AGGREGATIVE CLOSURE:
AN EXTENSION OF TRANSITIVE CLOSURE

Isabel F. Cruz Theodore S. Norvell
Department of Computer Science
University of Toronto
Toronto, Canada
M5S 1A4

Abstract: In common database applications, it is often the case that the user wants not only to know which pairs of entities are in the transitive closure of a relation, but also the aggregation of information along and among the paths connecting such pairs. We define a relational operator embodying this idea and call it aggregative closure. We also define two other relational operators: aggregative composition and aggregative union. These definitions, together with the theory of semirings, provide the framework for the presentation of new algorithms for the computation of aggregative closure which result from generalizing existing efficient algorithms.

1 Introduction

In recent years considerable research effort has been put into finding ways for expressing and efficiently evaluating recursive queries (see for example [GaMi78, BaRa86]). A transitive closure query is generally recognized to be the simplest example of a recursive query. Lately, this type of query has itself received an important share of the attention paid to recursive queries in general [Ioan86, VaBo86, AgJa87, Lu87].

A relation where two of the attributes are defined in the same domain can be graphically visualized as a finite directed labeled graph such that the labels on the vertices are constants in this domain, and the labels of edges are tuples of the values for the remaining attributes. We call this graph the database graph.

In practical database query systems, it is not only the case that the user wants to know if two entities defined in the same domain are related through transitive closure, but often how they are related by aggregating information from the paths in the database graph relating these two entities. We will be calling this the aggregative closure.

Queries which require the power of aggregative closure are needed in path applications (“What is the shortest path between each pair of cities?”), in bill of materials applications (“How many subparts are required to assemble part a?”), computer network routing (“What is the most reliable path between two nodes?”), or “What is the maximum capacity path between two nodes?”), and critical paths (“What is the path of longest duration between two tasks in a PERT network?”).

Proposals such as PROBE [DBGHM85, RHDM86], the α-operator extension to relations algebra [Agra87], and $G^+$ [Cruz87, CMW88] allow the expressing of the above queries. This paper is concerned with the efficient evaluation of queries requiring the expressive power of the aggregative closure operator that we will be defining. Other papers with similar intent include [DBGHM85, ADJ88, IoRa88].

The outline of this paper is as follows. We present the concepts of closed semirings, absorptive semirings and graphs, and maximizing semirings, as the formal context for defining aggregative closure. We then define three aggregating relational operators: aggregative union, aggregative composition, and aggregative closure.

These definitions provide a uniform framework for the third section of the paper. In this section, we extend existing algorithms to arrive at new algorithms to compute the aggregative closure of a relation. These algorithms can be classified into three families. We present extensions to the naive, semi-naive and logarithmic algorithms, which we will call path-growing algorithms since they iterate until all simple paths cease to grow. We present direct algorithms which are based on those of Kleene [AHU74] and Warren [Warr75]. And we present a path losing algorithm which takes advantage of the properties of maximizing semirings to reduce the number of paths which must be constructed.

We end this section by discussing under what conditions transitive closure algorithms can be extended to compute aggregative closure.

We end with some conclusions and suggestions for further research.

2 Aggregative Closure

In this section we consider an extended transitive closure operator which allows the aggregation of values on paths in the database graph. We first define closed semirings. We then introduce two binary operators for relations: the aggregative composition operator and the aggregative union operator. Finally we give the definition of aggregative closure.
2.1 Closed Semirings

Our operators aggregate the labels on the edges of the database graph. Because our definitions and algorithms apply to diverse applications, we give them in terms of abstract operators. Along paths, an attribute is aggregated according to a specified abstract operator, the path or product operator (written \( \odot \)), and over each set of paths connecting the same pair of vertices according to a specified abstract operator, the set or addition operator (written \( \oplus \)).

**Definition:** A closed semiring [Mehl84] is a system \((P, \oplus, \otimes)\), where \(P\) is a set, and \(\oplus\) and \(\otimes\) are operations on \(P\) with the following properties.

(i) \((P, \oplus)\) is a commutative monoid;\(^1\) its identity element is written \(0\) and is also called the zero element of the semiring. Although we will usually write \(\oplus\) as a binary operator, it must be defined over any countable sequence.\(^2\) The sum of the empty sequence is \(0\).

(ii) \((P, \otimes)\) is a monoid; its identity element is written \(1\) and is also called the unit element of the semiring. Although we will usually write \(\otimes\) as a binary operator, when it is applied to a sequence, we write it as a prefix operator. The product of the empty sequence is \(1\).

(iii) \(\otimes\) distributes over \(\oplus\).

This definition is similar to that of path algebras [Carr79] and of closed semirings [AHU74] except that the set operator is not required to be idempotent (\(\oplus\) is idempotent iff for all \(a\) in \(P\), \(a \oplus a = a\)).

Table 1 shows a number of useful and interesting closed semirings.

There are other problems, such as finding the set of all simple paths\(^3\) between two vertices, which also fit into the framework of closed semirings [Carr79].

There are also interesting triples that do not fit the axioms of a closed semiring. For example: in the connecting flight times with minor modifications to the graph, we are trying to find all pairs of arrival and departure times such that we can leave a city at the departure time and arrive at the other at the arrival time by taking a sequence of flights with exact connections. A more realistic product operator would allow some stop over time.

\[ x \otimes y = \{ (d, a) : \exists d', d' < d' \land (d, d') \in x \land (d', a) \in y \} \]

It is interesting to note that although this path operator has no identity element, we can still compute the set of connecting flight times with minor modifications to the algorithms we will be presenting.

---

1 A monoid is closed, associative and has an identity element.

2 Because \(\oplus\) is commutative, the order of the sequence is immaterial, and we will often leave it unspecified.

3 A simple path is a path in which all vertices are distinct except that the endpoints may be the same.

---

**Definition:** The closure of an element \(x\) is written \(x^*\) and is defined by

\[ x^* = \bigoplus_{i \geq 0} x^i \]

where

\[ x^i = x \otimes x \otimes \cdots \otimes x \]

\(i\) times

That is,

\[ x^* = 1 \odot x \odot (x \odot x) \odot (x \odot x \odot x) \cdots \]

For instance, \(o^* = 1\). We assume that the closure of an element can be calculated in finite time. This allows us to deal with the infinity of paths between two vertices which may arise in a cyclic graph.

There are two properties of semirings and graphs which can aid in finding efficient algorithms for their aggregative closure. These are the properties of being maximizing and of being absorptive.

**Definition:** A semiring is maximizing iff (i) the relation defined by

\[ x \leq y \iff y = x \oplus y \]

is a total order (i.e. it is total, reflexive, antisymmetric and transitive) and (ii) we have the following monotonicity

\[ x \otimes y \leq z \land x \otimes y \leq y \]

We read \(x \leq y\) as "\(x\) is as maximal as \(y\)." For any \(x\) in \(P\), we have \(0 \leq x \leq 1\). We will also be using \(x < y\) meaning \(y\) is more maximal than \(x\).

An example of a maximizing semiring comes from the shortest path application, where \(x \leq y\) means \(x \geq y\). The critical path application, on the other hand, is not maximizing because it lacks monotonicity.

**Definition:** A semiring is absorptive iff for any \(x\) in \(P\),

\[ x \oplus 1 \leq 1 \]

A graph is said to be absorptive iff for any directed cycle \(\rho\), the product \(x\) of the labels of edges in the cycle \(\rho\) is such that \(x \oplus 1 = 1\) [Carr79].

From this definition, we can see that any maximizing semiring is also absorptive. Note that all acyclic graphs are absorptive, and that when the semiring in question is absorptive, all graphs will be absorptive. Algorithms which work for absorptive semirings will also work for nonabsorptive semirings provided the graph they are applied to is itself absorptive. This is the case for the bill of materials application, where the semiring is nonabsorptive, but the database graphs are acyclic and therefore absorptive.

When the semiring is maximizing, we can ignore any paths other than the maximal one between any pair of
vertices. When we are dealing with absorptive semirings, we can ignore any paths containing cycles because they will be absorbed into the simple path which underlies them.

2.2 Aggregative Relational Operators

Next we define the aggregating relational operators. To simplify the presentation, we will consider only ternary relations. The extension to n-ary relations with $n > 3$ is straightforward.

Definition: The aggregative union operator $\cup$, unites two relations $R$ and $S$. $T = R \cup S$ contains those tuples $(i,j,d)$ such that

$$
\exists a. (i,j,a) \in R \cup S
$$

and $d$ is defined by

$$
d = \bigoplus_{(l,j,a) \in R} a \ominus b \bigoplus_{(l,j,b) \in S}
$$

Note that the matrix corresponding to $T = R \cup S$ is $A_R \oplus A_S$, the matrix sum of the corresponding matrices.

Example 2 When $R$ is binary, the ordinary reflexive transitive closure is built as follows.

$$
R^+ = \pi_{\mathbf{1}, \mathbf{1}}(R \times \{(1)\})
$$

where $\{(1)\}$ is the constant relation with a single tuple of value one, and $\mathbf{1}$ denotes the attribute in position $i$.

Table 1: Closed Semirings

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$x \otimes y$</th>
<th>$o$</th>
<th>$i$</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1</td>
<td>$x \lor y$</td>
<td>$x \land y$</td>
<td>0</td>
<td>1</td>
<td>Transitive closure</td>
</tr>
<tr>
<td>$R^+ \cup {0}$</td>
<td>max($x, y$)</td>
<td>min($x, y$)</td>
<td>0</td>
<td>$\infty$</td>
<td>Maximum capacity path</td>
</tr>
<tr>
<td>$R \cup {0, \infty}$</td>
<td>$x + y$</td>
<td>$x \ast y$</td>
<td>$-\infty$</td>
<td>0</td>
<td>Critical path</td>
</tr>
<tr>
<td>${z \in R \mid 0 \leq z \leq 1}$</td>
<td>max($x, y$)</td>
<td>$x \ast y$</td>
<td>0</td>
<td>1</td>
<td>Most reliable path</td>
</tr>
<tr>
<td>$R^+ \cup {\infty}$</td>
<td>min($x, y$)</td>
<td>$x + y$</td>
<td>$\infty$</td>
<td>0</td>
<td>Shortest path</td>
</tr>
<tr>
<td>$2^{P_{\text{max}} \times P_{\text{max}}}$</td>
<td>$x \lor y$</td>
<td>$(\forall \mathbf{d} \in P_{\text{max}}) (\exists \mathbf{s} ((\mathbf{d}, \mathbf{s}) \in x) \land (\mathbf{s}, \mathbf{a}) \in y)$</td>
<td>0</td>
<td>$\mathbf{I}$</td>
<td>Bill of materials</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Connecting flights</td>
</tr>
</tbody>
</table>
3 Evaluation of Aggregative Closure Queries

We now give some algorithms for computing the aggregative closure between all pairs of vertices in the database graph. Most of these are generalizations of algorithms developed for transitive closure.

We classify the algorithms into three groups: the path growing algorithms which work when the semiring or database graph is absorptive, the direct algorithms which work for all closed semirings, and the path losing algorithms which work when the semiring is maximizing.

3.1 Path Growing Algorithms

Perhaps the simplest method of computing the aggregative closure consists of repeatedly extending the known paths by adding a single edge to each of them until no new paths are found. This is called the naive algorithm and is the simplest example of a family of algorithms known as the path growing or iterative algorithms. They work only when the database graph is absorptive because they consider only paths no longer than the longest simple path in the graph.

This means that for each family of paths with the same underlying simple path, at least one, but possibly more representatives will be found by the algorithms. Because of absorptivity, the sum of the paths we actually find is equal to the sum of the entire family.

The correctness of these algorithms depends on the fact that when the semiring is absorptive

$$\{\oplus, \otimes\} R_0 = \bigcup_{0 \leq i < n} R_0^i$$

where \(n\) is greater than the length of the longest simple path in the graph of \(R_0\), and \(R_0^i\) means \(i\) copies of \(R_0\) composed with each other (\(R_0^i = I\) where \(I\) is the identity relation consisting of one tuple \((a, a, i)\) for each vertex \(a\) in the graph).

The naive algorithm (see for instance [BaRa86]) can be rewritten for the aggregative closure of a relation \(R_0\) as:

$$R := I$$
$$\Delta R := I$$
$$\text{do}$$
$$\Delta R := \Delta R \otimes [\otimes, \otimes] R_0$$
$$R := R \cup \Delta R$$
$$\text{while } R \text{ changes}$$

The time complexity of this algorithm is \(O(n^3 \times l)\) where \(n\) is the number of vertices in the database graph and \(l\) is the length of the longest simple path in the database graph.\(^4\)

As shown in [ADJ88], the semi-naive algorithm for aggregative closure can be written as follows.

$$\delta R := R_0$$
$$R := I$$
$$\text{do}$$
$$\Delta R := R \otimes [\otimes, \otimes] \delta R$$
$$R := R \cup \Delta R$$
$$\delta R := \delta R \otimes [\otimes, \otimes] \delta R$$
$$\text{while } R \text{ changes}$$

At the end of the \(i\)th iteration, the following hold

$$\Delta R = \bigcup_{0 \leq j < c^{\ell i}} R_0^{2^{\ell i - 1} + j}$$
$$R = \bigcup_{0 \leq k < c^i} R_0^k$$
$$\delta R = R_0^i$$

Comparing with the semi-naive evaluation method, we see that the number of iterations is reduced from the length of the longest simple path to the logarithm (base two) of its length.

---

\(^4\)Unless otherwise stated, complexity results will be in terms of the number of \(\oplus, \otimes, \text{ and } *\) operations required.
3.2 Direct Algorithms

The termination of algorithms for computing transitive closure such as the ones due to Warshall [War76] and Warren [War75] does not depend on the length of the longest simple path on the database graph. This is why they are called direct algorithms. Warshall's algorithm is a special case of a more general algorithm essentially due to Kleene (see for example [AHU74]), as is Floyd's shortest path algorithm [Floy62]. We will present both Kleene's algorithm and an analogous generalization of Warren's algorithm due to Lu [Luo82] to compute the aggregative closure. All of these algorithms compute the aggregative closure for any closed semiring.

There are other direct algorithms (such as those in [AgJa87]) which can also be extended.

Kleene's algorithm

Kleene's algorithm transforms a matrix $A$ from being the adjacency matrix for a relation $R_a$ to being the adjacency matrix of its aggregative closure.

\[
\text{for } k := 1 \text{ to } n \text{ do} \\
\quad \text{for } i := 1 \text{ to } n \text{ do} \\
\quad \quad \text{if } A(i, k) \neq 0 \text{ then} \\
\quad \quad \text{for } j := 1 \text{ to } n \text{ do} \\
\quad \quad \quad A(i, j) := A(i, j) \oplus (A(i, k) \odot A(k, k)^* \odot A(k, j)) \\
\text{for } i := 1 \text{ to } n \text{ do} \\
\quad A(i, i) := A(i, i) \oplus \epsilon
\]

As is noted in [AHU74], Warshall's algorithm and Floyd's algorithm are special cases: we obtain Warshall's algorithm if we replace $\odot$ by $\lor$ and $\oplus$ by $\land$; Floyd's algorithm is obtained by replacing $\odot$ by $+$ and $\oplus$ by $\land$.

After the $k^{th}$ iteration of the outermost loop, the algorithm will have considered all paths with intermediary vertices (vertices which occur between the endpoints) less than or equal to $k$. Furthermore each path is considered only once. The proof of the above algorithm is by induction on the number of iterations of this loop. The induction hypothesis is as follows. After $k$ iterations, for all pairs $i, j$ we have that: if $Q$ is the set of all paths of the form $i, t_1, \ldots, t_g, j$ such that $t_8 \leq k$ for $1 \leq h \leq g$, then

\[
A(i, j) = \bigoplus_{q \in Q} (\odot \text{lab}(q))
\]

where lab$(q)$ is the sequence of labels on path $q$.

The time complexity of Kleene's algorithm is $O(n^3)$. If the transitive closure is sparse (the number of edges in it is $O(n)$), the if-statement will be true no more than one time in $O(n)$, so the complexity of the whole algorithm is only $O(n^2)$.

Warren's algorithm

Warren's algorithm [War75] is similar to Warshall's algorithm, but executes faster in a paging environment.

The improvement of Warren's algorithm over Warshall's is essentially due to the fact that matrices are stored in row major form, that is, the column index varies fastest.

When the matrix is sparse, we want to access all the elements $A(i, k)$ with the smallest possible number of page faults. In Warshall's algorithm these elements of the matrix are accessed in column major order, since the index $k$ changes fastest, unlike in Warren's algorithm where access to the matrix is in row major order. For sparse matrices, this improves the locality of the needed data and hence reduces page faults. As the matrix gets denser, the efficiency (in terms of page faults required) by both algorithms is approximately the same. We note that if the data were stored in column major order, then the locality of the execution of the inner loop in Kleene's algorithm would deteriorate since the inner loop accesses the matrix in row major order.

Just as Kleene's algorithm can be seen as a generalization of Warshall's algorithm, we generalize Warren's algorithm to any closed semiring. The time complexity is $O(n^3)$ and $O(n^2)$ when the transitive closure is sparse, as with Kleene's algorithm. We call this the aggregating Warren's algorithm.

\[
\text{pass } 1: \\
\quad \text{for } i := 2 \text{ to } n \text{ do} \\
\quad \quad \text{for } k := 1 \text{ to } i - 1 \text{ do} \\
\quad \quad \quad \text{if } A(i, k) \neq 0 \text{ then} \\
\quad \quad \quad \text{for } j := 1 \text{ to } n \text{ do} \\
\quad \quad \quad \quad A(i, j) := A(i, j) \oplus (A(i, k) \odot A(k, k)^* \odot A(k, j)) \\
\text{pass } 2: \\
\quad \text{for } i := 1 \text{ to } n - 1 \text{ do} \\
\quad \quad \text{for } k := i \text{ to } n \text{ do} \\
\quad \quad \quad \text{if } A(i, k) \neq 0 \text{ then} \\
\quad \quad \quad \text{for } j := 1 \text{ to } n \text{ do} \\
\quad \quad \quad \quad A(i, j) := A(i, j) \oplus (A(i, k) \odot A(k, k)^* \odot A(k, j)) \\
\text{for } i := 1 \text{ to } n \text{ do} \\
\quad A(i, i) := A(i, i) \oplus \epsilon
\]

We sketch a proof that the algorithm correctly calculates the aggregative closure of the relation originally stored in $A$. We say that a path from $a$ to $b$ has been found by the algorithm when the product of the labels along the path has been added to $A(a, b)$.

Lemma 1 If $q = (a, t_1, t_2, \ldots, t_m, b)$ has been found by the end of pass 1, then $t_i < a$, for $1 \leq i \leq m$.

PROOF: This is proved by induction on the length of paths. $\blacksquare$

Lemma 2 For each path $q = (a, t_1, t_2, \ldots, t_m)$, there is a unique index $n, 1 \leq n \leq m$, such that $(a, t_1, t_2, \ldots, t_n)$
has been found by the end of pass 1 and either \( n = m \) or \( a \leq t_n \).

**Proof:** The proof that such an \( n \) exists is given by Warren [War76]. That \( t \) is unique follows from Lemma 1.

**Lemma 3** For each path \( q = (a, t_1, t_2, \ldots, t_m, b) \), there is a unique way to split \( q \) into subpaths

\[
q_0 = (a, \ldots, t_{r_1}),\quad q_1 = (t_{r_1}, \ldots, t_{r_2}) \cdots q_p = (t_{r_p}, \ldots, b)
\]

such that (1) \( a \leq t_{r_1} \leq t_{r_2} \leq \cdots \leq t_{r_p} \) and (2) each \( q_i \) has been found by the end of the first pass.

**Proof:** Each \( q_i \) is constructed by using Lemma 2 to find each \( t_{r_{i+1}} \) (or \( b \), in the case of \( q_p \)). The uniqueness of this construction can be proven by induction on the number of \( q \)'s using Lemma 1.

**Theorem 1** For each path \( q \), \( q \) is found once and only once.

**Proof:** Lemma 2 is used to decompose \( q \). If \( p = 0 \), \( q \) has been found at the end of pass 1. Otherwise, pass 2 concatenates the \( q_i \) together to form \( q \). That \( q \) is found only once comes from the uniqueness of the decomposition in Lemma 3.

### Relational Warren's algorithm

The previous algorithms apply to data stored in matrices. Lu et al. [LMRS7] recognize the fact that the large domains in database relations make the adjacency matrix of such a relation not practicable. The representation of a database graph as ternary relation is much more compact for sparse matrices. The attributes \( A \) and \( B \) represent the row and column indices and a third \( L \) represents the data in a location of a matrix. If an element of the matrix is \( o \) then that element is represented by no tuple.

Lu et al. convert Warren's algorithm to work with tuples. The aggregating Warren's algorithm can be converted as follows.

\[
T := R_0
\]

**sort** \( T \) primarily on \( A \) and secondarily on \( B \)

**for** \( t \in T \) in sorted order

**do**

\[
\text{if } t.A > t.B \text{ then }\]

\[
\text{if } \exists L'. (t.B,t.B,L') \in T \text{ then } L := L'
\]

\[
\text{else } L := o
\]

\[
\text{replace } \sigma_{A = L}(T) \text{ in } T
\]

\[
\text{by } \sigma_{A = L}(T) \cup \{ (a,\emptyset) \} \cdot \{ (t.B,t.B,L^*) \} \cdot \{ (a,\emptyset) \} \sigma_{A = L(B)}(T)
\]

**for** \( t \in T \) in sorted order

**do**

\[
\text{if } t.A \leq t.B \text{ then }\]

\[
\text{if } \exists L'. (t.B,t.B,L') \in T \text{ then } L := L'
\]

\[
\text{else } L := o
\]

\[
\text{replace } \sigma_{A = L}(T) \text{ in } T
\]

\[
\text{by } \sigma_{A = L}(T) \cup \{ (a,\emptyset) \} \cdot \{ (t.B,t.B,L^*) \} \cdot \{ (a,\emptyset) \} \sigma_{A = L(B)}(T)
\]

\[
T := T \cup I
\]

In understanding the correspondence between this and Warren's algorithm, it is useful to note that \( t.A, t.B \) and \( L \) correspond to \( i, k \) and \( A(k,k) \) respectively. The "replace" statement is equivalent to the innermost loop in Warren's algorithm, and can be implemented as a merge of two sorted streams. Since \( T \) is sorted first on attribute \( A \) and second on attribute \( B \), the two input streams of data \( \sigma_{A = L}(T) \) and \( \sigma_{A = L}(T) \) will be contiguous sequences of tuples, as will the new value for \( \sigma_{A = L}(T) \). Therefore, the disk accesses will be very local. The increase in locality in Warren's algorithm has a very big pay-off in the relational version.

The complexity is \( O(n^3) \) in the worst case, but \( O(n) \) when the transitive closure is sparse.

All the direct algorithms work for any closed semiring.

If the semiring is absorptive or the graph is known to be absorptive, then the \( A(k,k)* \) factor (or the equivalent factor in the relational Warren's algorithm) may be omitted from the computation, as it is always \( i \).

### 3.3 Path Losing Algorithms

If the semiring is maximizing, then we need not consider even all simple paths. When a path between two vertices is less maximal than the most maximal yet found between those two vertices, it can be safely ignored. This is the principle upon which Dijkstra's shortest path algorithm works (see for instance [AHU74]).

We next present an algorithm for computing the aggregative closure of a relation which works for semirings which are maximizing. It is similar to Dijkstra's algorithm, but differs in that it computes the aggregative closure for all pairs of vertices, not just those with a particular source.

As with the aggregating Warren's algorithm, it converts the adjacency matrix \( A \) of the original relation to one for the aggregative closure.

\[
\text{for } i := 1 \text{ to } n \text{ do }
\]

\[
A(i,i) := i
\]

\[
W := \{(i,j) : A(i,j) \neq o\}
\]

\[
F := \emptyset
\]

**while** \( W \neq \emptyset \) **do**

\[
\text{choose } (i,j) \in W \text{ to maximize } A(i,j)
\]

\[
W := W \setminus \{(i,j)\}
\]

\[
F := F \cup \{(i,j)\}
\]

**for** \( k \text{ such that } (j,k) \in F \land (i,k) \notin F \** do**

\[
A(i,k) := A(i,k) \oplus A(i,j) \oplus A(j,k)
\]

\[
W := W \cup \{(i,k)\}
\]

**for** \( k \text{ such that } (k,i) \in F \land (k,j) \notin F \** do**

\[
A(k,j) := A(k,j) \oplus A(k,i) \oplus A(i,j)
\]

\[
W := W \cup \{(k,j)\}
\]

Each pair \((a,b)\) is in one of three states. They move from one to the other in sequence. In the first state, the pair is in neither \( F \) nor \( W \); this means we have not yet determined it to be in the transitive closure. In the second
state, the pair is in $W$; this means that it is in the transitive
closure. In the third state, the pair is in $F$; this means that
the pair is in the transitive closure and that $A(a, b)$ is the
product along a maximal path from $a$ to $b$.

The algorithm is $O(n^3)$ and $O(n)$ for relations with
a sparse transitive closure. In practice we would expect
the algorithm to be fast because it only considers edges
catenations of maximal paths when searching for a
maximal path.

We sketch a proof of the correctness of the algorithm.

Lemma 4 The following is true prior to the test at the
top of the while loop: For any $(a, b) \in W$ which maximizes
$A(a, b)$, if $q = (a', \ldots, b')$ is any path such that $A(a, b) <$
@lab(q), then $(a', b') \in F$.

Proof: We prove by induction on the length of $q$ that
$(a', b')$ was previously placed on $F$. If $q$ is an edge or an
empty path, then $(a', b')$ is added to $W$ before entering
the main loop; at that time @lab(q) is equal to $A(a', b')$.
$A(a', b')$ won't subsequently become less maximal and so

$A(a, b) < \leq @lab(q) \leq A(a', b')$

Since $(a, b) \in W$ maximizes $A(a, b)$, $(a', b') \notin W$; the only
place it could have gone is $F$. If $q$ is not an edge, it is the
catenation of two subpaths $(a', \ldots, c')$ and $(c', \ldots, b')$ each
with fewer edges. By the monotonicity property of max-
imizing semirings, both subpaths are at least as maximal
as $q$, so by induction both $(a', c')$ and $(c', b')$ have been
found. When the second to be found was placed on $F$,
$(a', b')$ is placed on $W$ (unless it is already on $F$). At that
point $@lab(q) = A(a', b')$ (because of the assignment in the
for loop). $A(a', b')$ won't have become less maximal, so we
now have

$A(a, b) < \leq @lab(q) \leq A(a', b')$

As above, this means that $(a', b') \notin W$ and hence $(a', b') \in F$.

Lemma 5 The following is true prior to the test at the
top of the while loop: For any $(a, b) \in W$ which maximizes
$A(a, b)$, if $q = (a, \ldots, b)$, then $@lab(q) \leq A(a, b)$.

Proof: We suppose to the contrary that $A(a, b) <$
@lab(q). By Lemma 4, $(a, b)$ is on $F$. It can not also be
on $W$ so this is a contradiction.

Theorem 2 After termination, for all pairs $(a, b)$, $A(a, b)$
is the product along the maximal path between $a$ and $b$, or
$\infty$ if there is no such path.

Proof: This follows directly from Lemma 5, the fact that
$A(a, b)$ is never changed once $(a, b) \in F$, and the fact that
$F$ is the transitive closure upon termination.

3.4 Extending Transitive Closure Algorithms

It is interesting to reflect on why some algorithms for tran-
sitive closure extend nicely to give algorithms for aggrega-
tive closure and why others do not.

Most transitive closure algorithms take advantage of
the fact that the Boolean semiring $((0, 1), \lor, \land)$ is absorp-
tive and hence find only simple paths. In some cases, e.g. the
direct algorithms, this is easily remedied. In others, e.g. the path
growing algorithms, it does not seem so easily repaired, and so they are restricted to absorptive semirings
or graphs.

The second property of transitive closure that may
be exploited is that once a single path between two ver-
tices $a$ and $b$ has been found, no other paths between them
are of interest. The algorithm in [O'NO'N73] takes advan-
tage of this; it is path losing in that it ignores certain
paths. These algorithms are unsuitable for aggregative clo-
sure where each path must be found once. However, in a
maximizing closed semiring the situation is similar to that
of transitive closure; once we have found a maximal path
between $a$ and $b$, no other path is of interest. Hence, there

4 Conclusions

We have defined the aggregative closure operator and
demonstrated its usefulness in a wide variety of appli-
cations. The concepts and definitions of closed semirings
and the aggregating relational operators provide a math-
ematical framework for the presentation of algorithms for
these applications. We have also presented new algorithms
intended for the computation of the aggregative closure, all
but the last of these being generalizations of existing algo-
rithms intended for transitive closure.

Our definition of aggregative closure is similar to that

\footnote{However, for sparse matrices, there are also $O(n)$ priority queue
operations which can be implemented on a RAM machine to take
$O(\log |W|)$ instructions each.}
of traversal recursion in [RHDM86]. Traversal recursion, however, is more general in that it does not require distributivity of the path operator over the set operator.

We see several directions for future research.

The algorithms we have presented compute the aggregative closure between all pairs of vertices in the database graph. Also of interest is computing the aggregative closure only for a subset of the vertex pairs, for example, all pairs with a particular source (and/or sink). We may also wish to find the sum over all paths between one set of vertices and another. All these classes of queries are considered in $G^+$ [CMW88, CrMe88] and are mentioned in [ADJ88].

Algorithms presented here and elsewhere with similar complexity should be compared by simulation to determine their relative merits. It should be seen to what extent the algorithms presented here can be generalized to algebras weaker than closed semirings. Algorithms based on the topology of the database graph are important, for instance algorithms based on strongly connected components are presented in [IoRa88] and [ADJ88]; algorithms based on other decompositions, for instance interval analysis, should also be investigated. We have considered two classes of closed semirings: absorptive semirings and maximizing semirings; other classes should also be identified and investigated.

Acknowledgements

We would like to thank Alberto Mendelzon for suggesting the topic and Mariano Consens for his insightful suggestions. We are also indebted to H. V. Jagadish for his useful comments, which, in particular, helped us to improve Section 3.1. The first author’s research was supported by an INVOTAN grant and a Government of Canada Award.

References


