THEN
    BEGIN
      (* RESET COUNT OF HIGHEST DEGREE VERTICES TO 0 *)
      INDEX EQ DEG := 1;
      SAME HIGH DEG INDEX EQ DEG := TEMP + 1
    END
  ELSE IF (SUM ARR ROW [ SAME HIGH DEG INDEX EQ DEG ] = SUM ARR ROW [ TEMP + 1 ])
    THEN
      BEGIN
        INDEX EQ DEG := INDEX EQ DEG + 1;
        SAME HIGH DEG INDEX EQ DEG := TEMP + 1
      END
(* THERE ARE INDEX EQ DEG NUMBER OF VERTICES OF THE SAME HIGHEST DEGREE AND THESE VERTICES ARE IN ARRAY SAME HIGH DEG *)
(* IF THE HIGHEST DEGREE IS ZERO THEN YOU HAVE EXPLORED ALL THE VERTICES IN THE GRAPH THEREFORE GO TO EXIT AT LABEL 99 *)
IF (SUM ARR_ROW [ SAME HIGH DEG INDEX EQ DEG ] = 0) THEN
  GOTO 99;
(* IF THERE IS MORE THAN ONE HIGH DEGREE VERTEX AND THE NUMBER
OF HIGH DEGREE VERTICES IS LESS THAN THE NUMBER OF NON ZERO
DEGREE VERTICES THEN INITIALIZATION THE CONDITIONS NECESSARY FOR
FOR THE LOOK AHEAD PROCEDURE:* 
IF (INDEX_EQ_DEG > 1) AND (INDEX_EQ_DEG < MAX_NI_DEG_V)
THEN
LOOK_AHEAD(V_Z, SAME_HIGH_DEG);
ELSE
V_Z := SAME_HIGH_DEG [INDEX_EQ_DEG ];
(* CONSIDER ALL VERTICES ADJACENT TO V_Z *)
FOR V_W := 1 TO N DO
IF ( MATRIX_AT [V_Z, V_W ] = 1 )
THEN
BEGIN
EDGE_COMPLETE := FALSE;
FOUND_CYCLE := FALSE;
IF (V_TREE[SUB_T, V_Z ] = 1) OR
((GLOBAL_V_TREE[V_W ] = 1) AND
(V_TREE[SUB_T, V_W ] = 0)) THEN
BEGIN
(* DETERMINE WHICH SUBTREE YOU SHOULD
FOCUS ON...A NEW SUBTREE OF A PREVIOUS
SUBTREE BECAUSE YOU ARE HERE THE
VERTEX V_Z AND V_W IS NOT IN THE PRESENT
SUBTREE SUB_T.*)
(* DETERMINE IN WHICH SUBTREE V_Z AND V_W
ARE IN *)
TEMP := SUB_T[MAX];
FOUND_V_Z := FALSE;
FOUND_V_W := FALSE;
WHILE (TEMP > 0) DO
BEGIN
IF (V_TREE[TEMP, V_Z ] = 1)
THEN
BEGIN
FOUND_V_Z := TRUE;
SUB_T_V_Z := TEMP;
END;
IF (V_TREE[TEMP, V_W ] = 1)
THEN
BEGIN
FOUND_V_W := TRUE;
SUB_T_V_W := TEMP;
END;
TEMP := TEMP - 1
END; (* WHILE (TEMP > 0) *)
IF (FOUND_V_Z) OR (FOUND_V_W)
THEN
BEGIN
(* SET SUB_T TO THE APPROPRIATE VALUE DEPENDING
ON VALUES OF FOUND V_Z AND FOUND V_W *)
IF (FOUND_V_Z) AND (FOUND_V_W = FALSE)
THEN
SUB_T := SUB_T_V_Z;
IF (FOUND_V_W) AND (FOUND_V_Z = FALSE)
THEN
BEGIN
SUB_T := SUB_T_V_W;
EDGE_COMPLETE := TRUE
END:

IF (FOUND_V_Z) AND (FOUND_V_W)
THEN
  IF (SUB_T_V_W = SUB_T_V_Z)
  THEN
    SUB_T := SUB_T_V_Z
  ELSE
    BEGIN
      EDGE_COMPLETE := TRUE;
      (* MERGE TWO SUBTREES *)
      V_TREE := SUB_T_V_W
      FOR TEMP := 1 TO N DO
        IF (V_TREE[SUB_T_V_W, TEMP] = 1)
        THEN
          BEGIN
            V_TREE[SUB_T_V_Z, TEMP] := 1;
            V_TREE[SUB_T_V_W, TEMP] := 0
          END
          (* SET NEW SUBTREE TO SUB_T_V_Z *)
        SUB_T := SUB_T_V_Z
      END;
      /* ELSE */
    END
  END;

END) (* IF (FOUND_V_Z) OR (FOUND_V_W) *)

IF (FOUND_V_Z = FALSE) AND (FOUND_V_W = FALSE)
THEN
  (* START A BRAND NEW SUBTREE *)
  IF (ROOT_INSPECT = FALSE)
  THEN
    BEGIN
      (* ADD 1 TO SUBTREE COUNT *)
      SUB_T_MAX := SUB_T_MAX + 1;
      /* INITIALIZE THE NEW SUBTREE VERTICES */
      FOR VT := 1 TO N DO
        V_TREE[SUB_T_MAX, VT] := 0;
      SUB_T := SUB_T_MAX
      ELSE
        ROOT_INSPECT := FALSE)
  END

IF (FOUND_V_Z = FALSE)
THEN
  BEGIN
    (* PUT THE ROOT OF THE SUBTREE IN THE V_TREE *)
    V_TREE[SUB_T, V_Z] := 1;
    V_TREE[SUB_T, V_W] := 1;
    FOUND_V_Z := TRUE
  END;

END;

IF (EDGE_COMPLETE = FALSE) AND (V_TREE [SUB_T, V_W] = 1)
AND (V_TREE [SUB_T, V_Z] = 1)
THEN
  /* THE EDGE BETWEEN Z AND W IS PART OF A */
  /* BASIC CYCLE */
  BEGIN
    (* INCREMENT THE COUNT OF THE BASIC CYCLES *)

-19-
BASIC_COUNT := BASIC_COUNT + 1;

(* INITIALIZE THE TREE_EDGE_ARRAY TO ZERO *)
FOR TEMP := 1 TO N DO
  TREE_EDGE_ARRAY[ TEMP ] := 0;

(* CALL BUILD_TREE ARRAY PROCEDURE *)
BUILD_TREE_ARRAY ( V_Z, V_W, TOP_ADJ,
  TREE_EDGE_ARRAY);

(* WRITE THE COMPONENTS OF THE CYCLE *)
WRITE ("CYCLE :", BASIC_COUNT : 2,
  "CONSISTS OF THE PATH :"
);
WRITE (V_Z : 2, "->", V_W : 3);

TEMP := V_W;
CYCLE_LENGTH := 1;
WHILE (TEMP <> V_Z) DO
BEGIN
  CYCLE_LENGTH := CYCLE_LENGTH + 1;
  WRITE ("->",
    TREE_EDGE_ARRAY[ TEMP ] : 3);
  TEMP := TREE_EDGE_ARRAY[ TEMP ];
END;
WRITELN;
SAVE_CYCLE_LENGTH[ BASIC_COUNT ] := CYCLE_LENGTH;

FOUND_CYCLE := TRUE;
EDGE_COMPLETE := TRUE
END;

IF (V TREE_SUB_T, V_Z I = 0) AND
  (V TREE_SUB_T, V Z I = 1) AND
  (EDGE_COMPLETE = FALSE)
THEN
BEGIN
  (* ADD V W TO LOCAL AND GLOBAL TREES *)
  V TREE_SUB_T[I, V_W ] := 1;
  GLOBAL V TREE[I, V_W ] := 1;
  EDGE_COMPLETE := TRUE
END;

IF (EDGE_COMPLETE)
THEN
BEGIN
  (* DELETE EDGE i-W FROM THE GRAPH WHICH * )
  (* IS REPRESENTED BY AN ADJACENCY MATRIX *)
  MATRIX_A [ V_Z, V_W ] := 0;
  MATRIX_A [ V_W, V_Z ] := 0;
  (* ADJUST ROW SUMS *)
  SUM.ARR_ROW[V_Z ] := SUM.ARR_ROW[V_Z ] + 1;
  SUM.ARR_ROW[V_W ] := SUM.ARR_ROW[V_W ] + 1;
  IF (FOUND_CYCLE = FALSE)
  THEN
    (* ADD EDGE i-W TO SPANNING TREE *)
    (* CALL ADD_EDGE_TO_TREE PROCEDURE *)
    ADD_EDGE_TO_TREE ( V_Z, V_W, TOP_ADJ)
END;

MAX_NZ_DEG_V := MAX_NZ_DEG_V - 1;
GOTO 10;

FOR TEMP := 1 TO BASIC_COUNT DO
    TOTAL_CYCLE_LENGTH := TOTAL_CYCLE_LENGTH + SAVE_CYCLE_LENGTH[TEMP];
    AVE_CYCLE_LENGTH := TOTAL_CYCLE_LENGTH / BASIC_COUNT;
END.

WRITELN ('THERE WERE ', BASIC_COUNT, ' CYCLES FOUND OF ', AVERAGE_LENGTH, ' AVE_CYCLE_LENGTH : 3 : PRECISION')