DDB Graph Operations for the IFS/2

N.E.J. Dewhurst, S.H. Lavington and J. Robinson
Department of Computer Science
University of Essex
Colchester CO4 3SQ

Abstract: The IFS/2 is an add-on hardware unit which provides whole-structure operations on data such as sets, relations and graphs. This report, a sequel to CSM-164 and CSM-168, describes 16 procedures which perform operations on graphs held in the IFS/2's persistent memory. The graph operations are especially intended to support recursive query evaluation in deductive databases (DDB), though some of the operations clearly have a wider application. Additional IFS/2 graph operations may be implemented at a later date.

1. Introduction

The IFS/2 is an add-on hardware unit which is being designed and built by the Intelligent File Store group at the University of Essex [1, 2]. The IFS/2 provides support for knowledge-based systems by both storing and manipulating persistent structures such as sets, relations and graphs. The IFS/2 has a low-level procedural interface, available to programmers as a collection of C library procedures. This collection defines the IFS/2's External Procedural Interface (EPI).

At present the IFS/2's External Procedural Interface is made available as 44 C library procedure calls. The 44 procedures are grouped as follows:

- a) Housekeeping (e.g., 'open', 'close'): 5 procedures
- b) Search, insert, delete: 3 “
- c) Tuple descriptor management: 3 “
- d) Lexical token conversion (for handling strings): 5 “
- e) Label management: 5 “
- f) Relational algebraic operations: 7 “
- g) Graph descriptor management: 3 “
- h) DDB graph operations: 13 “

Report CSM-164 [3] describes the EPI procedures in groups (a) to (e). CSM-168 [4] describes group (f). This report describes groups (g) and (h).

A software simulator, known as SIERRA, has been written for the IFS/2. SIERRA is a suite of C procedures and associated data structures that gives, via the same EPI command structure, the same functionality as the actual IFS hardware.
This present report describes SIERRA procedures for performing operations such as transitive closure, reachable node set, composition of relations, and tests for paths and cycles within graphs. Theses are dealt with in Sections 4 and 5. Since the particular graph operations have been chosen to support deductive databases (DDB), Section 3 gives the background to recursive query handling in DDB applications.

Readers familiar with the IFS/2 and SIERRA should now skip to Section 3. For those new to the ideas, Section 2 gives a brief review of IFS/2 terminology. Further details are contained in [3] and [4], which should be regarded as companion reports.

2. Review of IFS Terminology

The IFS/2 operates according to the active memory principle, in which whole-structure operations are performed by composed commands. Since the IFS/2 has a one-level associative (i.e. content-addressable) memory, stored objects of various granularities may be identified and accessed by logical name or logical descriptor, rather than by physical address.

The IFS/2's basic unit of information is the tuple. An entry in a Tuple Descriptor Table (TDT), held within the IFS/2, contains useful information about each set of tuples. A tuple-set is referred to by a class ID, which is used by the IFS to access the TDT information.

Tuples and tuple-sets are held by the IFS/2 in its Associative Tuple Store (ATS). This consists physically of disc(s) and cache, the whole being integrated into a one-level memory by a 'paging' technique known as semantic caching [5]. The ATS cache section, which is implemented as many SIMD-parallel search engines, is also used to implement the relational and graph operations. The present IFS/2 prototype hardware unit contains 27 Megabytes of associative cache, backed by 2.2 Gigabytes of associatively-accessed disc [2]. The design is modularly extensible.

The IFS/2 is an add-on unit, attached to a host computer via a standard SCSI channel. User programs running on the host computer access the IFS/2 via the 44 EPI commands. For the purposes of understanding the graph operations of Sections 4 and 5, users may imagine that the IFS/2 consists in essence of four inter-related sub-units:

i) the Associative Tuple Store which hold, amongst other structures, the base relations from which graphs may be derived;
   ii) a Tuple Descriptor Table;
   iii) a Graph Descriptor Table;
   iv) a means for processing relations and graphs.
3. Graphs and DDBs

A Relational Database contains a collection of n-ary relations. Users can query the database in ways which use relations as the edge relations of graphs. A DDB is an extended Relational Database, allowing new facts to be deduced from stored facts. Rule clusters define virtual relations (sets of deduced facts) to be used in queries or other rules.

Conjunctive queries are a characteristic of deductive databases. The query is true if all the predicates in the conjunction are true.

For example, the query:

\[ ? - q(c, w), r(w, z), s(z, y). \]

where \( c \) is a constant, will be true if there is an instantiation of variables \( w, z \) and \( y \) from tuples in the base relations. A feature of DDBs which differs from Logic Programming systems is that a DDB expects all possible instantiations to be returned as the answer to the query, whereas the corresponding query in Prolog would return only a single instantiation to prove the theorem denoted by the query. The DDB returns all proofs of the theorem. Each proof is a tuple of values for variables \( w, z \) and \( y \). Another difference in terminology between Logic Programming systems and DDBs is that a DDB is regarded as Incomplete if it does not return all possible proofs, whereas an LP system is incomplete only if its search strategy might fail to find one or more of the proofs. The DDB finds and returns all proofs, as a Result Relation. Hence breadth-first exploration of the database digraph, assisted by parallel hardware, is appropriate in DDBs. A depth-first strategy (with backtracking) is more suitable in Logic Programming.

A conjunctive query in which all adjacent predicates in the conjunction share one or more attributes can be regarded as a graph traversal. The conjunction is usually implemented in DDB systems by a sequence of Equi-Joins, where the Join attributes are the columns common to adjacent relations in the query. This succession of Joins provides the so-called Sideways Information Passing (SIP) during query evaluation, and the SIP Graph for the query indicates the graph traversal aspect of these chaining conjunctions:

\[ ? - q(c, w), r(w, z), s(z, y). \]

Relation \( q \) provides a set of edges \{c ->w\} to be followed by a set of edges \{w->z\} from relation \( r \), which in turn are followed by a set of edges \{z->y\} from relation \( s \).

A feature of these DDB graphs is the absence of isolated nodes. The graph is completely specified by a set of edges, without the separate set of vertices needed in general graph definitions.

Graph traversal applies to n-ary as well as binary relations, e.g. in evaluating
the query:

\[ ? \rightarrow q(a, b, w), \ r(w, z), \ s(z, y, c). \]

where a, b and c are constants. Relation q provides a set of edges \{[a,b]\rightarrow w\} to be followed by edges w\rightarrow z from relation r, which in turn are followed by edges z\rightarrow[y,c] from relation s.

The start node [a,b] is a pair of values, rather than the single-valued node in the previous example. Nodes are tuple-valued, in general.

DDBs store rules denoting conjunctive subqueries, used during query evaluation. E.g.:

\[
p(x, y) \ :- \ q(x, v, w), \ r(w, z), \ s(z, y, u).
\]

The head predicate, p, may itself occur in a conjunction, either in the query itself or in the body of another rule, so the distance of graph traversal involved in query evaluation can become significant.

Recursive rules further increase the distance to be travelled during traversal of the database digraph. For example, the following linear recursion:

\[
p(x, y) \ :- \ s(x, y).
p(x, y) \ :- \ q(x, z), \ p(z, y).
?\rightarrow \ p(a, y).
\]

is equivalent to an infinite number of conjunctive queries, with increasing chain length, obtained by substitution:

\[
s(a, y).
q(a, z0), \ s(z0, y).
q(a, z0), \ q(z0, z1), \ s(z1, y).
q(a, z0), \ q(z0, z1), \ q(z1, z2), \ s(z2, y).
q(a, z0), \ q(z0, z1), \ q(z1, z2), \ q(z2, z3), \ s(z3, y).
\]

etc.

**DDB Relevance of IFS Graph Operations**

One of the most popular rule forms in DDBs is the Transitive Closure rule cluster:

\[
p(x, y) \ :- \ e(x, y).
p(x, y) \ :- \ e(x, z), \ p(z, y).
\]

or alternatively and equivalently:
\[ \begin{align*}
   p(x, y) & \leftarrow e(x, y). \\
   p(x, y) & \leftarrow p(x, z), \ p(z, y).
\end{align*} \]

or:
\[ \begin{align*}
   p(x, y) & \leftarrow e(x, y). \\
   p(x, y) & \leftarrow p(x, z), \ e(z, y).
\end{align*} \]

When any of these rule clusters is queried without constants:

\[- p(x, y). \]

the result is the transitive closure of base relation \(e\), provided by the \texttt{ifs\_trans\_close} operation (see Section 4).

However, when a TC rule cluster is queried by a query that includes one or more constant values, the constants identify nodes in the database digraph. For example, the query
\[- p(a, y). \]

denotes graph traversal starting at node \('a'\) to identify the set of nodes \(\{y\}\) reachable from \('a'\). This is the \texttt{ifs\_reachable\_nodes} operation. Similarly, the query
\[- p(x, b). \]

specifies graph traversal starting at node \('b'\) and traversing edges in the reverse of their declared direction. Query
\[- p(a, b). \]

means traverse the graph starting at node \('a'\) and searching for node \('b'\). This is the \texttt{ifs\_check\_path} operation.

The \texttt{ifs\_reachable\_edges} operation is useful in DDBs to extract from the database a subgraph relevant to a query, which may then be processed (using numerical methods) in a front-end general-purpose processor system.

The IFS operations providing specified Waves of responders are useful in evaluating \(n\)-sided recursions. Certain evaluation strategies are able to identify the set of wave numbers which will contain result values. These can utilise IFS waveset operations to retrieve results.

DDB systems usually support a range of evaluation strategies and choose an appropriate method of evaluation after analysing the query and its relevant rules. The method chosen for query evaluation may depend on whether or not the graph is cyclic. Certain efficient DDB query evaluation strategies can only be used on acyclic database digraphs or they will not terminate. Although a graph in the database may contain cycles, the part relevant to the current query may not. It is therefore useful if the database is able to follow paths from any specified node to ascertain whether the reachable subgraph is cyclic. The \texttt{ifs\_check\_cycles} allows the query-relevant part of the database to be tested for cycles.
4. Declaring Graphs

Information is stored in the IFS in the form of tuples, grouped into classes of tuples with equal numbers of fields. It is therefore natural to view the contents of the IFS as a relational database. [3] describes the facilities provided by the EPI for declaring new classes and inserting tuples into them, and [4] describes a set of in-store relational operations supporting SQL-style queries. The EPI graph interface builds on this relational view of the IFS by allowing the user to declare a view of the store as a set of graphs (binary relations) to which path-finding and other related operations may be applied.

IFS graphs are virtual graphs, in the sense that no data is associated explicitly with a stored graph. Rather, when declaring a new graph, the user provides a recipe describing how the arcs (binary tuples) of the graph are to be extracted from the data in the store. When a graph operation is to be performed on a graph, the graph is first realized in accordance with its recipe.

The simplest form of graph recipe is a triple (c, s, t), where c is a class ID, and s and t are two columns of c identified by their position, counting from the left. For each tuple x of c, there is an arc in the realized graph from node x.s to x.t. (s and t should be read as “source” and “target”).

More generally, a graph recipe may demand the chaining together of several stored relations. In effect, this means describing the graph as the result of composing several “simple” graphs - “simple” in the sense that they can be described as in the preceding paragraph. The general form of data for an IFS graph declaration is therefore an array of “simple” graphs, and the array members are composed in the order they appear in the array.

```c
#include "ifs.h"

simple_graphs = malloc(sizeof(IFS_edbgraph) * 3);
simple_graphs[0].c_no = c1;
simple_graphs[0].source = 1;
simple_graphs[0].target = 3;

simple_graphs[1].c_no = c2;
simple_graphs[1].source = 1;
simple_graphs[1].target = 2;

simple_graphs[2].c_no = c3;
simple_graphs[2].source = 3;
simple_graphs[2].target = 2;

graph_spec = malloc(sizeof(IFS_graph));
graph_spec->no_edbgraphs = 3;
graph_spec->edbgraphs = simple_graphs;

ifs_graph_declare (graph_spec, &new_id);
```

The box contains an example graph declaration. First an array of structures of
type IFSedbgraph is reserved, to represent the aforementioned “simple” graphs: “edb” is for “extensional database”. This array is then incorporated into a larger structure of type IFS_graph. Structures of this type represent graph recipes.

The IFS_graph structure, or rather its address, is then passed to the EPI function ifs_graph_declare(), which assigns the new graph an ID, and returns the ID to the user in the object new_id. new_id should be of type IFS_graph_id. The types IFS_graph, IFS_graph_id and IFSedbgraph are defined in the header file “ifs.h”.

An EPI graph operation may be performed on the new graph by passing its ID to the appropriate EPI function. The EPI graph operations are described in Section 5.

The figure below illustrates the idea of “chaining” relations by drawing the realization of the example declaration on an example database.

![Diagram](image)

The recipe assigned to an IFS graph may be recovered from the graph table by making a call to the EPI function ifs_graph_convert(), so:

```c
ifs_graph_convert (g_id, &recipe)
```

where g_id is of type IFS_graph_id, and recipe a pointer to a structure of type IFS_graph. An entry may be removed from the graph table via a call to ifs_graph_delete():

```c
ifs_graph_delete (g_id).
```
5. EPI Graph Operations

1. TRANSITIVE CLOSURE

The *transitive closure* of a binary relation is the smallest transitive relation containing the original relation as a subset. In terms of nodes and edges, this can be expressed as follows: the transitive closure of a graph A has an edge a-b for each pair of nodes a and b in A for which there is a path between a and b. It therefore summarizes information about connectedness within the graph.

The figure above provides a simple illustration. The unbroken lines represent the arcs of the original graph, which are also listed to the left of the diagram. The broken lines represent the arcs that are added in forming the graph’s transitive closure.

The graph EPI provides an operation forming the transitive closure of a stored graph. If the graph is assigned an ID \( g\_id \), the call has the following form:

\[
\text{ifs_trans_close} (g\_id, \ &result)
\]

Providing no error occurs, the result of this call is the formation of a new binary class containing a tuple \(<a, b>\) for each edge a-b of the transitive closure of graph \( g\_id \). The ID of this new class is deposited in \( \text{result} \).

**Note:** the result of this operation is not in itself an IFS graph. If further graph processing of the result set is needed, a new IFS graph having the result set as its data must be declared. A similar comment applies to each of the other EPI graph operations.

See section 4 for details of graph declarations.
2(i) REACHABLE NODE SET

A node \( n_1 \) of a graph A is said to be *reachable* from node \( n_2 \) if there is a path from \( n_2 \) to \( n_1 \) in A. For example, in the graph drawn above, node e is reachable from b because there is a path from b to e via a and d.

If the graph above is stored in the IFS and has ID \( g_{id} \), and node b is represented by the value node_b, the set of nodes reachable from b can be computed by making the call:

\[
\text{ifs_reachable_nodes}(g_{id}, \text{node}_b, \&\text{result}).
\]

The ID of a new class is returned in result. It is a unary class containing the tuples \( <a>, <b>, <d>, <e> \) and \( <f> \), representing the reachable nodeset of b.

2(ii) REACHABLE EDGE SET

An edge e of a graph A is said to be *reachable* from node n if e is part of some path originating at n. In the graph above, the edge a-d is reachable from b, as it is part of a path b-a-d-e which begins at b.

The Graph EPI provides a second unitary operation isolating reachable edges from a specified graph. If \( g_{id} \) and \( \text{node}_b \) are as in the previous example, the call:

\[
\text{if_reachable_edges}(g_{id}, \text{node}_b, \&\text{result})
\]

will obtain the ID of a new binary class containing the tuples \( <b, a>, <a, d>, <d, e> \) and \( <e, f> \), representing the reachable edgeset of b, and hence denoting the reachable subgraph.
A node \( n_2 \) in a graph \( A \) is said to be at a distance \( n \) from a node \( n_1 \) if there is a path of length \( n \) from \( n_1 \) to \( n_2 \) in \( A \). The set of nodes at a distance \( n \) from \( n_1 \) is called the \( nth \) wave of nodes from \( n_1 \).

The figure above illustrates this idea. The presence of a number \( r \) labelling an arc of the graph means that the arc is the last in some path of length \( r \) starting at \( b \). For example, a-d is labelled with the number five because it is the last arc in the path b-a-d-b-a-d. It follows that d is in the fifth wave from b. The members of the first five waves of nodes from b are listed to the left of the graph.

This particular example also illustrates an important point: in a graph with cycles, nodes on a path containing a cycle will appear in infinitely many different waves. For example, node d is in waves number 2, 5, 8 and so on; node f is in waves 4, 7 and 10.

The following EPI call can be used to generate the contents of wave

\[ \text{wave_no from node node in a graph with ID g_id:} \]

\[ \text{ifs_nth_wave_nodes (g_id, node, wave_no, &result)} \]

The result set is stored as a unary class in the ATS, and the class ID placed in result.

---

3(ii) Nth WAVE EDGES

In effect, the nth-wave-nodes operation returns the set of endpoints of all paths of length \( n \) beginning from some common node. It may in some cases be more
useful to have the last arc of each path instead: hence the concept of a wave of edges. In the example on the previous page, a-d is in the second wave of edges from b, and e-f and b-a are in the fourth.

The graph EPI call to generate wave wave_no of edges is:

    ifs_nth_wave_edges (g_id, node, wave_no, &result)

The result is a new binary class, whose class ID is placed in result.

4. CHECKING FOR PATHS

The graph EPI provides an operation to determine whether or not there is a path from some given node a to another given node b in a stored graph. Supposing that g_id is the ID of some stored graph, and node_a and node_b two nodes in it, the call:

    ifs_check_path (g_id, node_a, node_b)

returns the status code IFS_FOUND if there is a path from node_a to node_b, and IFS_NOT_FOUND if there is not.

5. RELATION COMPOSITION

Let R and S be binary relations. Then we may construct another relation, denoted by R•S, by collecting together all the pairs (x, y) for which for some y1 there is a pair (x, y1) in R and (y1, y) in S. We say we have composed R and S. R•S is their composition; if the two relations represent finite functions, it is the usual functional composition.

A composition operation has a place in a graph-oriented interface for this reason: if R_n denotes the set of tuples representing paths of length n in a graph R, then R_{n+1} can be got by composing R_n with R. Relational composition is thus an operation fundamental to graph processing that involves discovering paths.

If g1 and g2 are the IDs of two stored IFS graph, they can be composed by making the EPI call:

    ifs_compose (g1, g2, &result)

where, as usual, the value returned in result is the ID of the resulting class.
6(i) CHECKING FOR CYCLES

In graph terminology, a cycle is a path beginning and ending at the same node. A graph containing a cycle is said to be cyclic. The simplest example of a cycle is an arc (i.e., a path of length 1) whose source and target nodes are the same. In the graph drawn above, the path b-a-d-b is a cycle.

The graph EPI provides an operation to determine whether or not a stored graph is cyclic. Supposing that g_id is the ID of some stored graph, the call:

```c
ifsc_check_cycles (g_id)
```

will return the status code IFS_FOUND if the graph contains a cycle, or the code IFS_NOT_FOUND if it does not.

6(ii) CYCLES IN A SUBGRAPH

When checking for cycles, it may not be necessary to search the whole graph, but only that part of it which is accessible (reachable) from a given node. The graph EPI caters for this possibility. If g_id is as in 5(i), a call of the form

```c
ifsc_check_cycles_subgraph (g_id, node)
```

will return status code IFS_FOUND or IFS_NOT_FOUND, depending on whether or not the part of the graph reachable from node is cyclic.

For example, if g_id identifies the graph in the previous figure,

```c
ifsc_check_cycles_subgraph (g_id, node_a)
```

where node_a represents node a will return IFS_FOUND, owing to the presence of the cycle a-d-b-a. On the other hand,

```c
ifsc_check_cycles_subgraph (g_id, node_e)
```

will return IFS_NOT_FOUND, as there are no cycles in the part of the graph
reachable from e.

7. WAVESET OPERATIONS

The idea of a wave of nodes (or edges) was explained in 5.3 above. It may on occasion be necessary to retrieve the members of several different waves from some common initial node. Achieving this by making repeated calls to the EPI nth-wave operations would be possible, but inefficient, as computing an nth wave also involves computing the first, second, .. (n-1)th waves. If two separate calls are made to compute, say, waves 4 and 5, it follows that waves 1 to 4 will each be computed twice.

There are four EPI waveset operations. The idea is the same in each case: a unitary operation is provided to generate a user-specified set of waves, the aim being to avoid redundant computation of the kind described. Wavesets may be specified in two different ways, and in each case, there is one operation generating waves of nodes, and another for waves of edges.

A set of wave numbers may given either explicitly, as an array of wave numbers arranged in ascending order, or else as an arithmetic progression. In the latter case, the user provides an initial value i and a step s, specifying the waveset

\[ \{ i, i+s, i+2s, i+3s, .. i+ns, .. \} \]

A set given as a progression is in general (i.e., when s≠0) infinite. The stored graph is, however, finite, and so the result set is therefore finite, too.

Each node (or arc) in the result set may optionally be tagged with the number of the first wave it appeared in. One parameter of each waveset operation is a flag indicating whether results are to be tagged in this way or not.

The figure shows a graph with the first few waves of arcs from node d marked - an arc marked with number k appears in the kth wave.

```
2, 5, ..  3, 6, ..
  |         |
   a ---- b -- c
   |  \     |
   1, 4, ..
  d ---- e
  |  \    |
  1, 4, ..
  f ---- g ---- h
  |  \    |
  2, 5, ..
  e ---- i
```

13
Suppose this graph is stored with the ID g_id. Suppose, also, that waves is an array containing the numbers 1, 3 and 4 in that order. Then the following function call:

    ifs_waveset_edges_1 (g_id, node_b, 1, 3, waves, &result)

will collect the first, third and fourth waves of edges from node b, tagged with their wave numbers, in a single new class whose ID is returned in result. The new class will therefore contain the tuples:

    < d, b, 1 >,
    < d, e, 1 >,
    < a, c, 3 >,
    < a, d, 3 >

and

    < h, g, 3 >.

Each arc of wave 4 appears in a previous wave (the first), and because of this there are no arcs in the result set tagged with number 4.

The function ifs_waveset_edges_1() takes five arguments: a graph ID, a node in the graph, a flag (value 1 or 0) indicating whether results are to be tagged with wave numbers or not, the size of the array of wave numbers, the array itself, and an address at which to place the ID of the result set.

The function ifs_waveset_edges_2() is a version of ifs_waveset_edges_1() in which the waveset is given as an arithmetic progression, as described on the previous page. It takes six arguments: a graph ID, a node in the graph, the wave-numbering flag, an initial value and step for the progression, and an address at which to place the value of the result set.

For example, if g_id is as before, the contents of every even wave from node b can be computed by making the call:

    ifs_waveset_edges_2 (g_id, node_b, 0, 2, 2, &result)

or similarly, wave 5 and every third wave after it via the call:

    ifs_waveset_edges_2 (g_id, node_b, 0, 5, 3, &result).

Corresponding to the two edge waveset functions, the graph EPI provides two node waveset functions: ifs_waveset_nodes_1(), for which the waveset is given explicitly in the form of an array, and ifs_waveset_nodes_2(), for which it is given in the form of a progression. The waveset function nodes_1() takes the same arguments as edges_1(), and nodes_2() takes the same arguments as edges_2().
5. Acknowledgements

It is a pleasure to acknowledge the contribution of other members of the IFS team at Essex. The work described in this report has been supported by SERC grants GR/F/61028 and GR/G/30867.

6. References


