Edgar H. Sibley
Panel Editor

An abstract graph module that allows for easy and secure programming of a great number of graph algorithms is implemented by symmetrically stored forward and backward adjacency lists, thus supporting edge-oriented traversals of general directed and undirected graphs.

A VERSATILE DATA STRUCTURE FOR EDGE-ORIENTED GRAPH ALGORITHMS

JÜRGEN EBERT

It is widely accepted that graphs are a useful medium for modeling relevant parts of reality in computer programs. Graphs are rather natural models for road maps, electrical networks, chemical structure formulas, data and control flow of computer programs, state spaces of discrete games, sociological diagrams, timetables, etc. In addition, many discrete problems in several areas (e.g., formal language theory, automata theory, compiler construction, operating-systems theory, operations research) can be transformed into equivalent graph problems and then solved using graph algorithms.

Representation greatly influences the efficiency of graph algorithms. Often, linearity can only be achieved through appropriate storage of adjacency information (see, e.g., [6]). There are several different ways of internally representing graphs in procedural languages, including adjacency matrices, sequential or linked adjacency lists, and edge lists (cf. [2]). On the other hand, good and clear algorithm design is greatly enhanced by rather abstract graph representations that include operations on the graph, as well as control statements (e.g., `for`-loops) triggered by the graph.

In this article we describe an abstract module for graph handling that is especially suited for the edge-oriented paradigm of programming graph algorithms, and show how this module can be implemented efficiently in Algol-like languages. This graph realization is of the adjacency-list type and is suitable for directed and undirected graphs (with multiple edges allowed). Undirected graphs are represented as directed graphs through arbitrary assignment of directions to every edge (and not through storage of the corresponding symmetric graph). This representation has been used successfully in a number of applications; for instance, it has been used as a tool for representing the graphs in the EMS project on the implementation of functional languages [3].

GRAPHS

There is a large variety of graph types. Depending on the area of application, graphs can be directed or undirected, weighted or unweighted, and ordered or unordered. Multiple edges and loops are either permitted or forbidden. Here, we present a graph representation that is suitable to all of these variants ([1] and [5] are introductory books on graph theory; [2] and [4] introduce graph algorithms).

Using a very general type of graph definition, a

(finite) *directed graph* $G = (V, E, \alpha, \omega)$ consists of a finite nonempty set V of *vertices*, a finite set E of *edges* (with $V \cap E = \emptyset$), and two functions $\alpha: E \rightarrow V$ (denoting the *start vertex*) and $\omega: E \rightarrow V$ (denoting the *end vertex* of each edge). If there is an edge e with vertices $v = \alpha(e)$ and $w = \omega(e)$, then w is a *successor* of v and v is a *predecessor* of w. Those edges e with $v = \alpha(e)$ ("e goes out of v") constitute the *forward star* of v. Analogously, the *backward star* of v consists of those edges e with $v = \omega(e)$ ("e goes into v").

On the other hand, a (finite) *undirected graph* $G = (V, E, \varphi)$ contains only one function $\varphi: E \rightarrow \{W \subseteq V \mid 1 \leq |W| \leq 2\}$, assigning up to two vertices to every edge. A *self-loop* is an edge e with $|\varphi(e)| = 1$. If $v \in \varphi(e)$, we say v and e are *incident*. The *star* of a vertex v consists of those edges that are incident with v. The *degree* $\gamma(v)$ of v is the number of edges incident with v, where self-loops are counted twice.

For a directed graph $G = (V, E, \alpha, \omega)$, the *underlying* undirected graph $H = (V, E, \varphi)$ is given by $\varphi(e) = \{\alpha(e), \omega(e)\}$. Thus, all concepts and algorithms defined for undirected graphs can also be used for directed graphs through simple reference to their underlying undirected graphs. Thus, an edge e and a vertex v are incident if $v = \alpha(e)$ or $v = \omega(e)$, and e is a self-loop if $\alpha(e) = \omega(e)$.

With the implementation given below, the representation of the underlying undirected graph is *identical* to the representation of the graph itself. Thus, algorithms that were designed for undirected graphs (e.g., algorithms for finding spanning trees, testing (bi-)connectivity) can be executed on directed graphs without further adaptation.

EDGE-ORIENTED GRAPH ALGORITHMS

The graph module described here strongly supports an edge-oriented way of handling graphs. This paradigm makes programming more secure and is at the same time suitable for handling graphs with multiple edges.

The following conventions apply to edge-oriented programming of graph algorithms:

(1) Neighborhoods of vertices are traversed by reference to the *edges* incident with a given vertex.

(2) Edges are regarded as objects having two *states*. They may be pointing outwards (positive sign) or pointing inwards (negative sign).

Thus, for processing the successors of a given vertex v, one must proceed as follows:

```
for all edges e out of v do
  let w be the other vertex of e;
  process w
od.
```

```
procedure DFS (v : vertex);
  LOWPT[v] := NUMBER[v] := NUM := NUM + 1;
  for all e incident with v do
    let w be the other vertex of e;
    if NUMBER[w] = 0 then
      PARENT[w] := e;
      DFS(w);
      if LOWPT[w] ≥ NUMBER[w] then
        e is bridge
      fi;
      LOWPT[v] := min (LOWPT[v], LOWPT[w])
    else
      if NUMBER[v] ≥ NUMBER[w]
      and e is not a tree edge then
        LOWPT[v] := min (LOWPT[v], NUMBER[w])
      fi
    fi
  od;

NUM := 0;
for all v in V do
  PARENT[v] := NUMBER[v] := 0
od;
for all v in V do
  if NUMBER[v] = 0 then
    DFS(v)
  fi
od.
```

FIGURE 1. An Edge-Oriented Pseudocode Version of a Bridge Detection Algorithm

```
edge procedure first_out (v : vertex)
  returns first edge going out of v.
edge procedure next_out (e : edge)
  returns edge following e in
  forward star of alpha(e).
edge procedure first_in (v : vertex)
  returns first edge going into v.
edge procedure next_in (e : edge)
  returns edge following e in
  backward star of omega(e).
edge procedure first (v : vertex)
  returns first edge incident with v.
edge procedure next (e : edge)
  returns edge following e in
  star of this(e).
vertex procedure first_vertex ( )
  returns first vertex of the graph.
vertex procedure next_vertex (v : vertex)
  returns vertex following v in the graph.
edge procedure first_edge ( )
  returns first edge of the graph.
edge procedure next_edge (e : edge)
  returns edge following e in the graph.
```

These procedures return a `nil` value if there is no object to be returned.

FIGURE 2. Traversal Procedures for the Translation of the Pseudocode `for`-Statements

The sign of the edges makes it possible to talk about the "other" vertex.

Using loops over edges and giving a state to the edges allow a straightforward translation for **for**-loops into a combination of a **while**-loop and some standard function calls, since signed edges contain enough information to (re)enter a **while**-loop to process the next edge. This is not possible for loops over (e.g., successor) vertices. Furthermore, traversing neighborhoods by explicitly looking at all incident edges clarifies the fact that successors are listed more than once, if multiple edges exist. (A vertex-oriented **for**-loop is often valid only for graphs without multiple edges.)

In addition, the sign on the edges allows sufficient information to be kept for deferred processing, if—as in some search algorithms—edges are stored for later use in an intermediate data structure. When an edge is retrieved, its sign helps to deduce its provenance. Assigning directions (signs) to edges also helps to distinguish incoming from outgoing edges during undirected searches in directed graphs. Furthermore it helps to describe the direction of edges in (undirected) cycles and/or cuts of directed graphs.

As an example, Figure 1 shows an edge-oriented pseudocode version of a bridge detection algorithm (derivable from [6]). Note that this algorithm works on undirected graphs, though the input graph might be directed. Here, the PARENT-entries, which denote a spanning tree in a multigraph, contain edges (instead of vertices), since there is at most one incoming tree edge for every vertex.

GRAPH OPERATIONS

We now give the description of a module (in the sense of an abstract data structure) for implementing edge-oriented algorithms in Algol-like languages.

For programming a pseudostatement like the **for**-loop over the outward star from above, we use traversal functions `first_out` and `next_out` and an auxiliary function `omega` according to the following:

```
e := first_out (v);
while e <> 0 do
  w := omega (e);
  process w;
  e := next_out (e)
od.
```

Note that this is a very straightforward way of translating **for**-loops, which is made possible by the "direction" (sign) assigned to every edge value. Since the edges denoted by e point outward, it is possible to identify the "next" edge as the next one going out of the same start vertex.

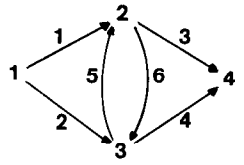
Figure 2 lists the traversal functions necessary for the translation of the pseudocode **for**-statements. Note that these functions assume an arbitrary but fixed order on all sets. For handling "signed" edges, some auxiliary transfer functions like those listed in Figure 3 are necessary. Figure 4 shows how the bridge detection algorithm of Figure 1 can be pro-

```
vertex procedure alpha (e : edge)
  returns start vertex of e.
vertex procedure omega (e : edge)
  returns end vertex of e.
vertex procedure this (e : edge)
  returns start vertex of e, if e is positive,
  and end vertex, otherwise.
vertex procedure that (e : edge)
  returns end vertex of e, if e is positive,
  and start vertex, otherwise.
edge procedure normal (e : edge)
  returns edge e with positive direction.
edge procedure reverse (e : edge)
  returns edge e with reversed direction.
```

FIGURE 3. Auxiliary Transfer Procedures for Handling "Signed" Edges

```
procedure DFS (v : vertex);
  LOWPT[v] := NUMBER[v] := NUM := NUM + 1;
  e := first (v);
  while e <> 0 do
    w := that (e);
    if NUMBER[w] = 0 then
      PARENT[w] := normal (e);
      DFS (w);
      if LOWPT[w] ≥ NUMBER[w] then
        output (e, "is bridge")
      fi;
      LOWPT[v] := min (LOWPT[v], LOWPT[w])
    else
      if NUMBER[v] ≥ NUMBER[w]
      and normal(e) <> PARENT[v] then          (*)
        LOWPT[v] := min (LOWPT[v], NUMBER[w])
      fi
    fi;
    e := next (e)
  od;
NUM := 0;
v := first_vertex ( );
while v <> 0 do
  PARENT[v] := NUMBER[v] := 0;
  v := next_vertex (v)
od;
v := first_vertex ( );
while v <> 0 do
  if NUMBER[v] = 0 then
    DFS (v)
  fi;
  v := next_vertex (v)
od.
```

FIGURE 4. A Concrete Program for Bridge Detection, Showing How the Procedures from Figures 2 and 3 Can Be Used



V = {1, 2, 3, 4}
E = {1, 2, 3, 4, 5, 6}

	1	2	3	4	5	6
α	1	1	2	3	3	2
ω	2	3	4	4	2	3

	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6
Node:	2	3	3	2	1	1		2	3	4	4	2	3
Next:	0	6	0	-4	4	3		2	0	-5	5	-6	0
First:								1	-1	-2	-3		

FIGURE 5. A Sample Graph and Its Array Representation

grammed using these procedures. Note that this program, though designed for undirected graphs, also handles directed graphs without any further action or adaptation.

Since edges are signed objects, some care has to be taken in testing edge equality. In the line marked with a star (*) in Figure 4, both sides of the inequality sign were simply normalized.

The next section shows that all operations used thus far can be implemented efficiently. A great number of graph algorithms can be formulated using only the operations described here. In [2], for instance, it is shown that all the generally used efficient graph algorithms can be based on the operations described here (except for the class of all-pair-shortest-path algorithms, which usually use some kind of `is_edge(v, w)`-test).

GRAPH REPRESENTATION

Adjacency lists are usually used to store the successors of every vertex by means of linked lists. We store the predecessors, as well, to enable algorithms on the underlying undirected graph. Using some practical "tricks," we get a storage schema, which implements all the operations of our graph module in an efficient way.

Let G = (V, E, α, ω) be a given directed graph with $|V| = n$ and $|E| = m$. We number the vertices in V from 1 to n, and the edges in E from 1 to m.

All concrete information about the vertices and edges (e.g., names, weights, lengths, markings) can now be stored separately in arrays of length n or m, respectively. (Note that this is one of the main advantages of adjacency list representations over adjacency matrices, at least when the graph is sparse.) Thus, values of vertices and edges have only to be stored once, even for undirected graphs.

We use these numbers as names for the corresponding objects. But we take care not to confuse vertex i with edge i (for $1 \le i \le \min(n, m)$). Thus, we take

$$\text{vertex} = 0 \quad .. \; n$$

$$\text{edge} = -m \; .. \; m$$

as types, with the integer sign as the direction indicator for edges and with zero as a `nil`-value for vertices and edges.

```
edge procedure first_out (v : vertex);
  e := FIRST[v];
  while e < 0 do
        e := NEXT[e]
  od;
  return e.
edge procedure next_out (e : edge);
  e := NEXT[abs(e)];
  while e < 0 do
        e := NEXT[e];
  od;
  return e.
edge procedure first_in (v : vertex);
  e := FIRST[v];
  while e > 0 do
        e := NEXT[e]
  od;
  return e.
edge procedure next_in (e : edge);
  e := NEXT[-abs(e)];
  while e > 0 do
        e := NEXT[e]
  od;
  return e.
edge procedure first (v : vertex);
  return FIRST[v];
edge procedure next (e : edge);
  return NEXT[e].
vertex procedure first_vertex( );
  return 1.
vertex procedure next_vertex (v : vertex);
  return if v = n then 0 else v + 1 fi.
edge procedure first_edge ( );
  return 1.
edge procedure next_edge (e : edge);
  return if e = m then 0 else e + 1 fi.
```

FIGURE 6. Implementation of the Traversal Procedures from Figure 2, Assuming Validity of Input Parameters and n and $m \ge 1$

Using an array

```
NODE : array [edge] of vertex
```

we can store $\alpha(e)$ in NODE[$-e$] and $\omega(e)$ in NODE[$+e$] for every edge e. This allows all edge-list-oriented algorithms to be used on our structure by accessing the array NODE alone.

Conversely, the adjacency lists for every vertex v are made traversable by using array indexes as links:

```
FIRST : array [vertex] of edge

NEXT : array [edge] of edge
```

These arrays link all edges incident with a vertex v to v by using the chain of indexes starting at FIRST[v], following the NEXT-entries, and ending with a zero-entry. The indexes are positive for edges going out of v and negative for edges going into v. Then, the nonzero entries in the FIRST/NEXT-arrays *are* the signed edges *themselves*, with their direction seen from the corresponding vertex. Figure 5 gives an example by showing a sample graph and its array representation. Figure 6 shows how the traversal procedures of Figure 2 can be implemented using our structure, and Figure 7 gives the implementation of the auxiliary functions of Figure 3. (Note that in practical applications all generally usable procedures should check their arguments. These checks have been skipped to simplify the presentation.)

These implementations of the traversal functions lead to a complexity of **for**-loops over (for/backward) stars of a vertex v proportional to $\gamma(v)$. Equally, **for**-loops over V and E have a complexity proportional to n and m, respectively. All the implementations of the auxiliary functions apparently use constant time.

The first/next-pair of functions enumerates self-loops twice. If this is not wanted, first and next should be modified to ignore negative edge values e, if

```
NODE[e] = NODE[-e].
```

The graph module can of course be augmented by additional operations, if they are needed, such as procedures for determining degrees, for testing the existence of edges, etc. Note that is_edge(v, w)-tests have a complexity at least proportional to min ($\gamma(v)$, $\gamma(w)$).

RECONSTRUCTION AND COMPRESSION

The NODE-array alone (being an edge-list representation of the graph) already contains all incidence information. Thus, for example, for external storage,

```
vertex procedure alpha (e : edge);
  return NODE[-abs(e)].
vertex procedure omega (e : edge);
  return NODE[abs(e)].
vertex procedure this (e : edge);
  return NODE[-e].
vertex procedure that (e : edge);
  return NODE[e].
edge procedure normal (e : edge);
  return abs(e).
edge procedure reverse (e : edge);
  return -e.
```

FIGURE 7. Implementation of the Auxiliary Procedures from Figure 3, Assuming Validity of Input Parameters

this array suffices. Figure 8 shows how the graph representation described in the previous section can be reconstructed from its NODE-array in linear time. This algorithm traverses the edge list in reverse order. It inserts each (positive) edge e at the front of the adjacency list of $\alpha(e)$ and inserts $-e$ to the list of $\omega(e)$.

```
for v := 1 to n do
  FIRST[v] := 0
od;
for e := m downto 1 do
  NEXT[e] := FIRST[NODE[-e]];
  FIRST[NODE[-e]] := e;
  NEXT[-e] := FIRST[NODE[e]];
  FIRST[NODE[e]] := -e
od.
```

FIGURE 8. Reconstruction of the Graph Representation from Its NODE-Array in Linear Time

This algorithm is *order preserving*: If the NODE-array is compatible with all the adjacency lists, then the adjacency lists are reconstructed as they were before. On the other hand, if the NODE-array gets sorted (e.g., according to some weight), then all adjacency lists are sorted as well—after reconstruction.

This procedure might also be used for simplifying graph *input* to simply reading an edge list. Graph *output* can be performed by just printing the edge list while traversing the NODE-array. If, however, the NODE-array order is not compatible with the orders of the adjacency lists, some topological sorting has to be done when the graph is written to an external

```
procedure test_and_mark (e : edge):
  if e <> 0 and not is_marked (e) then
    mark (e);
    if is_marked (-e) then
      enqueue (abs(e))
    fi;
    if NODE[-e] = NODE[e] then
      test_and_mark (-e)
    fi
  fi;

init_queue ( );
init_marking ( );
for v := 1 to n do
  test_and_mark (FIRST[v])
od;
while not is_empty_queue ( ) do
  e := extract_front_of_queue ( );
  output (NODE[-e], "→", NODE[e]);
  test_and_mark (NEXT[e]);
  test_and_mark (NEXT[-e])
od.
```

FIGURE 9. A Linear-Time Algorithm for Compressing the Graph Structure into an Edge-List

medium. Otherwise, the original order on the stars cannot be reconstructed. (This incompatibility might occur, for instance, if the graph was constructed dynamically using the procedures presented in the next section.)

Figure 9 gives a linear time algorithm for compressing the graph structure into an edge list. But here we need some (linear) additional work space for queueing (up to m) normalized edges, and marking (up to $2m$) signed edges. This algorithm prints an edge e only if its predecessors in the adjacency list of $\alpha(e)$ as well as $\omega(e)$ have all been printed. To decide this property, a marking of e and $-e$ is used. A queue helps to keep track of all printable edges. This algorithm fails (by not printing all edges) if there is no NODE-array ordering that is compatible with the adjacency-list ordering.

DYNAMIC GRAPHS

Many applications use graphs whose size and structure change during execution. This implies a need for procedures to create and delete vertices and edges. The representation presented in this article can easily be extended for graphs with a (moderately) varying number of vertices and/or edges, as long as a maximum number (*NMAX*/*MMAX*) can be given for both. Figure 10 gives the procedures necessary for handling dynamic graphs.

To implement dynamic graphs, we link all unused (nonnegative) entries in the FIRST/NEXT arrays. Using the zero-entries in these arrays (which are unused up to now) as a start pointer, we can link the unused entries together following a stack discipline. To distinguish used from unused entries, we add a large value LARGE (preferably *MMAX*) to the link values in FIRST and NEXT, if we use them for chaining unused entries.

In this case the graph should be initialized as an empty graph using the following procedure:

```
procedure init ( );
  for v := 0 to NMAX do
    FIRST[v] := v + 1 + LARGE
  od;
  for e := 0 to MMAX do
    NEXT[-e] := 0;
    NEXT[e]  := e + 1 + LARGE
  od;
  n := m := 0.
```

(In this case the procedures for traversing the vertex and edge sets have to be adapted.)

It is even possible to keep track of the order in which the edges are added to the vertices by using an additional array

```
LAST : array[vertex] of edge
```

to keep the last edge entry for every vertex. (This applies to forward and backward stars as well as to (undirected) stars.) In this case all loops varying over adjacency sets traverse these sets in the order of edge creation. This leads to ordered graphs, where the edges incident with every vertex are linearly ordered. Figure 11 shows how the creation/deletion procedures are implemented; two auxiliary procedures are given in Figure 12.

The create procedures use constant time. The complexity of delete_vertex is proportional to $\gamma(v)$. The delete_edge(e) procedure uses time proportional to $\gamma(\alpha(e)) + \gamma(\omega(e))$. It could be made constant time by using double chains on edges, but this does not seem worthwhile.

```
vertex procedure create_vertex ( );
  returns a new vertex.
procedure delete_vertex (v : vertex);
  deletes vertex v.
edge procedure create_edge (v, W : vertex);
  returns a new edge.
procedure delete_edge (e : edge);
  deletes edge e.
```

FIGURE 10. Creation and Deletion Procedures for Handling Dynamic Graphs

```
vertex procedure create_vertex ( );
  v := FIRST[0] − LARGE;
  FIRST[0] := FIRST[v];
  FIRST[v] := LAST[v] := 0;
  n := n + 1;
  return v.
procedure delete_vertex (v : vertex);
  while FIRST[v] <> 0 do
    delete_edge (FIRST[v])
  od;
  FIRST[v] := FIRST[0];
  FIRST[0] := v + LARGE;
  n := n − 1.
edge procedure create_edge (v, w : vertex);
  e := NEXT[0] − LARGE;
  NEXT[0] := NEXT[e];
  NEXT[e] := 0;
  m := m + 1;
  create_entry (v, e);
  create_entry (w, −e);
  return e.
procedure delete_edge (e : edge);
  e := abs (e);
  delete_entry (NODE[−e], e);
  delete_entry (NODE[e], −e);
  NODE[−e] := NODE[e] := NEXT[−e] := 0;
  NEXT[e] := NEXT[0];
  NEXT[0] := e + LARGE;
  m := m − 1.
```

FIGURE 11. Implementation of the Creation and Deletion Procedures, Assuming Validity if Input Parameters and without Checking Overflow Conditions

```
procedure create_entry (v : vertex, e : edge);
  if FIRST[v] = 0 then
    FIRST[v] := e
  else
    NEXT[LAST[v]] := e
  fi;
  LAST[v] := e;
  NODE[−e] := v.
procedure delete_entry (v : vertex, e : edge);
  if FIRST[v] = e then
    FIRST[v] := NEXT[e];
    if LAST[v] = e then
      LAST[v] := 0
    fi
  else
    i := FIRST[v];
    while NEXT[i] <> e do
      i := NEXT[i]
    od;
    NEXT[i] := NEXT[e];
    if LAST[v] = e then
      LAST[v] := i
    fi
  fi.
```

FIGURE 12. Auxiliary Creation and Deletion Procedures

CONCLUSION

Following the paradigm of edge orientation, our software module for general graphs and their implementation on von Neumann machines allows for easy and secure programming of a great number of graph algorithms, since traversal of forward and backward stars as well as edge enumeration is particularly easy. Since there is no distinction between a directed graph and its underlying undirected graph, our module allows a very straightforward transliteration of published algorithms into concrete programming code.

The module has successfully been used in various applications. A version written in the C language is one of the basic cornerstones of the EMS system for the implementation of functional languages by graphs.

REFERENCES

1. Berge, C. *Graphs and Hypergraphs.* North-Holland, Amsterdam, 1973. A textbook on graph theory.
2. Ebert, J. *Effiziente Graphenalgorithmen.* Aula, Wiesbaden, West Germany, 1981. A textbook on graph algorithms and their implementation in Algol-like languages using abstract data and refinements.
3. Ebert, J. Implementing a functional language on a vom Neumann computer. Tech. Rep. 3/85, EWH Koblenz, Fachbericht Informatik, Mar. 1985. A report on an implementation technique for functional languages using attributed and ordered directed graphs.
4. Even, S. *Graph Algorithms.* Pitman, Marshfield, Mass., 1979. A textbook on graph theory featuring algorithms.
5. Harary, F. *Graph Theory.* Addison-Wesley, Reading, Mass., 1969. A textbook on graph theory.
6. Tarjan, R.E. Depth-first search and linear graph algorithms. *SIAM J. Comput. 1,* 2 (1972), 146–160. A fundamental paper showing that some connectivity problems are solvable in linear time using depth-first search on adjacency-list representations of graphs.

CR Categories and Subject Descriptors: E.1 [**Data**]: Data Structures—*arrays; graphs; lists;* E.2 [**Data**]: Data Storage Representations—*contiguous representations; linked representations;* F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*computations on discrete structures;* G.2.2 [**Discrete Mathematics**]: Graph Theory—*graph algorithms*

General Terms: Algorithms

Additional Key Words and Phrases: Edge-oriented algorithms, graph traversal

Received 6/86; accepted 12/86

Author's Present Address: Jürgen Ebert, EHW Koblenz, Informatik, Rheinau 3–4, 5400 Koblenz, West Germany.