FAST RECURSIVE DATA PROCESSING IN GRAPHS USING REDUCTION

J.H. ter Bekke and J.A. Bakker
Faculty of Information Technology and Systems
Delft University of Technology
{J.H.terBekke, J.A.Bakker}@its.tudelft.nl

Abstract

This paper presents an algorithm for recursive data processing in directed graphs. The proposed algorithm applies graph reduction in order to determine both starting points and a correct ordering of recursive operations, provided the directed graph is a-cyclic. Therefore it is essential that the algorithm is also able to detect cycles efficiently. The algorithm arose from the implementation of recursive, semantic query specifications and is implemented in a DBMS prototype. Experiments confirmed that the theoretically estimated time complexity is $O(dN)$, where N is the number of arcs and d is the depth of the graph ($d \leq N$). The worst-case performance is $O(N^2)$, also for cycle detection.

Keywords: graph algorithm, longest path, recursion, query language, expressive power, semantic modeling.

1. Introduction

The main reason for the development of a generic algorithm for ordering arcs and nodes during recursive query processing was the objective that software should be able to determine starting points for data processing in directed a-cyclic graphs such that end-users do not have to specify these starting points. This complies with one of the main objectives of database systems based on the classical three-layer architecture to enhance that end-users can specify queries in a declarative (content-based) way; they should mainly specify *what* and not *how* the problem should be solved.

This approach resulted already in a new query implementation in a semantic DBMS [9] with an interpreter able to translate non-procedurally recursive queries into simple, well-ordered recursive operations. Such a layered approach, separating *what* and *how*, is impossible if recursive operations are specified directly in a programming language: then the choice of starting points is an inevitable part of the problem, which might be difficult for end-users. For example, the algorithms of Baily [1] assume given starting points. And so do problem specifications as those presented by Garey and Johnson [5].

In general, considering earlier approaches, ordering algorithms consist of two phases, in which successor lists play an important role [2, 4]. In the first phase, tuples from input are converted into successor lists in main memory. In the second phase the successor lists are expanded and written to disk. The large number of tuples created during this last phase results in time-consuming duplicate elimination and cycle detection, which might form an obstruction for processing complex graphs [6].

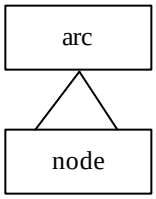
Contrary to these earlier approaches we use another solution; we consider only existing arcs of the graph and there is no need to expand it with successor lists. In this new algorithm only safe graph reduction takes place during the ordering of arcs, which makes the algorithm very efficient and creates only a minor dependency on main memory or disk. It makes the algorithm also suitable for large applications, even on small computers. Another advantage, in particular for casual environments as the Internet, is its reliability because graph reduction also enables us to detect cycles in a graph before starting recursive operations. This approach therefore shifts the investigation of the correctness of the input graph data towards the DBMS: software is able to detect cycles and to give an appropriate error message to the end-user, who needs not to worry about starting points and cycles.

This paper continues with an introduction to semantic modeling in section 2 and discusses some examples of recursive queries in section 3. Section 4 presents the ordering and cycle detection algorithm.

2. Semantic abstractions

This section contains an overview of the semantic data modeling abstractions needed for graph applications [8]. The concept of type is fundamental. Types are represented by rectangles in diagrams. Aggregation is defined as the collection of a certain number of types into a unit, which can be regarded as a new type. A type occurring in an aggregation is called an attribute of the new composite type. It is important to note the analogy with the mathematical set concept: attributes are considered as 'elements' of a type. Aggregation allows view independence (object relativity): we can discuss the obtained type (possibly also acting as a property of another type) without referring to its attributes. By applying this principle repeatedly, a hierarchy of types can be set up. An example of a hierarchy depicting two relationships between two types is given in figure 1. Normally only composite types are visualized in an abstraction hierarchy. If a line connects two facing

rectangle sides and the aggregate type (according to its definition) is placed above its attributes, this indicates aggregation. In our example database we consider the types 'description' and 'length' as base types. A type is completely defined by a list of its attributes, so we could apply the following type definitions to node and arc relationships shown in figure 1. Here we can consider, depending on the context, 'node' as a type or as an attribute in 'arc *its* from_node' or in 'arc *its* to_node'. Although these attributes suggest that the direction of arcs is from 'from_node' to 'to_node', this data model does not prescribe this interpretation. The model also allows for another interpretation, in which arcs have the reverse direction. In section 3 we show that the recursive *cascade* statement determines the direction of arcs and the order of recursive processing.



type node = description.
type arc = from_node, to_node, length.

Figure 1. Abstraction hierarchy for directed graphs

The corresponding semantic database contains two simple tables, partly shown in figure 2.

node	description
A	loc1
B	loc2
C	loc3
F	loc5
..	....

arc	from_node	to_node	length
1	A	B	l1
2	A	B	l2
3	A	C	l3
4	A	F	l4
..	..	..	....

Figure 2. Example database

3. Query specification

The final result of an ordering of nodes in a directed graph, which is a prerequisite for recursive query processing, is determined by the position of the arcs in the graph. Our primary goal is therefore to re-order the collection of arcs, not the collection of nodes. This ordering is used in the processing of the following examples of the recursive update statement *cascade*; we could compare it to keeping score in a game. Recursive applications can be found in critical-path problems related to project planning [9] or product databases [10]. Here we give an example of a longest path calculation, related to the graph in figure 3.

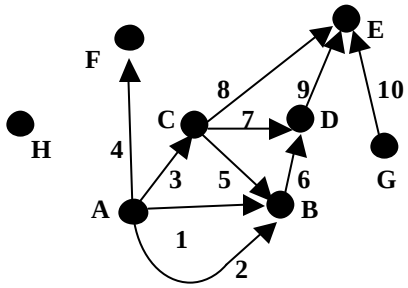


Figure 3. Example of a directed graph

Many different paths can be followed in such a directed graph and at least one of them is the longest. Using the Xplain query language the first part of the solution is to calculate for each node the longest path from a starting node to that node, whereas the second part is to calculate for each node the longest path from that node to a last node. First, we explain the first part of the solution, specified as follows:

extend node *with* first_path = 0.
/* Initialize all nodes, now ordering is irrelevant. */

cascade node *its* first_path =
max arc *its* length + from_node *its* first_path
per to_node.

After executing this cascading update operation each node has a certain value for the first path starting in some node not pointed to by any arc; more than one starting point is possible.

Using a reduction scheme, the value of the first path of the nodes A, G and H remains 0 because there are no arcs pointing to them. They could act as a starting point. After the first reduction step a temporal value of 'node *its* first_path' can be calculated for the destination nodes B, C, E, and F using the lengths of the removed arcs 1, 2, 3, 4 and 10. For the nodes C and F this value is the definitive value because they have only one in-coming arc.

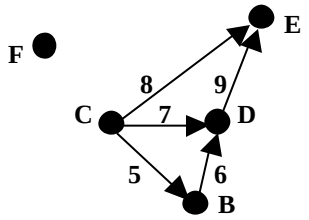
After the second reduction step the value of the first path can be updated for the nodes B, D and E. After the third reduction step the first path of the nodes D and E can be updated. After the fourth reduction the first path of node E can be updated. Finally, after the fifth step, the recursion stops because the remaining graph is empty.

After this short description, we describe the recursive processing in more detail. A first step is the removal of

starting nodes: nodes without any in-coming arc. We can consider these nodes as 'not referenced' by any incoming arc. This operation is specified in the language C, using a copy of the graph data (instances of 'node' and 'arc' including their attribute values). For reasons of comprehensibility we could specify these operations in Xplain terms:

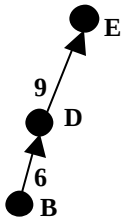
extend node *with* in_degree = *count* arc *per* to_node.
delete arc *where* from_node *its* in_degree = 0.
delete node *where* in_degree = 0.

Consequently the arcs 1, 2, 3, 4 and 10 and the nodes A, G and H are removed firstly, which results in the graph in figure 4. In a similar way the remaining graph can be reduced. The final result is a correctly ordered sequence of groups, each containing a subset of removed arcs: [1, 2, 3, 4, 10], [5, 7, 8], [6] and [9]. After this ordering, the data associated with the arcs of a group can be processed group after group, using 'arc *its* length' and 'arc *its* from_node *its* first_path'. In this way the value of instances of 'node *its* first_path' is updated in a correct order.



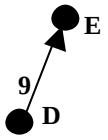
Removed arcs: [1, 2, 3, 4, 10]

Figure 4. Directed graph after the first reduction



Removed arcs: [1, 2, 3, 4, 10], [5, 7, 8]

Figure 5. Directed graph after the second reduction



Removed arcs: [1, 2, 3, 4, 10], [5, 7, 8], [6]

Figure 6. Directed graph after the third reduction

Removed arcs: [1, 2, 3, 4, 10], [5, 7, 8], [6], [9]

Figure 7. Directed graph after the fourth reduction

After the fifth reduction (removal of node E) the result is an empty graph.

Now we discuss the second part of the problem: how to calculate for each node the longest path between that node and some finish node (a node without any out-going arc):

extend node *with* last_path = 0.
cascade node *its* last_path =
    max arc *its* length + to_node *its* last_path
    per from_node.

Now the graph reduction has to start in finish nodes (here E, F and H). Arcs having a direction reverse to that in figure 3 could depict this reversed processing of data in a graph. Further reduction steps are similar to the first part of the solution. After this second cascading update we can calculate the longest path using the Xplain language:

value longest_path = *max* node *its* first_path + last_path.

All the arcs, including their nodes, on that longest path can be shown, sorted by increasing value for 'arc *its* from_node *its* first_path':

extend node *with* relevant =
    (first_path + last_path = longest_path).
extend arc *with* relevant =
    (from_node *its* relevant *and* to_node *its* relevant).
get arc *its* from_node, to_node, from_node *its* first_path
    where relevant *per* from_node *its* first_path.

If the graph contains more than one longest path, then the counting of start and finish arcs specified by the following two retrievals will reveal this situation, even if all these longest paths have both the same start and the same finish:

get count arc *where* relevant
    and from_node *its* first_path = 0.
get count arc *where* relevant
    and from_node *its* first_path = longest_path.

However, due to the nature of the problem, it is not possible to specify a declarative solution presenting each longest path and its correctly ordered sequence of constituting arcs.

From the semantic query specifications above follows that users do not specify start and finish nodes: all nodes

may act as a starting point. The DBMS software determines the ordering of reduction steps from the cascade statement.

4. Ordering algorithm

The result of a recursive query is primarily determined by the position of arcs: the connections determine the topological ordering of nodes. Our primary goal is therefore to re-order the collection of arcs, not the collection of nodes. Only this ordering is used in the processing of the cascade statement; it is comparable to score keeping in a game. The general form of the cascade statement is::

cascade <subtype> *its* <cascade attribute> =
<function> <maintype> *its* <expression>
per <grouping attribute>.

The following constraints regarding this statement must be satisfied:

- <expression> must contain the <cascade attribute>, this can be determined during the parsing process of the query statement. The reference of <cascade attribute> in <expression> (for example: 'from_node') must differ from the reference in the <grouping attribute> (for example: 'to_node'). If this condition is not satisfied the statement should be interpreted as a normal update statement without prescribed ordering.
- <grouping attribute> (for example 'to_node') must be identical to <subtype> (for example 'node'), possibly with a role added;
- It is evident that all usual constraints hold, for example: types, attributes and operations must comply and all specified attributes and types must exist in the underlying data model.
- It is only necessary to create a list of arcs such that an arc pointing to a node may only be followed by arcs starting in that node. Arcs are pointing to the node specified in the <grouping attribute>. For the calculation of 'node *its* first_path' it is 'to_node' and for 'node *its* last_path' it is 'from_node'. The desired ordering is therefore determined during the query parsing process, by Lex and Yacc.

The algorithm for implementing a cascade query statement therefore consists of two steps of which the first is essential for the cascade statement:

1. Determine the order in which the arcs can be processed. Note that only the cascade statement determines this ordering. The parameters for this ordering are determined during the parsing process of the query statement. This ordering is generally not needed in other query statements, such as the extend statement.
2. Process all arcs in the cascade according to the ordering determined in step 1. Contrary to that, other statements use a system-defined ordering; then there is no need to order the execution of the statement. An example is the initialization of all instances of 'node *its* first_path' to zero using the extend statement.

The parameters for the required ordering are determined by the DBMS during the parsing process of the query statement. The global structure of the ordering algorithm is straightforward. Including the detection of cycles, this algorithm is as follows:

```
init_structures ();
do {/* determine ordering */
        init_reference_counts ();
        update_reference_counts ();
        reduce_graph ();
}
until (no_reduction);

if (number_of_arcs > 0) {/* remaining arcs*/
    do {/* error handling */
        init_reverse_reference_counts ();
        update_reverse_reference_counts ();
        reduce_graph ();
    }
    until (no_reduction);
    print_arcs_in_cycles ();
}
```

The functions used fort the ordering algorithm:

- init_reference_counts ()
  The collection of nodes is determined by the contents of the database. The semantic model requires referential integrity: it does not allow any arc pointing from/to a non-existent node. This function initializes for each node all reference counts (number of arcs pointing to a node) to zero.
- update_reference_counts ()
  The collection of arcs determines the number of references to a finish node. This function requires a scan through the collection of arcs. For N arcs this scan has a time complexity $O(N)$. In this way the nodes not acting as a finish are determined. These nodes are the starting points and determine where in the graph reduction must start.
- reduce_graph ()
  Scan the collection of arcs. If the starting node of an arc is not referenced (not pointed to) by any other arc, put the arc on the ordering list for further processing in the semantic cascade statement and reduce the number of relevant arcs with one (remove the arc from the graph). The arc remains in the graph (a collection initially containing all arcs) if its start node is also the finish node of an (other) arc.

In [3] is proven that this reduction process results in an empty collection of arcs or in one or more cycles. The graph contains a cycle if the first reduction steps end with a non-empty collection of relevant arcs. In that case the reverse reduction is carried out until no further reduction is possible. At this point the arcs that contribute to cycles are found and can be reported to the user. The progress of the algorithm for a graph with cycles is illustrated by figure 8.

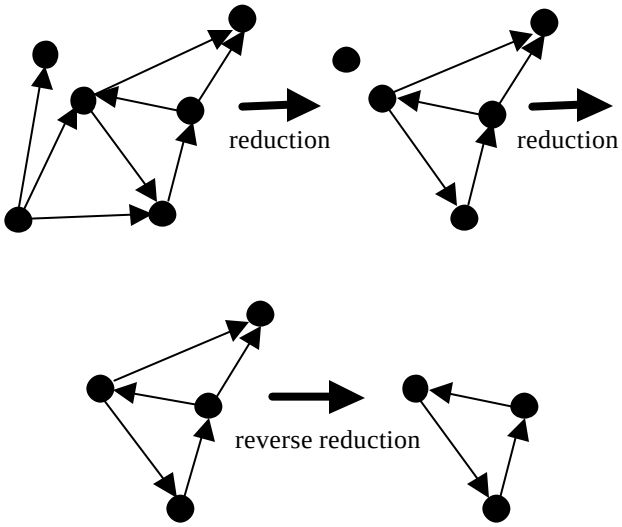


Figure 8. Cycle detection

The time complexity of the ordering algorithm can be determined as follows: define the depth d of a graph as equal to the maximum number of arcs of any simple path and N as the number of arcs. The number of reduction steps is proportional to d and normally each iteration step will reduce the number of relevant arcs with N/d (the worst case is a reduction of only one arc). The time complexity is therefore $O(dN)$. The worst-case performance is $O(N^2)$, also for cycle detection. The time complexity of the ordered recursive calculations is $O(N)$. These estimates are confirmed by measurements [10].

Conclusion

An efficient algorithm for recursive data processing in directed graphs has been presented. The technique of graph reduction instead of expansion is reliable because it can also detect cycles. It does not affect the existing database contents and is based on semantic query processing. Time complexity of recursive queries can be reduced to $O(dN)$, which means that the query execution time becomes predictable. This is especially important for open environments as the Internet, where systems cannot be protected by authorization tables and where querying by unknown users may not lead to denial of service.

References

[1] D.A. Bailey, Java structures: Data structures in Java for the principled programmer, MacGraw-Hill, Boston, (1999).

[2] F. Bancilhon and R. Ramakrishnan, An amateur's introduction to recursive query processing strategies, Proceedings 1986 ACM SIGMOD International Conference on Management of Data, Washington D.C. (1986), 16-52.

[3] J. Bang-Jensen and G. Gutin, Digraphs: Theory, Algorithms and Applications, Springer-Verlag, London, (2001).

[4] S. Dar, R. Ramakrishnan, A performance study of transitive closure algorithms, ACM SIGMOD Record, Vol. 23, No. 2 (June 1994), 454-465.

[5] M.R. Garey and D.S. Johnson, Computers and intractability: A guide to the theory of NP-completeness, W.H. Freeman, New York (1979).

[6] A. Karayiannis and G. Loizou, Cycle detection in critical path networks, Information Processing Letters, Vol. 7, No. 1 (January 1978), 15-19.

[7] J.C. Molluzzo, A first course in discrete mathematics, Wadsforth, Belmont CA, (1986).

[8] J.H. ter Bekke, Semantic data modeling, Prentice Hall, Hemel Hempstead (1992).

[9] J.H. ter Bekke and J.A. Bakker, Content-driven specifications for recursive project planning applications, Proceedings International Conference on Applied Informatics (AI 2002), Innsbruck (2002), 448-452.

[10] J.H. ter Bekke and J.A. Bakker, Recursive queries in product databases, Proceedings Fifth International Conference on Flexible Query Answering Systems (FQAS 2002), Lecture Notes in Artificial Intelligence Vol. 2522, Springer-Verlag, Berlin (2002), 44-55.