Theoretical Computer Science 296 (2003) 473 – 485

Theoretical Computer Science

www.elsevier.com/locate/tcs

The dynamic complexity of transitive closure is in DynTC^0

William Hesse

Department of Computer Science, University of Massachusetts, Amherst, MA 01002, USA

Abstract

This paper presents a fully dynamic algorithm for maintaining the transitive closure of a binary relation. All updates and queries can be computed by constant depth threshold circuits of polynomial size (TC^0 circuits). This places dynamic transitive closure in the dynamic complexity class DynTC^0, and implies that transitive closure can be maintained in database systems that include first-order update queries and aggregation operators, using a database with size polynomial in the size of the relation.
© 2002 Elsevier Science B.V. All rights reserved.

1. Introduction

Many restricted versions of transitive closure are known to be dynamically maintainable using first-order updates. The transitive closure of symmetric relations and acyclic relations can be maintained in first order [6,12]. In this paper, we show that the transitive closure of any binary relation can be dynamically maintained using a polynomial-size data structure, with updates that are computed by constant depth threshold circuits. We show that updating the data structure upon adding or deleting a tuple is in the circuit complexity class TC^0, described in Section 4. This means there is a uniform family of constant depth threshold circuits, with size polynomial in the size of the graph, computing the new values of all the bits in the data structure.[1] If the answer to a query can be maintained in this way, we say that query is in the complexity class DynTC^0.

Queries computed by TC^0 circuits are exactly those queries describable by first-order logic plus counting quantifiers [2]. We use this fact to show that the transitive closure of a relation can be maintained by SQL queries, using auxiliary relations of size

E-mail address: whesse@cs.umass.edu (W. Hesse).

[1] All circuit complexity classes are assumed to be DLOGTIME-uniform unless otherwise stated. This is discussed in Section 4.

polynomial in the size of the input relation. The contents of these auxiliary relations are uniquely determined by the input relation, independently of the order of the updates to the input relation.

This paper is organized as follows. We discuss previous related work in Section 2. In Section 3 dynamic complexity classes and the dynamic complexity of transitive closure are defined. In Section 4 circuit complexity classes including TC^0 are described. Section 5 presents the dynamic algorithm for transitive closure. Sections 6 and 7 show that all operations in the dynamic algorithm are in the complexity class TC^0 and can be written as SQL update queries. By extending the SQL algorithm to allow changes to the relation's domain, and showing that the size of the auxiliary data remains polynomial in the size of the input relation, we construct a polynomially bounded SQLIES for transitive closure. The final section offers some conclusions and directions for further work.

2. Previous related work

Libkin and Wong have previously shown that the transitive closure of a relation can be maintained using SQL updates and an exponential amount of auxiliary information [11].[2] The dynamic complexity class $DynTC^0$ used in this paper is less powerful than the class SQLIES (SQL incremental evaluation systems) used in their paper. SQLIES captures all queries maintainable using SQL updates, which allow the creation of large numbers of new domain elements. $DynTC^0$ lacks the capability to introduce new constants; the domain of all relations in a $DynTC^0$ algorithm is the integers from 0 to $n-1$, with ordering and arithmetic operations. Since dynamic computations in $DynTC^0$ use a constant number of relations of constant arity, they use an amount of auxiliary data polynomially bounded by the size of this domain. General SQL computations, by introducing new tuples with large integer constants as keys, can potentially square the size of the auxiliary databases at each iteration, leading to exponential or doubly exponential growth of the amount of auxiliary data kept.

The pair (s,t) is in the transitive closure of a graph G iff there is a path from s to t in G. Previous work showed that reachability in undirected and acyclic graphs could be maintained by first-order updates.

Patnaik and Immerman introduced the complexity class DynFO containing those dynamic problems with first-order dynamic algorithms. In these, each operation is implemented as a first-order update to a relational data structure over the finite domain $\{0,\dots,n-1\}$ with ordering and arithmetic [12]. They showed that reachability in an undirected graph (the transitive closure of a symmetric relation) is in DynFO.

Dong and Su defined a first-order incremental evaluation system (FOIES) in [6]. They showed that reachability in an acyclic graph can be maintained by a FOIES. The universe of first-order formulas in a FOIES is the set of constants in the input database, and does not include ordering or arithmetic on these constants. A DynFO algorithm

[2] Their algorithm keeps information about a possibly exponential number of paths through the directed graph induced by the relation.

evaluates first-order formulas over a fixed domain $\{0,\dots,n-1\}$, and includes ordering and arithmetic on this domain. The input is a database over this domain [6]. Etessami showed how to create FOIES algorithms from some DynFO algorithms by dynamically maintaining an ordering and arithmetic relations on the input domain [7].

This paper shows that transitive closure is in the dynamic complexity class DynTC^0. It is unknown whether transitive closure in DynFO. However, Dong, Libkin, and Wong showed that the transitive closure of a relation cannot be maintained using first-order updates on a data structure with only unary auxiliary relations [5].

3. Dynamic graph reachability

Let $G=(V,E,s,t)$ be a directed graph with two distinguished vertices s and t. The decision problem REACH is the set of all such graphs containing a directed path from s to t. The transitive closure of the graph G is the set of edges $E^\star$ such that

$$(\forall s,t\in V)\quad (s,t)\in E^\star \;\Leftrightarrow\; (V,E,s,t)\in \text{REACH}.$$

When we refer to the transitive closure $R^\star$ of a binary relation R, we will mean the transitive closure of that binary relation regarded as the edges of a directed graph.

We dynamically maintain the transitive closure of a binary relation R over the domain $\{0,\dots,n-1\}$ using a dynamic algorithm for the problem REACH$(\{0,\dots,n-1\},R,s,t)$. A dynamic algorithm is a set of auxiliary data structures and a set of update algorithms. The update algorithms update the graph G and the auxiliary data when an edge is inserted into or deleted from the graph. There are also update procedures for changing s and t. The complexity of this dynamic algorithm is the maximum complexity of any of these update algorithms.

Our dynamic algorithm for REACH remembers the number of paths of length k between s and t, for $0\leqslant s,t,k<n$. This is the auxiliary data kept by the algorithm. These numbers are all less than n^n, and so can be stored using a polynomial number of bits.

The operations we allow on the input graph are to add a directed edge between two vertices and to delete a directed edge between two vertices. The update algorithms compute the new numbers of paths, which are polynomials in the previous numbers of paths. We will show that these polynomials can be computed by TC^0 circuits, and that queries REACH(G,s,t) can be answered easily using this data. In this way, we show that we can create a polynomial-size data structure with updates and queries computable by TC^0 circuits.

This places the problem REACH in the dynamic complexity class DynTC^0. In [12], the complexity class Dyn-$\mathscr{C}$ is defined for any static complexity class $\mathscr{C}$. A summary of this definition is that a static problem is in Dyn-$\mathscr{C}$ if there is a dynamic algorithm for that problem that maintains a polynomial amount of auxiliary data and updates the auxiliary data with a computation in complexity class $\mathscr{C}$. The operations allowed are changes of a single bit of the input. There must also be a computation in $\mathscr{C}$ that computes from the input and auxiliary data whether the current input is accepted by the static problem.

Remark 1. In specifying the operations for the dynamic version of REACH, we did not include operations to add or delete vertices. As this approach derives from finite model theory, we conceive of dynamic REACH as being a family of problems, parameterized by the number of graph vertices, n. We show in Section 8 that this algorithm can be modified to yield a SQLIES which allows addition and deletion of vertices (new domain elements) while keeping the size of the auxiliary relations polynomial in the size of the input relation.

4. The parallel complexity class uniform TC^0

In static complexity theory, many natural low-level complexity classes have been parallel complexity classes, containing those problems which can be solved by idealized massively parallel computers in polylogarithmic or constant time. One model for these computations is as circuits made up of Boolean gates. A circuit is an acyclic directed graph whose vertices are the gates of the circuit and whose edges are the connections between gates. There is an input gate for each bit of the input, and a single output gate. The parallel complexity classes AC^0 and TC^0 contain problems accepted by uniform families of circuits with polynomial size and constant depth.

The circuit complexity class (DLOGTIME-uniform) TC^0 contains all decision problems computed by a family of constant depth circuits containing AND, OR, and threshold gates, all with unbounded fan-in, as well as NOT gates. There is one circuit for each value of the input size, n, and there is a single constant and a single polynomial in n such that these circuits have a size (number of gates) bounded by that polynomial and depth bounded by that constant. In addition, the circuits must be described by a DLOGTIME Turing machine. There must be a Turing machine that decides, given two gate numbers, the types of these gates and whether the first is an input of the second. This DLOGTIME uniformity will be assumed throughout the paper. It lets us use the fact that $\mathrm{TC}^0 = \mathrm{FO(COUNT)}$, those problems described by first-order formulas using counting quantifiers [2].

The addition of threshold gates distinguishes these circuits from the circuit class AC^0, which contains only AND, OR, and NOT gates. A threshold gate with n inputs and threshold k accepts if k or more of its inputs are true. Thus the majority gate which is 1 iff more than half of its inputs are 1 is computed by a threshold gate with n inputs and threshold $\lfloor n/2 \rfloor + 1$. Conversely, by adding enough dummy inputs, set to the constants 0 or 1, a majority gate can simulate a threshold gate with any threshold.

The relations between the classes AC^0, TC^0, and the logspace Turing machine classes are currently known to include the following inclusions:

$$\mathrm{AC}^0 \subset \mathrm{TC}^0 \subseteq \mathrm{L} \subseteq \mathrm{NL}.$$

The importance of the classes AC^0 and TC^0 to this paper is that first-order queries can be decided by AC^0 circuits, and that TC^0 is the smallest important complexity class containing AC^0. It has been shown to strictly contain AC^0 because important problems including parity, majority, and integer multiplication have been shown to be computable by TC^0 circuits but not by AC^0 circuits.

The other two complexity classes mentioned above are not circuit classes, but Turing machine complexity classes. The classes L and NL denote deterministic and nondeterministic logspace computability by Turing machines. The class NL is relevant because the static version of REACH is a complete problem for this complexity class. This shows that the dynamic complexity of REACH is potentially significantly smaller than its static complexity.

Because DLOGTIME-uniform TC^0 is being used, the equivalent logical class FO (COUNT) must also include ordering and arithmetic on its domain $\{0,\dots,n-1\}$. We include the order relation and BIT in all our first-order logical classes, so we could call this class FO(COUNT,$\leqslant$,BIT) to be completely explicit. The BIT predicate $\mathrm{BIT}(i,j)$ is true iff bit i of the number j is 1. This is first-order equivalent to the addition and multiplication relations on this domain.

Since we are using FO(COUNT) for its equivalence with relational database languages with aggregate operators, the use of arithmetic operators is appropriate. Any query language with counting operators must have a domain of integers, and should have arithmetic on those integers. We are only using arithmetic on numbers between 0 and n, with $\log n$ bits. Even though we use numbers with n^4 bits in our algorithm, these long numbers are represented as relations, with a tuple for each bit.

We will use these relations between circuit complexity classes to show that the updates computed by circuits in our algorithm can be computed by first-order query languages with aggregate operators, and to help show that the steps in our algorithm can be computed by a TC^0 circuit. We will use a recent result that polynomial evaluation is in DLOGTIME-uniform TC^0 [8].

5. A dynamic algorithm for REACH

Our dynamic algorithm for REACH keeps track of the number of paths $p_{i,j}(k)$ of length k from vertex i to vertex j, for every $0\leqslant k,i,j<n$. The number $p_{i,j}(k)$ is always less than n^n, so it is representable as an $n\log n$-bit binary number.

On adding or deleting an edge, we calculate the new values of $p_{i,j}(k)$ by adding and multiplying polynomials, and raising them to powers. We derive our update formulas from the generating functions

$$f_{i,j}(x)=\sum_{k=0}^{\infty} p_{i,j}(k)x^k.$$

These are formal power series, with infinitely many terms. The algorithm only maintains a finite number of terms, and calculates a truncated version of this generating function

$$A_{i,j}(x)=\sum_{k=0}^{n-1} p_{i,j}(k)x^k.$$

Our input is a directed graph G on n vertices, which are numbered $0,\dots,n-1$. The graph G is represented by its adjacency matrix, an n by n array of bits $e_{i,j}$, where $e_{i,j}$ is 1 if there is a directed edge from i to j, 0 otherwise. The auxiliary information we will keep is the array of n^3 numbers $p_{i,j}(k)$, where $p_{i,j}(k)$ is the number of paths of length k from i to j in the graph. Note that $p_{i,i}(0)=1$, $p_{i,j}(1)=e_{i,j}$, and that the paths counted are not necessarily simple paths; they may include cycles. Since the number of paths of length k from i to j is bounded by n^k, $p_{i,j}(k)$ is a number with at most $k\log n$ bits. We will only consider paths of length $n-1$ or less. This is sufficient to decide our queries, since if two vertices are connected, they are connected by a path of length $n-1$ or less. Therefore, for $k<n$, $p_{i,j}(k)<n^n$.

5.1. *Updating the generating functions $f_{s,t}(x)$*

We will calculate $p'_{s,t}(k)$, the updated values for $p_{s,t}(k)$ using the generating functions $f_{s,t}(x)$. We first show how to compute the new values of the generating functions $f_{s,t}(x)$ upon adding an edge.

Lemma 2. *Let $f_{s,t}(x)$ and $f'_{s,t}(x)$ be the generating functions counting the paths in the graphs G (not containing edge (i,j)) and $G'=G\cup\{(i,j)\}$, for all s,t:*

$$f_{s,t}(x)=\sum_{k=0}^{\infty} p_{s,t}(k)x^k, \qquad f'_{s,t}(x)=\sum_{k=0}^{\infty} p'_{s,t}(k)x^k.$$

Then

$$f'_{s,t}(x)=f_{s,t}(x)+f_{s,i}(x)x\left(\sum_{k=0}^{\infty}(f_{j,i}(x)x)^k\right)f_{j,t}(x). \tag{1}$$

Proof. Each path from s to t in G' of length k using edge (i,j) once can be decomposed into three parts: a path from s to i in G of length m, the edge (i,j), and a path from j to t in G. Thus the number of paths from s to t in G' of length k using edge (i,j) once is

$$\sum_{m=0}^{k-1} p_{s,i}(m)p_{j,t}(k-m-1).$$

This is the coefficient of x^k in $f_{s,i}(x)xf_{j,t}(x)$. The factor x adds the length of the edge (i,j) to the exponent of x.

Similarly, a path from s to t that uses edge (i,j) twice can be decomposed into a path from s to i, edge (i,j), a path from j to i, edge (i,j), and a path from j to t. The number of such paths of length k is

$$\sum_{m=0}^{k-2}\sum_{n=0}^{k-m-2} p_{s,i}(m)p_{j,i}(n)p_{j,t}(k-m-n-2).$$

This is the coefficient of x^k in $f_{s,i}(x)xf_{j,i}(x)xf_{j,t}(x)$. Summing these expressions, and those for paths using edge (i,j) 3,4, and more times, we get the RHS of Eq. (1). □

We have similar formulas for the new generating functions resulting from the deletion of an edge from G.

Lemma 3. *Let $f_{s,t}(x)$ and $f'_{s,t}(x)$ be the generating functions counting the paths in the graphs G (containing (i,j)) and $G' = G \setminus \{(i,j)\}$, for all s,t, as above.*
Then

$$f'_{s,t}(x) = f_{s,t}(x) - f_{s,i}(x)x\left(\sum_{k=0}^{\infty}(-f_{j,i}(x)x)^k\right)f_{j,t}(x). \tag{2}$$

Proof. This proof is more difficult, since the counts $p_{i,j}(k)$ count both the paths using (i,j) and those not using (i,j). We will show that the sum

$$\sum_{k=0}^{\infty} f_{s,i}(x)x(-f_{j,i}(x)x)^k f_{j,t}(x)$$

counts each path using (i,j) exactly once, no matter how many times the path uses the edge.

A path from s to t using the edge (i,j) l times can be decomposed into $k+1$ paths by choosing k of the uses of the edge (i,j). This results in one path from s to i, $k-1$ paths from j to i, and a path from j to t, as well as k copies of the edge (i,j). The number of such decompositions is $\binom{l}{k}$. These decompositions are counted by the generating function

$$f_{s,i}(x)x(f_{j,i}(x)x)^{k-1}f_{j,t}(x).$$

Thus a path using edge (i,j) l times is counted $\binom{l}{k}$ times by this term. The lengths of the paths and the powers of x correspond as in the above proof.

Let $\chi(l)$ be the number of times a path using edge (i,j) l times is counted by the alternating sum of all these expressions

$$\sum_{k=1}^{\infty}(-1)^{k-1}f_{s,i}(x)x(f_{j,i}x)^{k-1}f_{j,t}(x).$$

Then

$$\chi(l) = \sum_{k=1}^{\infty}(-1)^{k-1}\binom{l}{k}.$$

This sum is the binomial expansion of $(1-1)^l$, excluding the first term, and with opposite signs. Since $(1-1)^l = 0$, this sum is equal to the first term, 1. Note that this is true for $l \geqslant 1$. If $l = 0$, all terms in the sum are 0, and the path is not counted at all. This is seen to be true, since all the generating functions only count paths using edge (i,j) at least once.

Since the above sum counts exactly once all paths using edge (i,j), the RHS of Eq. (2) counts all paths not using edge (i,j). □

Remark 4. The formulas for updating $f_{s,t}(x)$ upon deletion and addition of an edge can also be verified by seeing that these transformations are inverses of each other. By composing the polynomial updates for an insertion and a deletion, we verify that $f_{s,t}(x)$ remains unchanged when we insert and delete an edge.

5.2. *Updating the counts $p_{s,t}(k)$*

The generating functions used above are infinite mathematical objects, and we want to compute using finite mathematical objects, such as integers. To turn a generating function into an integer, we truncate it after n terms, and replace the variable x with a large integer $r = 2^{n^2}$. This defines the n^3-bit integers

$$a_{i,j} = \sum_{k=0}^{n-1} p_{i,j}(k) r^k.$$

The constant $r = 2^{n^2}$ is large enough so that the binary representation of $a_{i,j}$ is the concatenation of the binary representations of $p_{i,j}(k)$, padded to n^2 bits with zeros, for all k. The result of our update calculations will contain the representations of the new values of $p_{i,j}(k)$, similarly separated by zeros. We can guarantee this separation since the coefficients of all polynomials involved in the update computations are smaller than r.

We must derive formulas for updating $a_{i,j}$ from the formulas for updating the generating functions $f_{i,j}(x)$. We accomplish this by truncating the summations, and computing mod r^n.

Lemma 5. *Let $a_{s,t}(x)$ and $a'_{s,t}(x)$ be the integers counting the paths in the graphs G (not containing edge (i,j)) and $G \cup \{(i,j)\}$, for all s,t, according to the equations*

$$a_{i,j} = \sum_{k=0}^{n-1} p_{i,j}(k) r^k, \quad a'_{i,j} = \sum_{k=0}^{n-1} p'_{i,j}(k) r^k.$$

Then

$$a'_{s,t} \equiv a_{s,t} + a_{s,i} r \left(\sum_{k=0}^{n-2} (a_{j,i} r)^k \right) a_{j,t} \pmod{r^n}.$$

Proof. If we truncate the summation in Eq. (1) for generating functions after the $(n-1)$th term, the coefficients of the first n powers of x remain the same. This is because all later terms include a factor x^n. If we then truncate the generating functions on the RHS of Eq. (1) after the nth term, the coefficients of x^k, for $k < n$, still remain the same. If we truncate the generating function $f'_{s,t}(x)$ on the LHS after the term $p'_{s,t}(n-1)x^{n-1}$, we still have an equation that is correct for the powers of x less than x^n, and this is now an equation relating finite polynomials, not formal power series. Since the sides of the equation differ only on terms including x^n as a factor, we have a polynomial equivalence modulo x^n. By substituting r for x, we have the statement of the lemma. □

We use the fact that $a'_{s,t} < r^n$ to replace the equivalence by an equation

$$a'_{s,t} = a_{s,t} + a_{s,i} r \left(\sum_{k=0}^{n-2} (a_{j,i} r)^k \right) a_{j,t} \bmod r^n.$$

In this equation, we reduce the RHS to an integer in the range $[0, r^n)$. This is the actual computation our algorithm performs to update the quantities $a_{s,t}$.

In a similar fashion, we convert the formula for updating generating functions on deletion of an edge to a formula for updating $a_{s,t}$

$$a'_{s,t} = a_{s,t} - a_{s,i} r \left(\sum_{k=0}^{n-2} (-a_{j,i} r)^k \right) a_{j,t} \bmod r^n.$$

The final operation we need to be able to do is to query whether s and t are connected by a directed path in our graph. But there is a path from s to t if and only if the value $a_{s,t}$ is non-zero. This can easily be checked by an FO formula, and thus by a TC^0 circuit.

6. Computing the updates in TC^0

The updated values of the numbers $a_{i,j}$, and thus the updated numbers of paths $p_{i,j}(k)$ can be computed by TC^0 circuits. We show this by demonstrating how all intermediate values in the algorithm can be computed by TC^0 circuits. The circuits will be simple to construct, and they will be composed in simple ways to compute the final results from the inputs. The most complex circuit used will be a circuit raising an n^3 bit number to the power n, and we cite a result showing that such a circuit exists. It is not obvious that the composition of all these circuits can be computed by a DLOGTIME Turing machine. It is simpler to see that since the computation effected by each of these circuits can be described by a formula in FO(COUNT), the combined computation can be described by a combined formula in FO(COUNT), and therefore can be performed by a TC^0 circuit.

The formula for updating the $a_{s,t}$ upon inserting an edge is

$$a'_{s,t} = a_{s,t} + a_{s,i} r \left(\sum_{k=0}^{n-2} (a_{j,i} r)^k \right) a_{j,t} \bmod r^n.$$

We shall see that all of these operations upon numbers with n^3 and n^4 bits can be performed by TC^0 circuits.

The computational power of constant depth threshold circuits was investigated by Reif and Tate in [13]. They found that polynomials with size and degree bounded by $n^{O(1)}$ and with coefficients and variables bounded by $2^{n^{O(1)}}$ could be computed by polynomial-size constant depth threshold circuits [13, Corollary 3.4]. We cannot use the result of Reif and Tate about the evaluation of polynomials directly because they only state that there exist polynomial-time-uniform TC^0 circuits to evaluate them. A series of results by Chiu, Davida, Litow, Allender, Barrington, and Hesse shows that the circuits for division, powering, and iterated multiplication used to evaluate polynomials can be made DLOGTIME-uniform [1,3,4,8]. In particular, finding the product of $n^{O(1)}$ (polynomially many) numbers, each with $n^{O(1)}$ bits, can be done by a DLOGTIME-uniform TC^0 circuit.

Evaluating the above polynomial requires us to raise numbers of $O(n^3)$ bits to powers up to $n-2$, multiply the results by other numbers, add $n-1$ of the results together, and find the remainder mod r^n. Multiplying pairs of numbers and adding n numbers together can be done with TC^0 circuits [10]. If we have a number in binary, it is easy to find its remainder mod r^n by dropping all but the low order n^3 bits. To raise the number $ra_{j,i}$ to the kth power we multiply together k copies of the n^3-bit integer $ra_{j,i}$ in uniform TC^0.

Because we have shown that there are uniform TC^0 circuits maintaining the transitive closure of a relation, starting with an empty relation and no precomputed data, we have our main result:

Theorem 6. Transitive Closure $\in \mathrm{DynTC}^0$.

7. Transitive closure has a polynomially bounded SQLIES

To construct a polynomially bounded SQLIES for transitive closure using our dynamic TC^0 algorithm, we must address differences between the two models of dynamic computation. We must show the TC^0 computations, expressed as FO(COUNT) formulas, can be expressed as SQL queries. We must also show that the size of the auxiliary database can be kept polynomial in the size of the input relation, not polynomial in a fixed parameter n. As part of this, we must show how to update the database on adding or deleting a vertex from the graph.

The FO(COUNT) formulas expressing the updates to the auxiliary data describe relations over the universe $\{0,\ldots,n-1\}$. They also include negation, which is not expressible in SQL. If we maintain a unary relation containing all the elements $\{0,\ldots,n-1\}$, though, we can express negation as the difference of two relations. The input binary relation in our SQLIES for transitive closure may not be over the domain $\{0,\ldots,n-1\}$. It may contain ordered pairs over an arbitrary domain with n elements. Therefore, we will also maintain a 1–1 correspondence between the domain of the input relation and the integers $\{0,\ldots,n-1\}$, so that all auxiliary relations in the database can be maintained as relations over the numeric domain, as described in our FO(COUNT) dynamic algorithm.

The existence of this numeric domain, in addition to the domain of the input relation, is certainly guaranteed by SQL. It would seem that any query language with aggregation operators would necessarily have a domain of integers, since the result of counting the number of tuples in a relation is an integer. The existence of BIT, or equivalently addition and multiplication over this numeric domain, also seems to be implied by the existence of aggregation operators; it is certainly part of SQL. Addition and multiplication could be simulated by counting the tuples in a database that is either the union or Cartesian product of two databases with the appropriate number of tuples.

As stated before, we do not assume that addition and multiplication on numbers with $O(n)$ bits is part of the database language. These large numbers are not represented using the database's numeric type, but are represented as relations over the database's numeric type. An n^3 bit integer is represented as a relation $A(i,j,k)$ that contains the tuple (i,j,k) iff bit in^2+jn+k of the integer is 1. Addition and multiplication on these large numbers is performed using SQL queries that implement the FO(COUNT, $\leqslant$,BIT) formulas describing the relation that encodes the sum or product.

We keep the size of the auxiliary database polynomial in the size of the input relation by dynamically changing the size parameter n to be equal to the size of the active domain of the input relation. We are therefore allowing the addition of a vertex to the graph by allowing the insertion of an edge containing that new vertex. When the last edge including a vertex is removed from the graph, that vertex is automatically deleted. The addition of a new vertex requires that a new pair be added to the 1–1 correspondence between vertices and numbers, and that the number n be added to the unary relation containing the entire numeric domain. The counts $p_{i,j}(k)$, for $0 \leqslant k < n$, which are n^2-bit integers represented as binary relations over the universe $\{0,\dots,n-1\}$, must be reencoded as binary relations over the universe $\{0,\dots,n\}$; this just requires a reshuffling of the bits. Finally, the counts $p_{i,j}(n)$ must be computed. But since every path of length n is the unique composition of a path of length $n-1$ and a path of length 1, these numbers can be computed from $p_{i,j}(n-1)$ and the input relation. All these computations can be expressed by SQL update queries. We should do these computations, adding a vertex and increasing n, before performing the update required to add the edge. In the opposite case, we should perform the update needed to delete an edge before removing the unneeded vertex and decreasing n.

Since we have resolved all of the issues arising from the differences between the DynTC^0 model and the SQLIES model in the case of this algorithm, we have proved that there exists a polynomially bounded SQLIES that maintains the transitive closure of a binary relation.

Theorem 7. Transitive Closure $\in$ polynomially bounded SQLIES.

A final note is that the algorithm as stated does not allow us to delete a vertex with all of its associated edges in one step. However, an update formula for deleting a vertex and all its edges even simpler than the update for deleting a single edge can be found, and is as follows:

$$a'_{s,t} = a_{s,t} - \sum_{k=0}^{n-2} a_{s,i}(-1)^k (a_{i,i} - 1)^k a_{i,t} \bmod r^n.$$

8. Conclusions

A major consequence of Theorem 6, Transitive Closure $\in$ DynTC^0, is that transitive closure can be maintained using SQL update queries while keeping the size of the auxiliary relations polynomial in the size of the input relation. As transitive closure has been used as the prime example of a database query not expressible in query languages without recursion, non-recursive algorithms for maintaining transitive closure are of significant interest. Our result reduces the space required by a non-recursive algorithm from exponential in the size of the input relation to polynomial in the size of the input relation.

This new algorithm does not, however, lessen the importance of finding efficient sequential algorithms for maintaining transitive closure in databases. The dynamic algorithm given here is unlikely to be usefully implemented in practice, because its work complexity is greater than the best sequential dynamic algorithms. For example, the transitive closure of a symmetric relation (i.e. undirected graph reachability) can be maintained by sequential algorithms with polylogarithmic amortized time per operation [9].

Though this algorithm may not be practically useful in contexts where total work is the crucial constraint, it is an important upper bound for the following reason. Parallel complexity classes, using constant or logarithmic time, and polynomial work, have been seen to be the natural complexity classes smaller than P in the study of static complexity. They are robust under changes in encoding, and have natural connections to descriptive complexity classes. If dynamic complexity classes are similar to static complexity classes, then discovering what dynamic problems are in DynFO, DynTC^0, and other similar classes may be important. Since the dynamic complexity of problems is often less than the static complexity, the lower complexity classes may be even more important than in static complexity. This new upper bound for the dynamic complexity of directed reachability helps us to understand the landscape of dynamic complexity better. We now know that all special cases of reachability can be placed in dynamic complexity classes below or equal to DynTC^0.

The algorithm given in this paper does not rely on the uniqueness of edges between two vertices, and thus performs correctly when applied to a multigraph containing multiple edges and self loops. The counts $p_{i,j}$ remain integers with $n^{O(1)}$ bits when we allow up to 2^n edges between each pair of vertices. Thus, the algorithm we give can be interpreted as an algorithm for maintaining the powers of an integer matrix.

The problem of directed reachability in a graph may be reduced to the problem of finding the entries of the nth power of the adjacency matrix of the graph. Our algorithm is based on this reduction, and our algorithm can be modified to maintain dynamically the first $n^{O(1)}$ powers of an arbitrary n by n integer matrix A with entries of $n^{O(1)}$ bits. Our algorithm can be seen upon inspection to handle correctly the addition of self-loops and multiple edges to our graph, so it correctly handles adjacency matrices with non-zero diagonal and entries other than 1. In our algorithm, we picked our constant $r = 2^{n^2}$ sufficient to separate the values which would be the entries of the kth power of the adjacency matrix, or any 0/1 matrix. For arbitrary matrices, we need to pick a larger r with polynomially many bits, sufficient to keep the entries separated. This dynamic algorithm will then allow arbitrary changes of a single matrix entry, and queries on the value of any entry of any of the computed powers of the matrix.

There are open questions related to this paper in many directions. It is, of course, still an open question whether REACH is in DynFO. Many other restricted subclasses of graph reachability have not yet been investigated. We suspect that directed grid graph reachability and plane graph (planar graphs with a fixed embedding) reachability are in DynFO. There may be problems that are complete for these dynamic complexity classes, and logics that express exactly those dynamic problems in some dynamic complexity class. The largest open question, however, is whether dynamic complexity classes exist that are as robust as the familiar static complexity classes, and what their relation is to these static complexity classes.

References

[1] E. Allender, D.A.M. Barrington, W. Hesse, Uniform circuits for division: consequences and problems, in: Proc. 16th Ann. IEEE Conf. on Computational Complexity, 2001, pp. 150–159.

[2] D.A.M. Barrington, N. Immerman, H. Straubing, On uniformity within NC^1, J. Comput. System Sci. 41 (1990) 274–306.

[3] A. Chiu, G. Davida, B. Litow, Division in logspace-uniform nc^1, Theoret. Inform. and Appl. 35 (2001) 259–275.

[4] G.I. Davida, B. Litow, Fast parallel arithmetic via modular representation, SIAM J. Comput. 20 (4) (1991) 756–765.

[5] G. Dong, L. Libkin, L. Wong, On impossibility of decremental recomputation of recursive queries in relational calculus and SQL, in: Internat. Workshop on Database Programming Languages, Gubbio, Italy, 1995.

[6] G. Dong, J. Su, Incremental and decremental evaluation of transitive closure by first-order queries, Inform. and Comput. 120 (1) (1995) 101–106.

[7] K. Etessami, Dynamic tree isomorphism via first-order updates to a relational database, in: Proc. 17th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (PODS '98), Association for Computing Machinery, New York, June 1998, pp. 235–243.

[8] W. Hesse, Division is in uniform TC^0, in: Proc. 28th Internat. Colloq. on Automata, Languages and Programming, Crete, July 8–12, 2001, Springer, Berlin, pp. 104–114.

[9] J. Holm, K. de Lichtenberg, M. Thorup, Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge and biconnectivity, in: Proc. 30th Ann. ACM Symp. on Theory of Computing (STOC-98), New York, May 23–26, 1998, ACM Press, New York, pp. 79–89.

[10] N. Immerman, Descriptive Complexity, Springer, New York, 1999.

[11] L. Libkin, L. Wong, Incremental recomputation of recursive queries with nested sets and aggregate functions, in: Proc. Database Programming Languages (DBPL'97), Estes Park, CO, Lecture Notes in Computer Science, Vol. 1369, Springer, Berlin, 1998, pp. 222–238.

[12] S. Patnaik, N. Immerman, Dyn-FO: A parallel, dynamic complexity class, J. Comput. System Sci. 55 (2) (1997) 199–209.

[13] J.H. Reif, S.R. Tate, On threshold circuits and polynomial computation, SIAM J. Comput. 21 (5) (1992) 896–908.