Symbiosis

www.symbiosisonline.org
www.symbiosisonlinepublishing.com

Research article | Journal of Computer Science Applications and Information Technology | Open Access

Matbase DFS Detecting and Classifying E-RD Cycles Algorithm

Christian Mancas* and Adrian Mocanu

Department of Mathematics and Computer Science, Ovidius University, Constanta, Romania

Received: November 06, 2017; Accepted: November 25, 2017; Published: December 07, 2017

***Corresponding author:** Christian Mancas, Department of Mathematics and Computer Science, Ovidius University, Constanta, Romania, Tel: +40722357078; E-mail: christian.mancas@gmail.com*

Abstract

A Depth First Search type algorithm for detecting and classifying all cycles of a directed graph was designed and implemented in *MatBase* for database Entity-Relationship Diagrams. Its time complexity, optimality, and utility for teaching both graph theory, sets, functions, and relations algebra, as well as, especially, for database non-relational constraints discovery and enforcement are discussed and exemplified with real world examples.

CCS Concepts

• Information systems~Entity relationship models

• Theory of computation~Dynamic graph algorithms

• ***Theory of computation~Data modeling***

• ***Theory of computation~Database constraints theory***

• Applied computing~Computer-assisted instruction

Keywords: (Elementary) Mathematical Data Model; MatBase; Depth First Search;

Introduction

The Relational Data Model (RDM) provides only five types of practical constraints: domain (range, in fact, co-*domain;* e.g. *Altitude* between 1000 and 8848), *totality* (not null; e.g. *CountryName* not null), key (uniqueness, i.e. minimal injectivity; e.g. *StateName* • *Country* unique), *referential integrity* (foreign key, typed inclusion; e.g. *CapitalCity* references *CITIES*), and tuple (check; e.g. *minAltitude* ≤ *maxAltitude*) [1-3]. Consequently, most Relational Database Management Systems (RDBMS) only provide these constraint types (some of them even not providing the tuple ones!) [4]. Unfortunately, they are not at all enough, either for accurate data modeling or for preventing databases (dbs) from storing implausible data.

For example, the obvious constraint "for any country, its capital should be a city of that country" (or, dually, "no country may have as its capital a city of another country") cannot be expressed in RDM and is not enforceable by any RDBMS. Failing to enforce it could result in storing highly implausible data, such as Helsinki is the capital of Romania or/and Bucharest is the capital of Finland. Therefore, besides RDM and the Entity-Relationship Data Model (E-RDM), for the undergraduate Databases and M.Sc. Advanced Databases disciplines we also teach to our students the (Elementary) Mathematical Data Model ((E)MDM), which provides a plethora of 60 types of constraints, out of which more than 55 are non-relational [5,6,3,7-9].

MatBase is a prototype data and knowledge based management system based on RDM [3,4,8,9], E-RDM, (E)MDM, as well as on Datalog [2,9], intensively used in our undergraduate Databases and M.Sc. Advanced Databases labs and projects.

Very many non-relational constraints are associated to E-R Diagrams (E-RD) cycles. For example, the above one is associated with the cycle shown in figure 1 and it is formalizable equivalently by either $Country^0\ CapitalCity = 1_{COUNTRIES}$ (where $1_{COUNTRIES}$ is the unity mapping of the set *COUNTRIES*) or $Country^0\ CapitalCity$ reflexive (i.e. $Country(CapitalCity(x)) = x, \forall x \in COUNTRIES$).

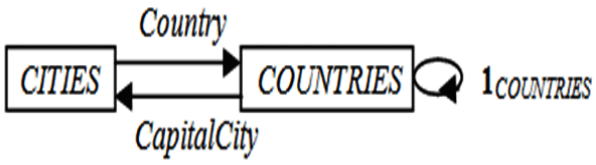

Figure 1: An example of an E-RD cycle having an associated non-relational constraint.

E-RDs are directed graphs having object sets (db tables and views) as nodes and mappings (structural functions) between them (db foreign keys) as edges. Autofunctions, like $1_{COUNTRIES}$, *Mother* : *PEOPLE* → *PEOPLE, Folder* : *FILES* → *FILES,* etc. are the simplest E-RD cycles, as they have length 1. They too may have associated non-relational constraints (e.g. both *Mother* and *Folder* are acyclic, as nobody may be his/her maternal or paternal ancestor, neither directly, as mother or father, nor indirectly), but they are very easy to detect, even from legacy undocumented dbs (as they are foreign keys referencing their own tables). E-RD cycles of length greater than one can only be of the following three types [9]:

Commutative (only one source and one destination nodes, i.e. one node from which arrows only depart and one into which arrows only arrive)

- *Circular* (all nodes are intermediate ones, i.e. both

 ***Corresponding author email:** christian.mancas@gmail.com*

source and destination)

- *General* (all others)

For commutative type ones (e.g. Figure 2), db designers should ask themselves first whether they should commute or not. Whenever they should, if one involved mapping may be defined with the help of the others (e.g. Figure 2, where *CustomerCountry* = *Country* ^{0} *CustomerCity*, as any customer should be located in the country to which the city where it is located belongs), then it should be either eliminated from the data model (thus avoiding the need to enforce the corresponding function diagram commutativity constraint) or kept as a controlled redundancy (e.g. *CustomeureCountry* should either be eliminated or kept as an automatically computed column of table *CUSTOMERS*, read-only for users, and stored only for increasing querying speed).

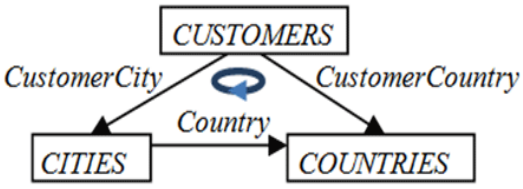

Figure 2: An example of a (conventional) commutative E-RD cycle.

Those commutative type ones that should not commute might instead anti-commute (e.g. *Departure*(x) → *Destination*(*x*), ∀ *x* ∈ *PLANE_TICKETS*, Departure : *PLANE_TICKETS* → *AIRPORTS*, Destination : *PLANE_TICKETS* → *AIRPORTS*).

The E-RD cycle from figure 1 above is a circular one. For any such cycle, in all their nodes, db designers should investigate the properties of the corresponding composed autofunctions. As autofunctions are particular cases of dyadic relations, they may be reflexive, irreflexive, symmetric, asymmetric, idempotent, anti-idempotent, acyclic, etc. Fortunately, for example, *CapitalCity* ^{0} *Country* from A. 1 has none of these properties, so no additional non-relational constraint is associated to this E-RD cycle.

Finally, for example, the E-RD cycle of figure 3, made out of mappings *#SH* : *SEA_HARBORS* →*CITIES* (the canonical injection associated to the inclusion *SEA_HARBORS* ⊂ *CITIES*), Sea : *SEA_HARBORS* → *SEAS*, *Country* : *CITIES* → *COUNTRIES*, *NSea* : *NEIGHBOR_SEAS* → *SEAS*, and *NCountry* : NEIGHBOR_SEAS →COUNTRIES is of type general, having length 5, *SEA_HARBORS* and *NEIGHBOR_SEAS* as source nodes, *CITIES* as intermediate, and *SEAS* and *COUNTRIES* as destination ones (please note that, generally, such cycles have length at least 4 and have at least two source and/or destination nodes).

This cycle has an associated non-relational constraint too, namely "any sea harbor should be a city of a country that is neighbor to that sea" (formally, (∀ *x* ∈ *SEA_HARBORS*) (∃y∈*NEIGHBOR_SEAS*)(*Country(#SH(x))*= *NCountry(y)* ∧ *Sea(x)* = *NSea(y)*)). Failing to enforce it, the db might store such implausible data as the French Marseille city is a harbor of the Baltic sea (although France is not neighbor to this sea).

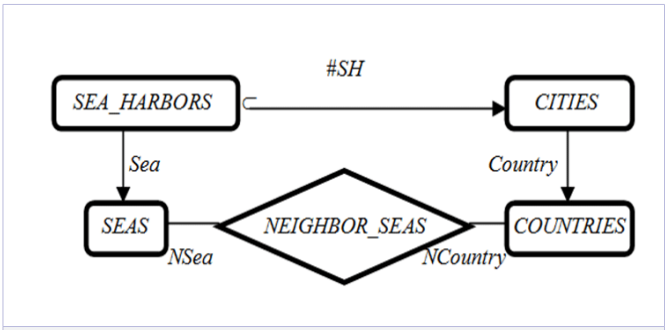

Figure 3: An example of a general E-RD cycle having an associated non-relational constraint

Such constraints are called in the (E) MDM *generalized commutativities.* Commutative and even circular type cycles may also have associated generalized commutativities [9].

To conclude with, for accurate data modeling, db designers should always also detect all corresponding E-RD cycles and, depending on their types, analyze them algorithmically for adding to the corresponding db schemes all constraints existing in that sub-universe of discourse [9].

Detecting and classifying all E-RD cycles is not at all an easy task. For example, a geographic well-designed db having only 63 object sets (tables and views) and 352 structural functions (foreign keys) between them has 10 autofunctions and 142,855 cycles of length at least 2 (out of which 31 are circular, 350 commutative, and the rest are general ones). Therefore, we incorporated in *MatBase* an algorithm for detecting and classifying E-RD cycles that is presented in this paper. This algorithm uses in a first step the Depth First Search (DFS) approach, which detects whether an undirected graph has a cycle or not, then detects all undirected cycles, and then all corresponding oriented ones, which are finally classified into the three possible types [10].

This work is original and without any counterpart: neither in math, nor in computer science this topic has not been extensively tackled like in our approach. Math only deals with function diagram commutativity (almost only for length 3 diagram), while computer science ignores it altogether, except for an early partial attempt of ours [11].

The second section of this paper gradually introduces MatBase DFS-based algorithms for detecting and classifying all cycles of a db scheme E-RD (in fact, as a MatBase db scheme may also contain tables with foreign keys referencing tables from other dbs, such an E-RD may include not only tables and views from a desired db scheme, but also tables and views from other related dbs): subsection "Detecting all cycles of length greater than 2 in an undirected graph" presents the DFS algorithm that detects all cycles of length greater than 2 in an undirected graph; subsection "Managing directed cycles of length 2" the one for detecting and classifying all length 2 cycles in a directed graph; subsection "Decomposing undirected cycles into the corresponding E-RD directed ones" the one for decomposing undirected cycles of length greater than 2 in corresponding directed ones; subsection "MatBase implementation" discusses

MatBase object-oriented implementation of all these algorithms and presents the algorithm for analyzing and classifying directed cycles of length greater than 2; finally, subsection "MatBase DFS-based algorithm for detecting and classifying all E-RD cycles" presents the top MatBase DFS detecting and classifying E-RD cycles algorithm, which calls all of the above introduced ones. Section "Results and Discussion" discusses both the complexity and utility of this algorithm, not only for db design, but also for the graph theory and sets, functions, and relations algebra. The paper ends with conclusions and further work, acknowledgements, and references.

The *Matbase* DFS Detecting and Classifying "E-RD (Remove Quotes)" Cycles Algorithm

Detecting All Cycles of Length Greater Than 2 in an Undirected Graph

The DFS algorithm for deciding whether an undirected graph is acyclic or not [10] completely ignores vertices that have been fully explored, as well as the ones that have been marked as FINISHED, and it only considers the direct parent of a vertex (node), not taking into consideration its ancestors too. For an algorithm based on DFS to be able to detect all cycles of length equal to or greater than 3, the following facts must be considered:

✓ The edges that lead to fully explored vertices which have been marked as FINISHED must also be taken into consideration. If we ignore such edges, the algorithm will not consider all the possible paths that make up E-RD directed graph cycles.

✓ During exploration of all possible paths, a vertex may have more than one parent. More precisely, until a vertex *v* is fully explored, it only has one parent, which is the node *u* that leads to node *v*, when node *v* was still undiscovered. After *v* has been fully explored and marked as FINISHED, if node *v* is visited again through some other vertex *w*, then w becomes another parent of *v*.

✓ When an edge from the current vertex *u* leads to an adjacent vertex *v* that is marked as DISCOVERED, before considering a cycle the algorithm must check v against the entire list of parent nodes of *u*, not just checking whether *v* is the latest parent of *u*.

Thus, our algorithm considers edges that lead to fully explored nodes and stores all the parent nodes of a vertex in a linked list type data structure, out of which the current parent (latest discovered one) receives special consideration.

One fundamental question should then be answered: "I*f the algorithm does not stop once all the vertices have been fully explored, when will it stop?*". To begin with, edges that lead to fully explored nodes are not treated the same way as the edges that lead to undiscovered nodes, since this would obviously cause the algorithm to loop indefinitely. When an edge from the current vertex *u* leads to an adjacent vertex *v* that has been fully explored, the following facts must also be considered:

✓ If the current vertex u has also been fully explored, thus marked as FINISHED, and the current (latest discovered) parent node of *u* is the adjacent node *v*, then the edge (*u, v*) is ignored, since considering it would mean to go one step back on the current path.

✓ If the above condition does not hold, then, regardless the state of vertex *u* (DISCOVERED or FINISHED), if the transitive closure between nodes *u* and *v* contains a cycle (or, simpler put it, if *v* is an ancestor node of *u*), then the edge (*u, v*) should be ignored. Not ignoring it would mean considering the same path over and over again, thus looping endlessly. Fortunately, in determining whether one vertex is the ancestor of another one, only the latest parents of each involved vertices need to be considered (and not any other parent nodes that have been assigned during the exploration).

Considering the above, one can infer that, at a given point, after all the vertices have been fully explored, except for the source node, all possible distinct paths have been explored and thus no more edges are to be considered, allowing the algorithm to end its computations.

In the first step of our algorithm, for any given E-RD, an underlying undirected graph structure is constructed, in which the vertices are the same as in the E-RD, but regardless of the number and orientation of the edges between any two adjacent nodes, a single undirected edge exists instead. For example, the two arrows from figure 1 are consolidated into a single line (undirected edge between nodes *CITIES* and *COUNTRIES*).

Given a directed E-RD graph *G*, let *V(G)* be the set of vertices belonging to *G* and *E(G)* the one of directed edges linking the nodes of *V(G)*. Unlike the classical directed graph structure, there may be multiple distinct edges having the same orientation between any two vertices. Let *A(G)* be the adjacency matrix associated to *G* and $A(G)_{i,j}$ be the number of edges directed from vertex i to vertex j, A(G)i,j ≥ 0.

Let G' be an undirected graph such that:

✓ $V(G') = V(G)$

✓ $E(G') = \{(i,j) \mid i,j \in V(G) \wedge (\exists\,(i,j) \in E(G) \vee \exists\,(j,i) \in E(G))\}$

✓ $A(G')_{i,j} = \{1 \text{ if } (A(G)_{i,j} \geq 1 \text{ or } A(G)_{j,i} \geq 1) \text{ or } 0 \text{ otherwise}\}$

It is quite trivial that an undirected cycle may contain between 1 and n directed E-RD cycles, based on the number of edges existing between the vertices involved in the associated E-RD graph structure. More precisely, the exact number of E-RD cycles of a given undirected cycle can be computed by using the following intuitive formula (where mod is the algebraic *modulo* operator):

$$\prod_{i=1}^{n} A\left[i,(i+1) \bmod n\right] + A\left[(i+1) \bmod n, i\right]$$

The following notations have been used:

✓ *n* represents the number of vertices involved in the cycle

✓ *i* represents the current node, counting starting from 1 and going up to *n*

✓ *A* is the adjacency matrix corresponding to the E-RD graph

✓ A[*i,* (i+1) mod *n*] represents the number of edges directed from the current node i to the next adjacent node. We have used the modulo operator to denote that when the current vertex is the last one (i.e. *n*), the first node is considered as its next one.

✓ A[(*i*+1) mod *n, i*] represents the number of inverted edges, directed from the next vertex to the current vertex *i*.

Hence, the exact number of E-RD cycles associated to an undirected cycle is the result of the multiplication between the numbers of directed edges existing between any two adjacent nodes. However, this formula is only valid for cycles of length equal to or greater than 3.

To sum up the overall strategy employed by our DFS-based algorithm for detecting all cycles in an undirected graph, relevant pseudocode is presented in the remainder of this subsection. The set of parent nodes for the current vertex *u* is stored by *parents*[*u*], a linked list stored in an array type data structure, whereas the current (latest discovered) parent of *u* is stored by *p*[*u*].

When exploring an arbitrary edge (*u, v*) which leads from the current node *u* to an adjacent node *v* that has already been discovered, the following method is used to determine whether *v* is a parent node of *u*:

ALGORITHM 1: ***Is Child Of (u, v)***

if v ∈ parents[*u*] then return true else return false;

If this method returns false, a cycle has been detected; otherwise, the edge (*u, v*) is ignored, since it would take us one step back on the path that leads to the current vertex *u* in the first place.

When an arbitrary edge (*u, v*) leads from the current node *u* to an adjacent node *v* which has been fully explored and node *u* is not fully explored too and is a child of node *v*, then the following method is used to check whether *v* is an ancestor of *u*, as given by the family tree containing the latest parents of the nodes involved:

ALGORITHM 2: ***Is Ancestor (u, v)***

```
w := p[u];
while w != NIL do
  if w = v then return true;
  w := p[w];
end while;
return false;
```

The pseudocode for the overall strategy of the presently discussed DFS-based algorithm is the following (where methods AnalyzeLength2ERDLoops and Analyze Loop are presented in the subsections Managing directed cycles of length 2 and MatBase implementation, respectively):

ALGORITHM 3: ***DFS(V, E)***

```
for each vertex u in V do
  state[u] := UNDISCOVERED;
  p[u] := NIL;
  parents[u] := ∅;
end for;
for each vertex u in V do
  if state[u] = UNDISCOVERED then Visit(u);
end for;
```

ALGORITHM 4: ***Visit(u)***

```
if state[u] = UNDISCOVERED then state[u] := DISCOVERED;
for each vertex v adjacent to u do
  if state[v] = UNDISCOVERED then
```

```
    parents[v] := parents[v] ∪ {u};
    if there are several foreign keys between u and v then
      AnalyzeLength2ERDLoops(u, v);
    Visit(v);
  else if state[v] = DISCOVERED then
      if not IsChildOf(u, v) then AnalyzeLoop(u, v);
      else if state[v] = FINISHED then
          if not state[u] = FINISHED and p[u] = v then
            if not IsAncestor(u, v) then
              parents[v] := parents[v] ∪ {u} ;
              Visit(v);
            end if;
          end if;
        end if;
    end if;
  end for;
  if state[u] = DISCOVERED then state[u] := FINISHED;
```

If we denote by $d[u]$ the moment that vertex u has been discovered and by $f[u]$ the moment that vertex u has been fully explored, the following can be stated: before u has been fully explored, every vertex v for which the inequality $d[u] < d[v] < f[v] < f[u]$ holds (i.e. every descendant of u) is explored and u is visited a number of times equal to the number of edges between u and any other such vertex v.

For example, in the graph presented in figure 4, where nodes *B*, *C*, and *D* are descendants of *A*, we have also figured for each vertex its ratio $d[u] / f[u]$.

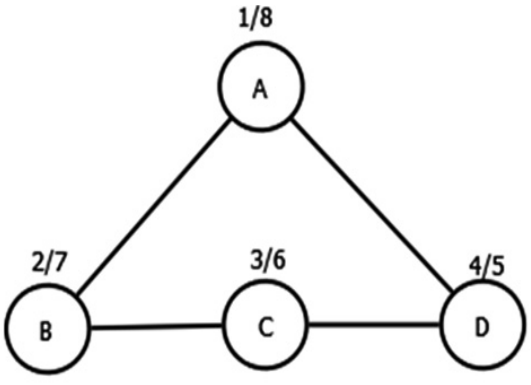

Figure 4: Example of a DFS graph exploration.

When the *DFS* algorithm presented in this section is applied to it, then, before vertex *A* is fully explored, i.e. $f[A] = 8$, each of the vertices *B, C,* and *D* have been visited 2 times, as follows:

- ✓ vertex *B* first time when it is discovered by exploring edge (*A, B*) and the second time when the following edges are explored (in this order): (*A, D*), (*D, C*), (*C, B*)
- ✓ vertex *C* first time when it is discovered by exploring edge (*B, C*) and the second time when the following edges are explored (in this order): (*A, D*), (*D, C*)
- ✓ vertex *D* first time when it is discovered by exploring edge (*C, D*) and the second time when edge (*A, D*) is explored.

Since there are at most two edges between vertex *A* and any of its descendants, *B, C,* and *D* are visited 2 times each one.

Managing Directed Cycles of Length 2

In the underlying undirected graph, length 2 cycles (e.g. the one in figure 5) does not exist; therefore, any undirected edge between two nodes could potentially represent multiple lengths 2 E-RD cycles. In such cases, the number of length 2 cycles is given by *n choose 2*, where *n* is the number of directed edges between the two vertices involved.

Therefore, above method Visit(u) has to call a method AnalyzeLength2ERDLoops(u, v) that discovers and classifies all directed cycles of length 2 which exist between current node u and each of its children v, as follows: for every unordered pair of adjacent nodes, if the number of directed edges (db foreign keys) existing between them is greater than 1, method AnalyzeLength2ERDLoops is called. To avoid checking the same pair of nodes every time, the undirected edge linking them is explored; for every such pair, a unique string value is generated by concatenating the smaller node id with a '.' character followed by the greater node id. Thus, based on the ids of the two adjacent nodes, a unique value is generated and stored in a list; then, every time that such another pair is encountered the value obtained for it is checked against this list for distinguishing between already considered edges and newly discovered ones.

Method AnalyzeLength2ERDLoops takes as input the two adjacent nodes involved and computes the set of directed edges existing between the two. For every distinct unordered pair of directed edges, the following computations are performed:

- ✓ Detects the cycle type as follows: if both edges have same direction, then the cycle is commutative else it is circular.
- ✓ Inserts the directed E-RD cycle data in *MatBase's ERDLoops* table.
- ✓ Inserts the corresponding directed edges data in *MatBase*'s *ERDLoopsFunctions* table.
- ✓ For circular type cycles, computes the two-associated compound autofunctions (one for each vertex) and inserts their data in MatBase's FUNCTIONS table (where data on all db columns, be them computed or fundamental, which includes foreign keys too, are stored).

Here is the pseudocode of this method (where |E| denotes E's cardinal):

ALGORITHM 5: ***AnalyzeLength2ERDLoops(u, v)***

```
E := {the set of directed edges between nodes u and v}
n := |E|
for i := 1, n – 1, i + 1 do
   for j := i + 1, n, j + 1  do
      if E[i] and E[j] start from u then type := commutative
      else type := circular;
      insert cycle data in table ERDLoops;
      insert E[i] and E[j] data in table ERDLoopsFunctions;
      if type = circular then
         insert E[i] ∘ E[j] and E[j] ∘ E[i] data in table FUNCTIONS;
   end for;
end for;
```

Decomposing Undirected Cycles into the Corresponding "E-RD (Remove Quotes)" Directed Ones

For undirected cycles of length equal to or greater than 3 the process of decomposing them into the corresponding E-RD directed ones is much more complex, but the underlying idea is simple: every possible path is explored, by considering every possible combination of existing directed edges, starting from and ending at the first node of each undirected cycle.

Therefore, whenever an undirected cycle is detected, a method for analyzing and classifying all corresponding directed E-RD cycles of length equal to or greater than 3 must be called. This method takes as input the list of vertices involved in a current undirected cycle, in which the first node of the cycle is stored on both the first and last positions, thus allowing for a circular traversal; the following data is stored on the system stack: the position of the current node within the cycle, the set of foreign keys which are the edges of the E-RD cycle that is currently being analyzed, and the numbers of edges leaving from and arriving to the current node, respectively (needed for cycle classification).

The overall simplified process is as follows:

If the current position is not the last one in the list of nodes, compute the set of foreign keys between the current vertex and the next one in the list. If there is a single edge between the two nodes, then add it to the set of foreign keys on the position of the current node and, based on the direction of the edge, either increase the number of functions defined on or the number of functions taking values from the current node and increase the opposite ones for the next node; immediately after, a recursive call to the main method with the current data for the next vertex in the list is performed.

If there are multiple directed edges between the current two adjacent nodes, then for each such edge perform the following computations:

- ✓ If the set of foreign keys contains no element on the position associated to the current vertex, the current edge is added and, based on its orientation, the number of functions defined on or the number of functions taking values from the current node is updated;
- ✓ else, compare the current edge with the foreign key that is stored in the set of foreign keys on the position of the current node; if they are distinct, the current foreign key is stored separately and the element on the position of the current node is updated, thus becoming the current edge. The orientations of the previous and the new current edges are compared and, if they are different, the number of functions defined on and the number of functions taking values from the current node are correspondingly updated.
- ✓ In either case, the method is recursively called for the next vertex in the list, if any.

• else, if the current position is the last one in the list, it means that the starting node has been reached again, thus completing the current possible circular traversal of the cycle. Based on the number of functions defined on and taking values from each of the nodes, the type of cycle and the associated compound function(s) are computed, and all this data is stored in the corresponding MatBase meta-catalog tables. Please note that a commutative type cycle of length at least 3 has either one or two such associated compound functions (corresponding to the two paths between its source and destination nodes), any circular type cycle has one compound function associated to each of its nodes, whereas general type ones may have between none (e.g. $f_{11} : S_1 \to D_1$, $f_{12} : S_1 \to D_2$, $f_1 : S_2 \to D_1$, $f_{22} : S_2 \to D_2$) and $n/2$ associated compound functions, where n is the cycle length (e.g. $f_{111} : S_1 \to N1$, $f_{11} : N_1 \to D1$, $f_{121} : S_1 \to N_4$, $f_{12} : N_4 \to D_2$, $f_{211} : S_2 \to N_2$, $f_{21} : N_2 \to D_1$, $f_{221} : S_2 \to N_3$, $f_{22} : N_3 \to D_2$, which has length 8 and the following 4 compound functions: $f_{11} \circ f_{111}$, $f_{12} \circ f_{121}$, $f_{21} \circ f_{211}$ and $f_{22} \circ f_{221}$).

Here is the pseudocode of this method that, for any undirected cycle, explores all possible paths in the directed E-RD graph and detects all corresponding directed cycles, classifies them, and stores their data in the corresponding *MatBase* meta-catalog tables (where: i is the current node position in the cycle; vector nodes stores the cycle's nodes; vector *functions* stores the foreign keys involved in the cycle; vectors *fin* and *fout* store the number of directed edges that start from and arrive into each cycle node, respectively; *sources* and *destinations* store the number of source and destination nodes of the current cycle, respectively; vector *edges* stores all foreign keys between the current and the next cycle nodes; and n stores the cycle length):

ALGORITHM 6: ***AnalyzeERDLoops (i, nodes, functions, fin, fout)***

```
n := |nodes|;
if i = n then    // cycle traversal completed: first node re-reached
  sources := 0;
  destinations := 0;
  for j := 1, n, j + 1 do
    if fin[j] = 0 then sources := sources + 1;
    elseif fout[j] = 0 then destinations := destinations + 1;
  end for;
  if sources = 0 and destinations = 0 then type := circular;
  elseif sources = 1 and destinations = 1 then type := commutative;
  else type := general;
  insert cycle data in table ERDLoops;
  for j := 1, |functions|, j + 1 do
    insert functions[j] related data in table ERDLoopsFunctions;
    based on type, compute the associated compound functions
        and insert their data in table FUNCTIONS;
  end for;
else edges := {the set of edges between nodes[i] and nodes[i+1]};
  if |edges| = 1 then
    if functions[i] = NIL then
      functions[i] := edges[1];
      if functions[i] defined on nodes[i] then
        fout[i] := fout[i] + 1;
        fin[(i mod n) +1] := fin[(i mod n) +1]+ 1;
      else
        fin[i] := fin[i] + 1;
        fout[(i mod n) +1] := fout[(i mod n) +1] + 1;
        AnalyzeERDLoops(i+1, nodes, functions, fin, fout);
      end if;
    else
      for j := 1, |edges|, j + 1 do
        if functions[i] = NIL then
```

```
                functions[i] := edges[j];
                if functions[i] defined on nodes[i] then
                  fout[i] := fout[i] + 1;
                    fin[(i mod n) +1] := fin[(i mod n) +1] + 1;
                else
                    fin[i] := fin[i] + 1;
                      fout[(i mod n) +1] := fout[(i mod n) +1] + 1;
                end if;
            else
              if functions[i] != edges[j] then
                if functions[i] and edges[j] are not defined on the
                                                same set then
                  if functions[i] is defined on nodes[i] then
                    fout[i] := fout[i] – 1;
                    fin[i] := fin[i] + 1;
                    fout[(i mod n) +1] := fout[(i mod n) +1] + 1;
                     fin[(i mod n) +1] :=  fin[(i mod n) +1] - 1;
                  else
                    fout[i] := fout[i] + 1;
                    fin[i] := fin[i] – 1;
                    fout[(i mod n) +1] := fout[(i mod n) +1] – 1;
                    fin[(i mod n) +1] := fin[(i mod n) +1] + 1;
                  end if;
                end if;
                Functions[i] := edges[j];
                AnalyzeERDLoops(i+1, nodes, functions, fin, fout);
              end if;
            end if;
          end for;
        end if;
      end if;
    end if;
```

MatBase Implementation

Currently, *MatBase* has two versions: one developed in MS Access 2016 and one in C# and MS SQL Server 2016. The above presented DFS-based algorithms have been successfully implemented and tested in both versions.

For designing and implementing these algorithms, an object-oriented approach has been considered. Therefore, for each element of the problem domain a class has been defined and implemented, allowing for accurate modeling of the vertex, edge, and graph concepts.

The UML diagram shown in figure 5 illustrates these classes and the relationships between them.

A vertex represents an entity, relationship, or computed set of an E-RD (i.e. a table or view of the corresponding db scheme). The ERDVertex class contains methods for initializing the adjacency

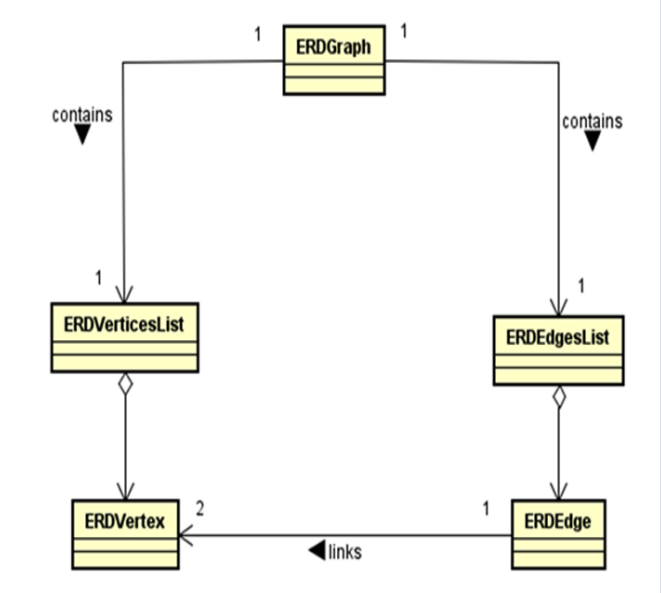

Figure 5: The main UML class diagram for implementing the above presented DFS-based algorithms in *MatBase.*

list, adding a parent node, checking whether a vertex has any parent nodes, and for checking whether a given vertex is a direct descendant of another one.

The *ERDEdge*, class (that has no methods) is modelling the concept of edge (arrow of an E-RD connecting two object sets), which represents a structural function between sets (i.e. a foreign key of the corresponding db scheme).

The *ERDVerticesList* is a singleton class representing an aggregation of *ERDVertex* objects used to store in memory the list of all the vertices corresponding to the sets that are the nodes of the currently analyzed db E-RD. This class contains methods for adding a vertex object to the list, fetching a specific vertex based on its id, and for checking whether an ancestor-descendant relationship exists between two nodes; it also contains a method for reinitializing the list of vertices, which marks all nodes as being UNDISCOVERED and for each vertex voids its list of parents (which is useful when users wish to run the algorithms again on a same db E-RD, since there will be no need to perform the initialization process again, thus saving running time).

The *ERDEdgesList* is a singleton class representing an aggregation of *ERDEdge* objects used to store in memory the list of all edges corresponding to the underlying db scheme foreign keys; it also stores, for every pair of adjacent nodes, the list of directed edges between them, thus preventing additional computations. This class contains methods for adding an edge to any of the two lists, for retrieving an edge based on its id, for checking whether a length 2 loop exists between two vertices (i.e. that there are at least two directed edges between them), and for retrieving the list of edges between two nodes.

ERDGraph is the main class, whose methods implement the DFS-based algorithms for detecting and classifying E-RD cycles. This class does not model the concept of a graph in its literal sense, but rather provides the collection of all the sets (db tables and views) that represent potential nodes and the collection of all the structural functions (db foreign keys) that represent potential edges of a potential graph structure. Actual E-RD graphs are dynamically defined during the DFS exploration. This class contains the method used to initialize the list of vertices, edges, and compound functions, as well as the methods used to implement the DFS-based algorithms that were presented in the previous subsections (Visit, AnalyzeLength2ERDLoops, and AnalyzeERDLoops, as well as those called by them).

Moreover, since compound functions are not edges of E-RD graphs (but paths of them made from adjacent edges having same direction), the following specific classes have been designed and developed for them:

• *ERDCompFunc* is the class representing a compound function designed to store the required details of a compound function in memory, thus preventing additional read operations from disk.

• *ERDCompFuncList* is a singleton class used to store in memory and manage the list of compound functions as a collection of *ERDCompFunc* objects. Existing compound functions of the current db are loaded at the beginning of the algorithm, in the initialization stage, and new compound functions are added as they are detected. This class also contains methods for generating mathematical and SQL expressions of compound functions, as well as for inserting compound function details into the corresponding *MatBase* tables, as they are constructed by the DFS-based algorithms.

Based on the type of computations it performs, the overall execution of the *MatBase* algorithm for detecting and classifying all E-RD cycles may be divided into the following three stages:

• The initialization stage

• The cycle detection stage

• The cycle classification and data saving stage

In the *initialization stage*, the data required for running the algorithm is loaded into memory. After users select a db and request to run the algorithm for it, if it has been previously run for the same db, users are asked to confirm whether they indeed wish to delete the current cycle data for this db and re-compute it.

When the *ERDGraph* class is instantiated, it will instantiate all other singleton classes. The *ERDvertices* and *ERDEdges* temporary tables are truncated and reloaded (from the *MatBase FUNCTIONS* table), after which the relevant data from the MatBase SETS meta-catalog table (which stores data on all db schemes tables and views) is stored in memory, in the list attribute of the *ERDVerticesList* class, whereas the data from the *ERDEdges* table is stored in the list attribute of the *ERDEdgesList* class. Finally, the compound functions details will be loaded in the list attribute of the *ERDCompFuncList* class. Before proceeding to the next stage, the algorithm checks that the selected db scheme contains at least one node, by checking the number of elements in the *CurrentDbVertices* list attributes of the *ERDvertices class*. If it does not, users are notified and the algorithm stops.

The *cycle detection stage* consists of the Visit method presented in the subsection detecting all cycles of length greater than 2 in an undirected graph above. Although it is discussed separately, the cycle classification and data saving stage is included in the detection one, since, once an undirected cycle is detected, it is immediately decomposed into the corresponding directed E-RD cycles, which are also then classified and their data saved into the corresponding *MatBase* metacatalog tables. During the DFS exploration, when a new node is discovered its *HasNeighbours* attribute (which is initialized in the first stage, when the vertex data is loaded into memory) is checked and if it is equal to its default value 0 the *CheckNeighbours* method of the *ERDEdgesList* class is called to check whether there are edges linking the current node to other ones. If this is the case, its HasNeighbours attribute value is set to 1 and the adjacency list is computed for it; otherwise, the current vertex is ignored and its *HasNeighbours* value is set to -1. For every adjacent node yet undiscovered, the number of edges between it and the current node is checked and if it is greater than 1 at least a length 2 cycle (potentially even more) has been detected, and the analysis, classification, and data saving stage begins for it. The same pattern applies when a length ≥ 3 cycle is detected, by visiting an already discovered adjacent node that is not a parent of the current one.

The *cycle classification* and *data saving stage* is indeed the most complex and time consuming one. For length 2 cycles this stage consists of the computations performed by the *AnalyzeLength2Loops* algorithm presented in the subsection Managing directed cycles of length 2. For cycles of length ≥ 3 it consists of the *AnalyzeERDLoops* algorithm presented in subsection Decomposing undirected cycles into the corresponding E-RD directed ones.

Moreover, when such a cycle is detected, another process takes place prior to running one of these algorithms that was not yet discussed: during the DFS exploration, once an undirected cycle is detected, the current vertex *u* and the adjacent already discovered vertex *v* are sent as parameters to a method called *AnalyzeLoop* that computes all the vertices involved in the cycle, starting from the current vertex *u* and going backwards on the edges that lead it to *u,* until it reaches vertex *v.* The discovered nodes are stored in an array, staring with node *u* on the first position and ending with node *v* on the last one. Afterwards, *AnalyzeLoop* checks whether at least one node belongs to the current db scheme, case in which it inserts the details of the cycle in the Loops table and the details of the nodes in the *LoopsNodes* table of the *MatBase* meta-catalog. Finally, *AnalyzeLoop* calls the *AnalyzeERDLoops* method providing it with the required parameters (the array of nodes and a reference to the undirected cycle which is to be decomposed into one or several directed E-RD cycles).

Here is the pseudocode of this method (where *node* is the first undirected cycle vertex, *node2* is the last corresponding one, *Length* is the cycle length, *parent* stores the latest discovered parent of the current node, vector *Nodes* stores all the current cycle nodes, and *nrLoops* stores the number of corresponding directed E-RD cycles discovered):

ALGORITHM 7: ***AnalyzeLoop****(node, node2)*

belongsToCurrentDb := if *node or node2* belong to the current db
then true else false;
Length := 0*;*
nrLoops := 0*;*
Nodes(0) *:= node;*
parent := latest discovered node parent;
while not *(parent* = NIL or *parent* = *node2) do*
if not *belongsToCurrentDb* Then
belongsToCurrentDb := if *parent* belongs to the current db
then true else false;
end if;
Length := *Length* + 1*;*
Nodes(Length) := *parent;*
parent := latest discovered *parent* parent*;*
end while*;*
Length := *Length* + 1*;*
Nodes(Length) := *node2;*

if *belongsToCurrentDb* Then

insert into *MatBase* table *Loops* current cycle details (db,

length, and sets of nodes ids and names);

nrLoops := *nrLoops* + 1*;*

Nodes(Length + 1) := node;

for *i* := 0*, Length + 1, i* + 1 do

fin(i) := 0*; fout(i)* := 0*;*

end for;

analyzeERDLoops(0, *Nodes, FUNCTIONS, fin, fout);*

end if;

Considering all these, we can now provide in the following subsection the pseudocode for the MatBase DFS detecting and classifying all E-RD cycles algorithm.

Matbase DFS-Based Algorithm for Detecting and Classifying All E-RD Cycles

ALGORITHM 8: **DFS detecting and classifying E-RD cycles**

Input: a *MatBase* database scheme

Output: the details of the cycles contained in the underlying undirected graph associated to the db scheme, those of the E-RD directed cycles of the db scheme, along with the details of the compound functions associated to these latter cycles

Strategy:

check the *ERDLoops* table for cycles belonging to the selected db

scheme;

if cycles found then

ask user to confirm deletion and re-computation of cycles data

for the desired db scheme; if not obtained, then stop;

end if;

open a connection to the selected db;

if cycles found then delete db cycle data from all *MatBase* tables;

instantiate the *ERDVerticesList* class;

instantiate the *ERDEdgesList* class;

instantiate the *ERDCompFuncList* class;

insert all relevant sets from the SETS table into the temporary

ERDVertices one;

for each vertex x in *ERDVertices* do

create a new *ERDVertex* object v and fetch its columns values;

add v to the list of vertices of the *ERDVerticesList* class;

insert from the *FUNCTIONS* table into the temporary *ERDEdges*

one all the foreign keys of *x*;

for each edge f in *ERDEdges* do

create a new *ERDEdge* object e and fetch its columns values;

```
        add e to the list of edges of the ERDEdgesList class;
    end for;
    for each compound function f in FUNCTIONS defined on x do
        create a new ERDCompFunc object cf and fetch its columns
values;
        add cf to the list of compound functions of the
ERDCompFuncList class;
    end for;
end for;
if there is a vertex belonging to the selected db scheme then
    for each ERDVertex v do
        if v.State = UNDISCOVERED then Visit(v);
    end for;
end if;
inform user about the number of cycles detected and the elapsed
execution time;
close the connection to the database;
```

Results and Discussion

Algorithm Complexity and Optimality

The time complexity of this algorithm exceeds by far the one of the most DFS-based algorithms, which is $O(|V|+|E|)$ (i.e. linear in the sum of the cardinals of the vertices and edges sets). Since fully explored vertices are also considered, except for the source node, the method Visit(u) might get called more than once for each vertex u: once when its state changes from UNDISCOVERED to DISCOVERED and possibly once or even several times more when the vertex is in a FINISHED state (once for every distinct path that it is a part of).

Therefore, as when given n nodes that are all directly connected (i.e. for every vertex pair (u, v) there is an edge between them), computing all the distinct paths between them using backtracking takes $O(n!)$ time, because all permutations of the n nodes must be considered, in a worst-case scenario this *MatBase* DFS-based algorithm runs in factorial time too, more precisely in $O((n-1)!)$ time, where n is the number of nodes in the corresponding E-RD graph (as the first node is visited only once).

Unfortunately, this is obviously huge for actual dbs that might have hundreds or even thousands (fundamental) tables and views, even if, generally, E-RD graphs are very loosely connected (i.e. a node is at most connected to some 8-10 others and for most them only 2-3 connections exist).

For example, even for the geographic db mentioned in the introduction, detecting and classifying its nearly 143,000 cycles on a DELL Latitude E7450 PC with an Intel I7 vPro x-64 CPU, 16GB RAM, and a 256GB SSD Samsung PM851 mSATA disk, running under MS Access 2016 on Windows 10, took *MatBase* more than 10 days of running (265h)!

Fortunately, from our more than 40 years' experience with hundreds of real world db schemes, constraints are merely associated only to short cycles of length at most 16, which is normal: as length grows, the probability that too distant object set elements (db table and view rows) are related by some constraints decreases sharply, especially for general cycles (which are the most and longest ones), but also for commutative ones, and even for circular ones.

Moreover, more than 90% of such constraints are associated to cycles of length at most 8. For example, in the geographic db constraints are associated to cycles having length at most 7; up to length 8, there are 35 circular cycles, 835 commutative, and 9,559 general ones: it took us a couple of months only to analyze all these 10,429 E-RD cycles, discovering that only 51 of them have associated constraints.

Consequently, before starting computations, the actual MatBase algorithm presented in this paper asks users whether they would like to run it only for cycles of a maximum length (the provided default being 16, which users can modify to any positive natural value, including 1 which stands for infinite) or for all of them.

For example, running the algorithm in the same conditions as above for the geographic db, but only for cycles under length 17, took less than 3 days (and yielded only 21,806 cycles, out of which 36 are circular, 850 commutative, and the rest being of the general type).

Please also note that, within the boundaries of currently available technologies, our implementation is, however, optimal: all needed data on nodes and edges, as well as the one of existing compound functions is read only once, at the beginning of the algorithm and then is stored in memory. Consequently, all subsequent computations, including discovering both undirected and directed graphs of any length, as well as declaring not yet existing compound functions that are paths of the directed E-RD cycles discovered are done solely in memory. Moreover, the undirected graph is not written to disk, but only temporarily kept in memory while the algorithm runs.

Unfortunately, as VBA has no technology yet available to bulk write on disk through SQL from memory, most of the time that the algorithm needs is spent on writing discovered directed cycles, as well as corresponding compound functions paths not yet known one by one. Although .NET has such a technology (namely method WriteToServer 0f the C# SqlClient framework, which writes into SQL Server tables from DataTable memory objects), we did not used it yet, for the sake of comparing the two implementations on equal grounds.

Algorithm Utility

First, detecting all cycles of both an undirected and a directed graph is an interesting problem per se in graph theory, which very probably has not been fully tackled before as no practical use was known for it. To our knowledge, only discusses E-RD cycle detection problems [6]. Consequently, we draw the attention of our colleagues teaching graph theory on this algorithm, suggesting them to include it in their lectures and labs.

Secondly, classifying directed cycles of length greater than 1 in the three types we defined (i.e. commutative, circular, and general) might be interesting within the study of sets, functions, and relations algebra, even if this is rather minor, but it helps getting a better understanding on function diagram commutativity and anti-commutativity, as well as on dyadic relation properties of compound autofunctions. Moreover, the generalized commutativity constraints (e.g. the one attached to the cycle presented in figure 3) are excellent real-world examples of closed first order predicate calculus with equality formulas.

Of course that the main utility of this algorithm is within the data modeling, db constraints theory, db and db software applications design and development fields of endeavor. As we discussed it in the introductive section, very many non-relational db constraints are attached to E-RD cycles and discovering them can be algorithmically done only after these cycles are discovered and classified [9].

Unfortunately, the state of the art in db and db software application design and development related to non-relational constraints is exclusively using ad-hoc approaches: very few db and/or software architects are aware of their types, optimal enforcing ways per type, algorithmic approaches to discover all of them for any sub-universe of discourse, and even of their paramount importance; only from experience and common sense (e.g. nobody may be simultaneously present in several places, no slot may be simultaneously occupied by several objects, etc.) are they considering some such constraints and enforce them in db software applications (through either RDBMS triggers in extended SQL or/and high level programming languages trigger-type methods).

Very many such constraints are only discovered in production, in time, generally by db software application users, who report them as bugs. Of course that it would be highly preferable that db and software architects discover all of them in the data modeling phase and then developers enforce them from the beginning (up until *MatBase* or other similar advanced DBMS products will become worldwide available, to provide automatic code generation for enforcing such constraints too, just like it is the case today with the relational ones).

Therefore, we are teaching (E)MDM basics even to undergraduate students during DB lectures and labs and the rest of it to M.Sc. ones during the Advanced DB lectures and labs. However, we are not teaching this algorithm to our students during these DB lectures and labs, but only provide them *MatBase* to use it for the labs, as a prerequisite of applying then the algorithm that assists discovery of non-relational constraints associated to E-RD cycles [9,11].

For example, in the above-mentioned geography db, there are 7 circular cycles that have a reflexivity constraint associated to one of their associated compound autofunctions, 4 having irreflexivity ones, 4 commutative-type cycles that commute and are unbreakable, 8 anti-commutativity constraints, 9 irreflexivities, 8 asymmetries, one acyclicity, and 12 generalized commutativity constraints.

Conclusions and Further Work

In summary, we have designed and implemented in both MatBase latest versions (for MS Access and C# and SQL Server) a DFS-based algorithm for detecting and classifying all cycles in a db E-RD, analyzed its complexity, optimality, and outlined its utility for both graph theory, sets, functions, and relations algebra, and, especially, data modelling, db constraints theory, db and db software application design and development practices.

Further work will be done first to drastically reduce running time when re-running the algorithm on a same db because of changes in its scheme: instead of deleting all its cycles data and then re-computing it from scratch, existing data should only be updated accordingly.

Then, to check whether for another range of cycle lengths there also exist constraints, we should add the facility to (re-) compute data only for cycles of lengths between a *min* and a *max* values (e.g. between 17 and 24).

Finally, another major improvement would be to (re-)compute data only for cycles of lengths between a *min* and a *max* values per type (e.g. between 2 and 12 for general ones, between 2 and 16 for commutative ones, and between 2 and 32 for circular ones).

For the C# and SQL Server *MatBase* version, we will store results in eight DataTable memory objects, one for each

involved metacatalog table, and use method WriteToServer of the C# SqlClient framework to write them at the end of the DFS exploration (thus minimizing writes to disk too: only eight such operations instead of millions of them).

Acknowledgments

Miss Sabina Maria Motoc, a M.Sc. student colleague of Mr. Adrian Mocanu, also under the scientific coordination of Prof. C. Mancas, designed and developed in parallel a similar algorithm to the one presented in this paper, but only for the MS Access *MatBase* version, which runs slightly faster, mainly because its implementation is not also explicitly object-oriented.

References

1. Codd EF. A relational model for large shared data banks. Commun of the ACM. 1970;377-387.
2. Abiteboul S, Hull R, Vianu V. Foundations of Databases. Addison-Wesley. 1995.
3. Mancas C. Conceptual Data Modeling and Database Design: A Completely Algorithmic Approach. Volume I: The Shortest Advisable Path. Waretown:Apple Academic Press;2015.
4. Mancas C, Dorobantu V. On enforcing relational constraints in MatBase. London Journal of Research in Comp. Sci. and Technology 2017:39-45.
5. Chen PP. The entity-relationship model: Toward a unified view of data. ACM Trans on DB Syst. 1976:9-36.
6. Thalheim B. Fundamentals of Entity-Relationship Modeling. Springer-Verlag. 2000.
7. Mancas C. A Deeper Insight into the Mathematical Data Model. In Proceedings of the 13th Intl Seminar on DBMS (ISDBMS'90). 1990:122–134.
8. Mancas C. On knowledge representation using an Elementary Mathematical Data Model. In Proceedings of the 1st International Conference on Information and Knowledge Sharing (IKS'02). 2002:206-211.
9. Mancas C. Conceptual Data Modeling and Database Design: A Completely Algorithmic Approach. Volume II: Refinements for an Expert Path. Waretown:Apple Academic Press;2018.
10. Papamanthou C. Depth First Search & Directed Acyclic Graphs. A Review for the Course Graph Algorithms. 2004.
11. Mancas C. On Modeling Closed E-R Diagrams Using an Elementary Mathematical Data Model. In Proceedings of the 6th Conference on Advances in DB and Inf Syst. 2002:65-174.