Proceedings of the IASTED International Conference
DATABASES AND APPLICATIONS
February 17-19, 2004, Innsbruck, Austria

A QUERY LANGUAGE SOLUTION FOR SHORTEST PATH PROBLEMS IN CYCLIC GEOMETRIES

J.A. Bakker and J.H. ter Bekke
Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
E-mail: {J.A.Bakker, J.H.terBekke}@ewi.tudelft.nl

ABSTRACT

This paper presents a recursive query language solution for the shortest path problem based on the transformation of a cyclic geometric graph into an acyclic time graph, i.e. an acyclic Petri Net specifying the possible transitions (turns) between successive rides on different roads. Using this transformation, the shortest path problem is transformed into a fastest tour problem. The time complexity of the proposed solution is $O(N^5)$, where N is the number of towns in the underlying geometry. The underlying algorithm guarantees termination.

KEY WORDS
Petri Net, recursive query, semantic model, shortest path, time complexity

1. Introduction

We previously described the application of the recursive *cascade* update operation of the Xplain language [6, 7] to several data structures corresponding with a weighted directed acyclic graph [1]. Examples using non-recursive data structures are project planning [9] and product planning [10]. We have also demonstrated the usability of this approach to problems related to recursive (hierarchical) data structures as the family tree [12, 13].

The algorithm [11] interpreting the *cascade* command applies graph reduction as a preparation to a well ordered serial processing. As an example we describe the reduction of the graph in figure 1 showing data based on the following data model:

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

We identify nodes by a capital letter and arcs by a small letter. The purpose is to determine the shortest route from S to F. First, we determine the distance 'sdist' of each node to the starting node S:

value inf = *total* arc *its* length.
extend node *with* sdist = inf. /* initialization */
update node "S" *its* sdist = 0.

Then we apply the recursive cascade update operation in order to calculate the correct distances to S. The result is that in a number of steps the minimal distance of each node to S is assigned to the destination of an arc:

cascade node *its* sdist =
min arc *its* length + from_node *its* sdist
per to_node.

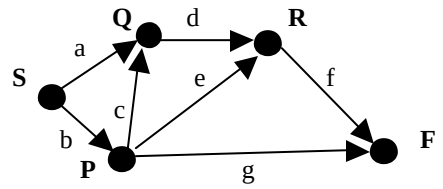


Figure 1. Example of an acyclic graph

The order of the steps is determined by graph reduction. This reduction is always the same, irrespective the set function applied. First the algorithm determines which nodes do not have any incoming arc; here it is S. The data representing the arcs starting in S (including the length of arcs) are placed in a list with groups of removed arcs. The first group is [a, b]. Figure 2 shows the result of the first reduction step.

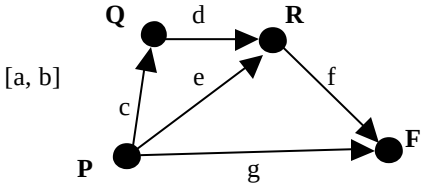


Figure 2. Result of the first reduction step

In a similar way the second reduction step is executed. Now the list is extended with the group [c, e, g]: figure 3.

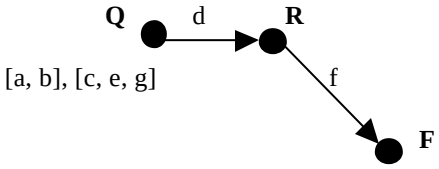


Figure 3. Result of the second reduction step

The result of the third step is shown in figure 4.

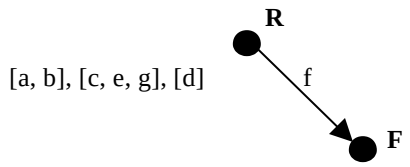


Figure 4. Result of the third reduction step

The result of the last reduction step is an empty graph and a list of four groups: [a, b], [c, e, g], [d] and [f]. After this ordering, the data of the arcs in the first group are processed: for each of these arcs the minimum of 'arc *its* length + arc *its* from_node *its* sdist' is assigned to the destination: 'arc *its* to_node' and this node gets a new value for 'node *its* sdist'. In this way, group by group of arcs, the value of 'node *its* sdist' is recalculated for the destination nodes of the arcs in each group.

In a similar way graph reduction and data processing is executed in a reverse order (starting in F) in order to calculate all instances of 'node *its* fdist', the shortest distance of a node to F.

extend node *with* fdist = inf.
update node "F" *its* fdist = 0.
cascade node *its* fdist =
min arc *its* length + to_node *its* fdist
per next_node.

Then for each node the sum of the distances to S and F, 'node *its* sdist + node *its* fdist', is calculated; the minimal value of sum is calculated and the nodes having this minimum value are the nodes on the shortest route from S to F: S, P and F.

extend node *with* totaldist = sdist + fdist.
value minimum = *min* node *its* totaldist.
get node *its* name *where* totaldist = minimum.

A weakness inherent in graph reduction is that it cannot be applied to cyclic graphs; for further details we refer to [11]. However, we have recently demonstrated that a cyclic geometric graph for flights can be transformed into an acyclic time graph for flight connections [3], which is the basis for finding the fastest series of flights between two airports. A first idea was to apply the following semantic data model, which allows us to specify the connections between roads.

type town = name.
type road = from_town, to_town, distance.
type connection = from_road, to_road.

As an example, figure 5 shows an example of a geometric graph in which the nodes A, B, C, D and E are representing towns and the arcs are representing single direction roads.

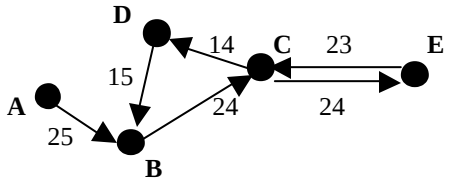


Figure 5. Example of a road network with a cycle

However, this data model still allows for graphs with cycles of connections between roads as shown in figure 6, where XY indicates a road from X to Y. This graph shows six roads (nodes) and eight possible connections (arcs) between roads.

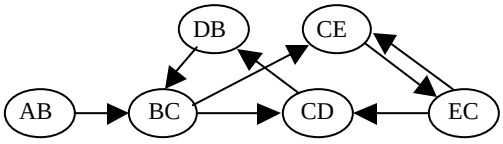
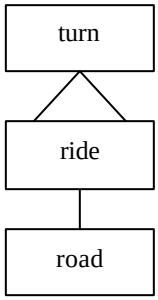


Figure 6. Roads (nodes) and connections (arcs)

Section 2 describes a more usable model supporting the construction of an acyclic time graph.

2. An appropriate data model

The data models in section 1 ignore the time dimension of the shortest path problem. A better solution is to introduce the notions of rides and turns as modeled in figure 7: a ride has both geometric dimensions and a time dimension (time level); a turn indicates which rides may follow after a ride. This model should enable us to deal with the correct time ordering of rides.



type turn (i7) = previous_ride (a3), next_ride (a3).
type ride (a3) = road (a2), time_level (i3).
type road (a2) = from_town (a2), to_town (a2),
distance(i4).

assert turn *its* correctness (*true*) =
(next_ride *its* time_level =
previous_ride *its* time_level + 1
and previous_ride *its* road <> next_ride *its* road
and previous_ride *its* road *its* to_town =
next_ride *its* road *its* from_town).

Figure 7. Data definitions for rides and turns

Types have a value domain; for example 'i7' denotes an integer with 7 decimal digits and 'a2' denotes an alpha-numeric string of 2 characters. Data on turns must be correct, which we express by an assertion specifying the calculation of a derivable Boolean property 'turn *its* correctness' that must be true for all turns. Other included restrictions must be applied by software deriving the instances of 'ride' and 'turn' that must be added to the geometric road data in order to create a database suitable for a solution of path problems based on graph reduction.

The data definitions in figure 7 enable us to present the required data as a directed acyclic graph: each turn (arc) always connects two rides (nodes) having successive time levels. We illustrate this by transforming the cyclic geo-metric graph of figure 5 into the acyclic time graph of figure 8 showing the possible transitions (turns) between successive rides. Therefore this graph can be considered as an acyclic Petri Net [5].

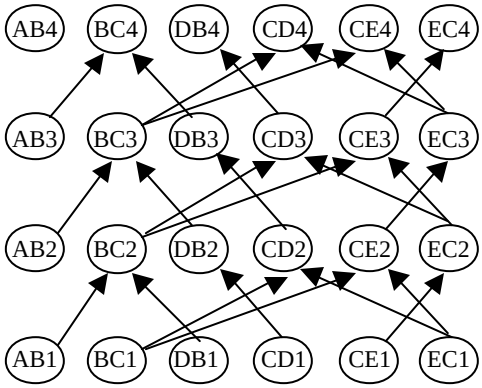


Figure 8. Tours as rides (nodes) and turns (arcs)

A shortest tour from A to D can be found by the three successive rides AB1, BC2 and CD3 (three nodes) and requires two turns (two arcs). Another suitable series of rides is: (AB2, BC3, CD4). If we want to get only the first series, then we have to select the chain starting at time level 1. The number of rides (ride: between adjacent towns) needed for a shortest trip between two towns is at most: ***N-1***, if ***N*** towns are represented in a geometric database. This worst case also determines the required number of time levels (depth of the time graph) that we have to register: ***N-1***.

We suppose that each town has single-direction roads to a fraction ***f*** (***f < 1***) of the other ***(N-1)*** other towns. Then the total number of roads is: ***fN(N-1)*** and the total required number of rides is $fN(N-1)^2$. Further, each ride can be followed by ***f(N-1)*** other rides, so $f^2N(N-1)^3$ possible turns (arcs) must be registered. The time complexity of the graph reduction algorithm is ***O(dP)***, where ***d*** is the depth of the graph and ***P*** is the number of arcs. Therefore the time complexity of reducing the time graph is $O(N^5)$. Graph reduction produces a well-ordered list of arcs. Processing this list has a time complexity $O(P) = O(N^4)$. In order to demonstrate the usability of the proposed approach, section 3 specifies a query language solution for the shortest path between two towns.

3. Shortest path calculation

First, we specify the interactive part of the query, enabling a user to choose start and finish town:

value start = *input*(a2) "Enter the start town: ".
value finish = *input*(a2) "Enter the finish town: ".

The following specifications derive for each ride the distance (sdistance) to the start town:

value inf = *total* road *its* distance.
/* a shortest path is not longer than 'inf' */
extend ride *with* distance = road *its* distance.
/* simplifies the following specifications */

The rides starting in the start town have a time level of 1:

extend ride *with* sdistance = inf. /* initialization */
update ride *its* sdistance = 0
where road *its* from_town = start
and time_level = 1.

Then we apply the recursive cascade update operation:

cascade ride *its* sdistance =
min turn *its* previous_ride *its* sdistance +
previous_ride *its* distance
per next_ride.

In a similar way we determine for each ride its distance to the finish town, but now we do not know a priori the time level of the ride ending in that town. This time level depends on the selected towns and the structure of the road network.

extend ride *with* fdistance = inf. /* initialization */
update ride *its* fdistance = 0
where road *its* to_town = finish.
cascade ride *its* fdistance =
min turn *its* next_ride *its* fdistance +
next_ride *its* distance
per previous_ride.

Now we determine for each ride the total chain length of the tour from start to finish it belongs to:

extend ride *with* chainlength =
sdistance + distance + fdistance.
value minimum = *min* ride *its* chainlength.

It is possible that all rides found by the previous operations have a total chain length larger than 'inf'. This occurs if we try to determine the shortest walk between two towns without any connection. For example, there is not any route from D to A (figure 2, figure 5). Therefore we have to deal with unusable rides with 'fdistance = inf' or 'sdistance = inf':

extend ride *with* suitable = (chainlength = minimum
and fdistance < inf *and* sdistance < inf).

```
extend turn with suitable =
      (previous_ride its suitable
       and next_ride its suitable).
value suitablenumber = count ride where suitable.
value comment = "If suitablenumber = 0, then there is no
                 connection from start to finish town: "
value suitablenumber. newline.
value comment. newline.                    /* print commands */
```

It is also possible that we find more than one shortest trip between start and finish. Although not present in figure 2, it is possible that A and D also have other connections in addition to the roads AB, BC and CD. Consequently, there can be many suitable rides (on different roads) starting at the same time in the same town. Then a presentation of suitable rides ordered by time level is not suitable. Analogous to our solution for fastest air connections [4], a better presentation of results can be obtained by deriving the position of suitable rides. If more than one suitable ride has a same position then the user knows that there are alternative routes.

```
value maxposition = count flight.          /* initialization */
extend ride with position = maxposition.
update ride its position = 1
      where suitable
      and road its from_town = start.      /* the first ride */
cascade ride its position =
      min turn its previous_ride its position + 1
                where suitable
      per next_ride.
```

The following retrieval presents the desired results, ordered by the position of suitable rides. The result of this retrieval is empty if there is not any path from start to finish town:

```
get ride its position, road, sdistance, distance, fdistance,
          chainlength where suitable
      per position.
```

Section 4 shows the obtained results.

4. Results

Using the data in the tables 1-3, which are related to figure 2, we examined whether the proposed query produced correct results.

road	from_ town	to_ town	distance
AB	A	B	25
BC	B	C	24
CE	C	E	24
CD	C	D	14
DB	D	B	15
EC	E	C	23

Table 1. Road data

ride	road	time_level
AB1	AB	1
AB2	AB	2
AB3	AB	3
BC1	BC	1
BC2	BC	2
BC3	BC	3
CE1	CE	1
CE2	CE	2
CE3	CE	3
CD1	CD	1
CD2	CD	2
CD3	CD	3
DB1	DB	1
DB2	DB	2
DB3	DB	3
EC1	EC	1
EC2	EC	2
ED3	EC	3

Table2. Some possible rides

turn	previous_ ride	next_ ride
1	AB1	BC2
2	AB2	BC3
5	BC1	CD2
6	BC2	CD3
9	BC1	CE2
10	BC2	CE3
13	CD1	DB2
14	CD2	DB3
17	CE1	EC2
18	CE2	EC3
21	EC1	CE2
22	EC2	CE3
25	EC1	CD2
26	EC2	CD3
29	DB1	BC2
30	DB2	BC3

Table 3. Some possible turns

Some results for different pairs of start and finish towns are presented in table 4.

trip	suit num	ride	pos	s dist	dist	f dist	chain length
AA	0						
AB	1	AB1	1	0	25	0	25
AC	2	AB1	1	0	25	24	49
		BC2	2	25	24	0	49
AD	3	AB1	1	0	25	38	63
		BC2	2	25	24	14	63
		CD3	3	49	14	0	63
AE	3	AB1	1	0	25	48	73
		BC2	2	25	24	24	73
		CE3	3	49	24	0	73
BA	0						
BB	3	BC1	1	0	24	29	53
		CD2	2	24	14	15	53
		DB3	3	38	15	0	53

Table 4. Results of shortest path calculations

All rides constituting a desired trip satisfy the following rule: sdistance + distance + fdistance = chainlength.

5. Discussion

Because of the required construction of an acyclic time graph, our solution for the shortest path in a cyclic network is not the fastest one, but faster solutions in $O(N^2)$ [1] have a procedural character; they require to specify (nested) control statements, which is a dis-advantage for end users. Another advantage of the semantic solution is that termination is guaranteed, which is very important in open systems that cannot be protected by authorization tables [4].

Contrary to the semantic approach, the application of SQL in open environments can lead to denial of service because the deliberate or accidental specification of Cartesian products or complex joins can produce more data than present in the database [4]. In SQL3 recursion can be specified by recursive views [14]. Using SQL3 it is possible to specify endless processing that only stops if a run-time error occurs. An example is the calculation of a series of positive integers; it starts with only one tuple with value '0' in the table 'integer', informally specified as: RELATION integer (number);

Recursive calculation:

```
WITH RECURSIVE calculated (number) AS
     (SELECT number
      FROM integer)
     UNION
     (SELECT (number + 1)
      FROM calculated
      WHERE number IN (SELECT MAX (number)
                       FROM calculated));
SELECT number
FROM calculated;
```

We produced the test results using a notebook with an Intel Celeron ™ 1.7 GHz processor and the processing time (30 rides and 32 turns) was 0.04 seconds. In the case of another cyclic geometry with 20 roads, converted into an acyclic time graph with 184 rides and 480 turns, the calculation of the shortest route between two towns toke 0.07 seconds. Future research, using large, more realistic data sets, has to reveal whether the measured processing time remains acceptable and complies with the theoretical time complexity.

References

[1] J.M. Aldous and R.J. Wilson, Graphs and Applications, an Introductory Approach, Springer-Verlag, London (2000).

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

[3] J.A. Bakker and J.H. ter Bekke, A Query Language Solution for Fastest Flight Connections, *Proceedings International Conference on Database Applications (DBA 2004)*, Innsbruck, Austria 2004, ACTA Press, Calgary (2004).

[4] Bert Bakker and Johan ter Bekke, Fool Proof Query Access to Search Engines, *Proceedings Third International Conference on Information Integration and Web-based Applications & Services (IIWAS 2001)*, Linz, Austria, W. Winiwarter, S. Bressan and I. K. Ibrahim (Eds.), Österreichisches Computer Gesellschaft (2001), pp. 389-394.

[5] A.A.S. Danthine, Protocol Representation with Finite-State Models, IEEE Transactions on Communication, Vol. COM-20 (1980), pp. 632-643.

[6] F. Rolland, The essence of databases, Prentice Hall, Hemel Hempstead (1998).

[7] J.H. ter Bekke, Semantic Data Modeling, Prentice Hall, Hemel Hempstead (1992).

[8] J.H. ter Bekke, Advantages of a compact semantic meta model, *Proceedings 2nd IEEE Metadata Conference*, Silver Spring, USA (1997), http://www.computer.org/conferen/proceed/meta97/papers/jterbekke/jterbekke.html.

[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, Austria, M.H. Hamza (Ed.), ACTA Press, Calgary (2002), pp. 448-452.

[10] J.H. ter Bekke and J.A. Bakker, Recursive queries in product databases, Flexible Query Answering Systems, *Proceedings 5th International Conference (FQAS 2002)*, Copenhagen, Denmark, October 27-29, 2002, Lecture Notes in Computer Science (subseries LNAI) Volume 2522, T. Andreasen, A. Motro, H. Christiansen and H. Legind Larsen (Eds.), Springer-Verlag, Berlin (2002), pp. 44-55.

[11] J.H. ter Bekke and J.A. Bakker, Fast Recursive Data Processing in Graphs Using Reduction, *Proceedings International Conference on Applied Informatics (AI 2003)*, Innsbruck, Austria, M.H. Hamza (Ed.), ACTA Press, Calgary (2003), pp. 490-494.

[12] J.H ter Bekke and J.A. Bakker, Modeling and Querying Recursive Data Structures I: Introduction, *Proceedings International. Conference on Artificial Intelligence and Soft Computing (ASC 2003)*, Banff, Canada, H. Leung (Ed.), ACTA Press, Calgary (2003), pp. 278-282.

[13] J.H ter Bekke and J.A. Bakker, Modeling and Querying Recursive Data Structures II: A Semantic Approach, *Proceedings International Conference on Artificial Intelligence and Soft Computing (ASC 2003)*, Banff, Canada, H. Leung (Ed.), ACTA Press, Calgary (2003), pp. 283-289.

[14] J.D. Ullman and J. Widom, A First Course in Database Systems, Prentice Hall, Hemel Hempstead (1997).