GRAPH-THEORETIC METHODS IN DATABASE THEORY

Mihalis Yannakakis

AT&T Bell Laboratories
Murray Hill, NJ 07974

1. INTRODUCTION

As in many areas of computer science and other disciplines, graph theoretic tools play an important role also in databases. Many concepts are best captured in terms of graphs or hypergraphs, and problems can then be formulated and solved using graph theoretic algorithms. There is a great number of such examples from schema design, dependency theory, transaction processing, query optimization, data distribution, and a host of other areas. We will not attempt to touch on the wide range of all these applications. Rather, we will concentrate on a particular, basic type of problems that has attracted a great deal of attention in the database literature over the last few years and has come to play a central role: techniques for searching graphs and computing transitive closure, and some of the applications and related problems in query processing. There is an extensive literature on these types of problems, which we cannot reasonably hope to cover in this space, but we shall give a flavour of the issues that arise in solving these problems in various frameworks.

In Section 2 we review the basic algorithms for searching graphs and computing transitive closure and the generalization to semiring computations, assumming the standard sequential, main memory model of computation. In Section 3 we discuss algorithms when there is not enough space to hold the whole graph. Section 4 concerns parallel algorithms. In Section 5 we discuss recursive queries, and path problems on database graphs. Section 6 concerns the dynamic problem of processing on-line updates and queries. Section 7 contains concluding remarks.

2. GRAPH SEARCHING BASICS

We will be concerned with *directed* graphs here, and regard undirected graphs as the special case when the graph is symmetric (for every arc, the graph contains its inverse arc). The most fundamental graph problem is to compute reachability information. In the *all-pairs transitive closure* problem, we are given a graph G and want to compute the set of all pairs of nodes (u,v) such that u can reach v, i.e., there is a path in G from node u to node v. In the *single-source* transitive closure problem, we want to compute only the nodes that are reachable from a specified source node s; similarly, in the dual *single-sink* problem, we want to find all the nodes that can reach a given sink node t. Finally, in the single-source, single-sink problem (or reachability, or graph accessibility problem, abbreviated sometimes as *GAP*), we are given a source node s and a sink node t and we want to decide whether there is a path from s to t.

We will summarize here the basic algorithms and complexity results on these problems; these topics are covered in most textbooks on algorithms, for example [AHU]. First, we note that the algorithms for the GAP problem do not take much advantage of both the source and sink specification. That is, we find a path from a given node s to another node t by either solving the single source problem for s or the single sink problem for t. Naturally, the search can be stopped when it encounters the other specified node, or we may run the two searches simultaneously until they meet; this does not decrease in general the complexity, but sometimes it helps, for example in geometric graphs [SeV].

The single source problem is solved by a graph *search* or *traversal* method: starting with the source node s, we form the reachable set $R(s)$ visiting nodes one by one, where we can "visit" a new node y, if it has an arc (x,y) from an already visited node x. There are different search strategies depending on how the next node y is chosen. In *depth-first search* (DFS) we choose a node y that is adjacent to the most recently visited node x. In *breadth-first search* (BFS) we choose a node that is adjacent to the earliest visited node x. During a search we can

compute a tree T to recover paths from the source to the reachable nodes. The root of T is the source node s, and the parent of each other node y is that node x from which y was visited. The tree takes its name from the search strategy: DFS or BFS tree.

Depth-first-search is useful for problems having to do with connectivity properties of the graph: finding biconnected and triconnected components of undirected graphs, and finding strongly connected components of directed graphs. Breadth-first-search is useful for finding shortest paths from the source s to the other nodes (where the length of a path is its number of arcs): in BFS the nodes are added to the reachable set in order of their distance from s, and a shortest path from s to each node can be found in the BFS tree. Another strategy is *maximum cardinality search* MCS, where we choose a node y that is adjacent to the largest number of already visited nodes. This search strategy (and lexicographic BFS, a more complicated version of BFS) are useful in recognizing chordal (undirected) graphs and finding a perfect elimination ordering of the nodes. The hypergraph version of MCS is useful in recognizing and processing acyclic hypergraphs.

Let G be a graph with n nodes and e edges. If the graph is given by its adjacency list representation (i.e., for every node we have a list of its adjacent nodes), then all these search strategies can be implemented easily to run in linear time $O(n+e)$ (typically e is the dominant term, for example if every node has an incident edge). In the case of undirected graphs, linear time suffices to find also the connected components of the graph, which give the reachability information among all pairs of nodes. In the case of directed graphs, the all-pairs problem is more complicated because reachability is not a symmetric relation; in this case the all-pairs problem can be solved in $O(ne)$ time by solving n single-source problems, one for each source node. The hard case is when the graph is acyclic, in the sense that the problem for general graphs can be reduced in linear time to the acyclic case: We can partition the nodes of the graph into strongly connected components (using depth-first search), and then shrink the components to form an acyclic graph (see eg [AHU]).

There is a family of other algorithms, which are appropriate if we want to solve the all-pairs problems and the graph is dense, i.e., the number of edges is near n^2. These are best viewed as algorithms that manipulate the adjacency matrix of the graph. The adjacency matrix is the $n \times n$ Boolean matrix A whose rows and columns correspond to the nodes of the graph. The entry A_{ij} is 1 or 0 depending on whether the graph contains an edge from i to j or not. If we raise A to the kth *Boolean* power (i.e., + is logical 'and' and $\times$ is logical 'or'), the resulting matrix has the ij entry equal to 1 or 0 depending on whether there is a path of length k from node i to node j. If I is the $n \times n$ identity matrix, then the matrix $(I+A)^k$ has its ij entry equal to 1 or 0 depending on whether there is a path from i to j of length *at most* k. To compute the (reflexive and) transitive closure of A, denoted A^*, it suffices to raise $I+A$ to the nth power. The nth power can be computed by squaring the matrix $\log n$ times. That is, all-pairs transitive closure can be reduced to $\log n$ multiplications of matrices of dimension n.

Transitive closure can be also reduced to *one* multiplication of matrices of larger dimension $3n$ (see [AHU]), and thus computing the transitive closure and the product of two matrices is of the same order of complexity. Using the fast matrix multiplication algorithm of Coppersmith and Winograd [CW], the all-pairs transitive closure problem can be solved in time $O(n^{2.37})$. In general, this is the theoretically fastest algorithm known, but the constants are too large for it to be practical. Warshall's algorithm [W1] and its modification by Warren [W2] are practical matrix-based algorithms of complexity $O(n^3)$. These algorithms have also the property that they can be extended to more general types of path problems.

There are several papers that develop more efficient algorithms for "random" graphs, i.e., graphs constructed by choosing with a fixed probability whether to include each arc or not in the graph. These algorithms run in expected time $O(n^2)$, although their worst-case time is still $O(n^3)$ [BFM, S, Si].

Generalized transitive closure. In the more general setting, every arc a of the graph has a label $l(a)$ which comes from a domain D equipped with two operations "sum" + and "product" $\times$. For every path in the graph, we can define its label to be the product of the labels of its arcs. The label $L(u,v)$ of a pair of nodes u, v is defined to be the sum of the labels of all the paths from u to v. The operations + and $\times$ are assumed to satisfy several properties which make this quantity well-defined, and which are useful in computing labels efficiently. A *closed semiring* satisfies the following properties:

1. There are elements 0 and 1, which are (both left and right) identities for + and $\times$ respectively; furthermore, 0 is an annihilator for $\times$.
2. $\times$ is an associative operator, and D is closed under finite products.
3. Sum is an accosiative and commutative operator, and D is closed under infinite sums.
4. Multiplication distributes over (infinite) sums, both from left and right; i.e., $a \times (\Sigma b_i) = \Sigma(a \times b_i)$ and $(\Sigma a_i) \times b = \Sigma(a_i \times b)$.

The reason for assumption 3 is that there might be an infinite number of paths from one node u to another node v. If the graph is acyclic, we only need to consider finite sums.

There is a number of problems that can be formulated in terms of closed semirings by interpreting + and $\times$ appropriately: transitive closure, shortest paths, bill of materials, critical paths, regular expressions, and others; we will not give their definitions here — see [AHU, C, U]. These problems can be solved by the following algorithm, versions of which have been discovered many times in different contexts (Kleene's algorithm for regular expressions, Floyd's algorithm for shortest paths,

Warshall's algorithm for transitive closure). The matrix A^0 holds initially the labels l of the arcs of the graph, where $A^{0_{ij}} = 0$ if the arc $i \rightarrow j$ is not present. After the kth iteration, $A^{k_{ij}}$ is the sum of the labels of all paths from i to j that use only the first k nodes. Superscripted matrices are used below for clarity: we do not need n matrices, but can do the computation in place, except that if addition is not idempotent, then we must be careful to order properly the pairs i, j in the inner loop. The algorithm assumes that we can compute the *closure* (or asteration) $a^* = \sum_0^\infty a^i$ of an element a of D. Assuming that the operations +, × and * take unit time, the complexity is $O(n^3)$.

for $k := 1$ **to** n **do**
  for all i, j **do**
    $A^k_{ij} := A^{k-1}_{ij} + A^{k-1}_{ik} \times (A^{k-1}_{kk})^* \times A^{k-1}_{kj}$

Unlike the simple transitive closure problem, in the generalized problem we cannot always take advantage of (1) sparsity in the graph, and (2) restriction to single source or single sink problem. The easy case here is when the graph is acyclic and the hard case when the graph is strongly connected, in the sense that the problem for a general graph can be essentially reduced to its strongly connected components with linear overhead. In the acyclic case the single source problem can be solved in $O(e)$ time, and therefore the all-pairs problem in time $O(ne)$, as follows. First, we compute a topological ordering of the nodes, that is, an ordering from 1 to n so that edges go from lower to higher nodes. Suppose we want to solve a single source problem, and assume without loss of generality that the source is node 1 and it can reach all the other nodes (we only care about nodes reachable from the source). Below, we use $l(k,j)$ for the label of arc $k \rightarrow j$, and compute the sum of the labels of all the paths from node to 1 to the other nodes j into $L(1,j)$.

for all j **do** $L(1,j) := l(1,j)$;
for $k := 2$ **to** n **do**
  for each immediate successor j of k **do**
    $L(1,j) := L(1,j) + L(1,k) \times l(k,j)$

For some classes of graphs one can do better than $O(n^3)$ using techniques from data flow analysis and linear algebra. Tarjan presents in [T2] a decomposition technique based on dominator trees, and shows how to solve the single source problem on reducible flow graphs in almost linear time. And of course, for particular types of semiring problems, there are specialized algorithms that do better on general graphs (as, for example, in the special case of the ordinary transitive closure problem or shortest paths with positive weights). Indeed, a great deal of work in graph algorithms aims at developing efficient solutions for important cases of semiring problems, like shortest paths.

The following table summarizes the upper bounds that we know in path problems. In the table we assume $e \geq n$, and have dropped for clarity the big $O(\cdot)$ notation. $M(n)$ denotes the time needed to multiply two $n \times n$ Boolean matrices. The standard algorithms used in practice for all-pairs transitive closure on dense graphs take time $O(n^3)$, as does the ordinary matrix multiplication algorithm. In the generalized TC it is also true that the all-pairs problem is equivalent to the multiplication of two matrices over the closed semiring, but in general there is no "fast" method and multiplication using only the semiring operations requires $O(n^3)$ time.

	single source	all pairs
simple TC sparse graphs	e	ne
simple TC dense graphs	n^2	$M(n)$
generalized TC acyclic graphs	e	ne
generalized TC cyclic graphs	n^3	n^3

3. LIMITED SPACE ALGORITHMS

Transitive closure is an important problem in studying space complexity classes. The graph reachability problem is in NSPACE($\log n$), i.e., it can be solved by a nondeterministic algorithm using logarithmic extra space (besides the input); the algorithm just "guesses" a path from the source to the sink node by node. Conversely, a nondeterministic computation that uses a given amount of extra space S can be modeled by a reachability problem in a graph of exponentially larger size c^S, whose nodes correspond to the configurations of the machine and the arcs correspond to the transitions. Graph reachability is a complete problem for the class NSPACE($\log n$). Immerman [Im] and Szelepcsenyi [Sz] showed recently the surprising result that nondeterministic space complexity classes are closed under complementation, by providing a $\log n$-space nondeterministic algorithm for verifying that a source node *cannot* reach a sink node.

The relation of nondeterministic to deterministic space complexity is an old open problem. By Savitch's result, NSPACE($\log n$) is contained in DSPACE($\log^2 n$). The deterministic $\log^2 n$-space algorithm for transitive closure that is implied by this simulation rediscovers the same path facts over and over again, and as a consequence, its time complexity is not even polynomial, but $n^{\log n}$. Very little has been achieved in designing algorithms that are simultaneously space- and time-efficient. At present, we do not know of any algorithm that works in polynomial time with less than linear space *. The

* For undirected graphs, or more generally, Eulerian directed graphs (the number of arcs into each node is equal to the number of arcs coming out), it is possible to achieve polynomial time with a randomized algorithm that uses $\log n$ bits of space [A+]: A random walk will visit will high probability every reachable node within polynomial time, and of course to implement

algorithm that computes the transitive closure by repeatedly squaring the adjacency matrix (the "logarithmic" algorithm) can be implemented in two different ways, one time-efficient (the standard implementation), the other space-efficient (with $O(\log^2 n)$ bits of extra space). Tompa proves that no implementation of this algorithm can achieve both properties: if an implementation uses sublinear space, then it must take superpolynomial time [To]. He also shows that any implementation of Warshall's algorithm requires linear space.

The above space complexity model is appropriate if we can generate the graph efficiently in main memory (say, given a node, we can produce its adjacency list, or we can test whether a given edge is present), and we want to perform the whole computation in main memory. This may be the case if the graph is defined implicitely by some succinct description, for example, a program defining a view derived from relations that are not too large, or a hierarchical specification of a circuit.

Suppose that we want to compute the transitive closure of a database relation that resides in secondary memory but is too large to fit in main memory. As in the above setting, we want to make most efficient use of the limited space available. The important difference here is that actually, we have essentially as much space available as we want, except that it is in secondary memory and it costs to access it. Thus, the major goal here is to minimize the *Input/Output* (I/O) complexity of the algorithm, i.e., the amount of data that it moves between main and secondary memory, while keeping of course the time complexity within reasonable bounds. An additional concern follows from the fact that the unit of transfer between main and secondary memory is not an individual data item, but a page or block.

There has been some work adressing I/O complexity issues for various problems. McKellar and Coffman [MC] studied methods for storing matrices in a paged memory system, and presented algorithms with low I/O complexity for performing various operations, such as transposition, matrix multiplication etc. They found that storing and manipulating a matrix by rows or columns (or stripes of rows or columns) is inferior to a scheme that partitions the matrix into square submatrices of appropriate size. With the latter scheme, matrix multiplication can be performed with I/O complexity $O(n^3/\sqrt{s})$, where s is the size of the main memory. The algorithm allows for complete blocking, so that if b is the block size, the I/O measured in block transfers is reduced by b. Hong and Kung [HK] examine also the I/O complexity, and develop techniques to prove lower bounds, where they model algorithms by their computation DAGs. For the matrix multiplication problem, they show that no implementation of the ordinary algorithm can beat the above partitioning scheme. Aggarwal, Chandra and Snir analyze a hierarchical model where there are different levels of memory [ACS].

A number of papers in the database literature over the last few years have studied the problem of computing the transitive closure of a very large relation, obtaining experimental and analytical results on various implementations [AJ, I, IR, Lu]; we refer to [BS] for a survey and comparison. If we want to solve the single-source problem, and we have space of $O(n)$ words (with each word capable of holding a node identifier, i.e., $\log n$ bits), then it is easy to achieve minimal I/O, under reasonable assumptions. For example, if we apply a breadth-first search, we have to read the succesor list of every node only once. If we have less than linear space, then it does not appear possible to perform an efficient search (such as BFS or DFS), without encountering frequent page faults, since we do not have enough space to remember whether we have visited each node earlier or not. Finding a good way to do this, is a major open problem also in other areas.

The I/O complexity of transitive closure as a function of the space s and the numbers n, e of nodes and edges of the graph is analysed in [UY1]. In the dense case (e close to n^2), the situation is similar to the problem of matrix multiplication: $O(n^3/\sqrt{s})$ I/O complexity can be achieved by a scheme which partitions the nodes into $n/\sqrt{s}$ groups of $\sqrt{s}$ nodes each, and then essentially applies Warshall's algorithm to the groups. This algorithm works also for the generalized transitive closure problem where we have a closed semiring. A matching lower bound holds for a class of algorithms called "standard" in [UY], a property that is satisfied by usual algorithms that extend to semiring computations.

The sparse case is more complicated. If the graph is acyclic, then there is an algorithm that achieves I/O complexity $O(n^2\sqrt{e/s})$. This algorithm uses a more sophisticated partitioning scheme that takes into account the density of the graph. The algorithm is standard and thus generalizes to TC on semirings, and furthermore, it is optimal among standard algorithms. If the graph is not acyclic and the space s is at least $O(n)$, then it is still possible to solve the ordinary Transitive Closure problem with the same I/O complexity. However, in this case the algorithm is nonstandard and does not solve the generalized TC problem. It is not known whether one can do better than the dense scheme if one wants to solve the generalized TC problem on sparse cyclic graphs.

Related issues occur in other areas, for example, in verifying properties of communication protocols (see eg [Ho]). Protocols can be modeled by (typically huge) finite state machines, i.e. directed graphs, and one needs to analyze properties of the states that are reachable from some distinguished start state, for example whether it can reach a deadlock state. The algorithmic issues that arise in this setting have common features with both the space complexity and the I/O complexity models.

this, we only need to remember at each point the current node of the walk ($\log n$ bits). For general directed graphs this idea does not work: a random walk can get easily trapped, and needs exponential time to visit all the nodes.

4. PARALLEL ALGORITHMS

Theoretically, transitive closure is a higly parallelizable problem; it belongs to the class NC of problems that can be solved in polylogarithmic time (i.e., $O(\log^c n)$ for some constant c) with a polynomial number of processors. This has been extended significantly to more general classes of recursive queries: linear and piecewise linear queries, programs with the polynomial fringe property, and the polynomial stack property [AP, CS, UV]. On the other hand there are Datalog queries (even some simple ones) whose evaluation is a P-complete problem.

The paper [K] by Kanellakis presents a thorough overview of results concerning the parallel complexity issues in logic programming. As he points out, although the basic problem of transitive closure is in NC, in many respects we still lack a good parallel algorithm. A straightforward NC algorithm is the algorithm that squares $\log n$ times the adjacency matrix. It can be easily implemented on the usual models of parallel computation, for instance an Exclusive-Read-Exclusive-Write PRAM, to run in $O(\log^2 n)$ time using $n^3/\log n$ processors. In fact, $M(n)/\log n$ processors suffice, where $M(n)$ is the time for matrix multiplication. Since the total amount of *work* (processor-time product) of a parallel algorithm cannot be lower than the sequential time complexity, the straightforward algorithm is within a log factor of optimal in work if we want to solve the all-pairs problem on a dense graph. However, this algorithm is far from optimal if (a) the graph is sparse, or (b) we want to solve the single-source problem. If we consider n^3 as being the time of matrix multiplication in practice, then the work of the straightforward algorithm can be as much as n^2 off from optimal.

The stumbling factor is that we do not have any good parallel search algorithms. It is not known how one can perform a search from a source node in polylogarithmic time, without essentially computing the full all-pairs transitive closure. With some effort, breadth-first search can be accomplished with the same work ($M(n)\log n$) as transitive closure [GM]. Performing a depth-first search in parallel is considerably more difficult than transitive closure. It was considered for some time to be an inherently sequential process, and it is only recently that a randomized NC algorithm has been discovered. Unfortunately, it uses more processors than Transitive Closure: $nM(n)$ processors for $\log^7 n$ time [AAK]. It is not known whether there is a deterministic NC algorithm for DFS, even in the undirected case. In general, it appears that DFS is not a well-suited technique for parallel computation.

There are many other important problems that can be solved in linear time sequentially by searching: testing acyclicity of a graph, computing a topological order of the nodes, finding the strong components. These problems can be placed in NC, but the only way we know how to do this is by solving the more general all-pairs TC problem. This is sometimes refered to as the transitive closure bottleneck problem.

There is a lot of research devoted to developing more efficient solutions for interesting classes of graphs. For example, undirected graphs are better behaved. One can find the connected components of undirected graphs in NC with essentially linear speed-up [SV], and there are also efficient algorithms for finding the biconnected, triconnected components, and for a number of other important problems. Planar directed graphs are also easier to handle; for example, one can find the strong components in NC with $O(n)$ processors [KS].

Another direction is to consider general graphs, allow more than polylogarithmic time, and find algorithms which utilize fewer processors more efficiently. If we have at least $O(n)$ time, then almost linear speed-up can be easily achieved both for the single-source and the all-pairs case. The problem is challenging for sublinear times. This approach is taken in [UY2], where probabilistic algorithms are developed that use time that grows as a ε-root of n. The all-pairs Transitive Closure problem can be solved in $\tilde{O}(n^\varepsilon)$ time with $\tilde{O}(en^{1-\varepsilon}+n^{3-3\varepsilon})$ processors (the notation $\tilde{O}$ is a shorthand for "within log factors"). Thus, an almost linear speed-up is possible as long as one does not aim for very small times and the graph is not too sparse: $e \geq n^{2-2\varepsilon}$.

One can also take some advantage of a single-source restriction: the single-source TC problem can be solved within $\tilde{O}(n^\varepsilon)$ time using $\tilde{O}(en^{1-2\varepsilon}+n^{3-5\varepsilon})$ processors. For example, for $\varepsilon=1/2$, this gives $\tilde{O}(\sqrt{n})$ time with $O(e)$ processors. It is shown also that one can perform breadth-first search from a source node with the same bounds.

5. RECURSIVE QUERIES AND GRAPH DATABASES

Many database application domains require the computation of queries, such as transitive closure, that cannot be expressed in first-order languages like relational algebra and calculus [AU]. Solutions proposed to increase the capabilities of relational systems include the addition of a transitive closure operator as a primitive [A, Z], stepping up to a more powerful logic-based query language that permits recursion, or the use of a graph-oriented language [CMW, MW, RHDM].

Relational databases can be viewed as labelled "directed" hypergraphs, where the nodes are the domain elements, the labels are the database relation schemes, and the labelled edges are the tuples of the relations. If the relations are binary, then the database is simply a directed labelled graph (with multiple edges allowed). Consider such a labelled directed graph G, and let Σ be its set of labels. Every path in G "spells" a word over Σ, the word obtained by concatenating the labels of the edges of the path. If L is a language over Σ, then we say that a path is an *L-path* (or is in L) if it spells a word in L. We can define then the *L-transitive closure* problem where we restrict attention to L-paths. As in the ordinary TC problem, a source or a sink (or both) may be specified, or we may want to solve the all-pairs problem. In this section we shall discuss problems of this type.

There is a subtle, but important, point that arises in the presence of labels. In the ordinary transitive closure problem, if a node u can reach another node v, then it can reach it by a *simple* path, one that does not use any node or edge more than once. This is not true for L-paths: there may be an L-path from u to v but no simple L-path. Finding simple paths with desired properties in directed graphs is very difficult. Essentially every nontrivial property gives rise to an NP-complete problem [FH]: For example, it is NP-complete to tell whether there is a simple cycle containing two specified nodes; given three nodes x, y, z, it is NP-complete to tell whether there is a simple path from x to y that goes through z *. For this reason, we should allow for arbitrary paths, and will not require them to be simple. Mendelzon and Wood study conditions on the graph and the language under which the existence of an L-path implies the existence of a simple one [MW].

The class of (elementary) *chain* queries in Datalog corresponds to L-path graph queries for context-free languages L [BKBR, UV]. A chain rule is a rule of the form:

$$p(X,Y) :- q_0(X,Z_1),\ q_1(Z_1,Z_2),\ldots,\ q_k(Z_k,Y),$$

where the q_i's are EDB (base) or IDB (derived) binary predicates, and X, Y and the Z_i's are distinct variables. That is, the graph of the body of the rule is a simple directed path from the first to the second argument in the head of the rule. We can assume that for every EDB predicate q we have also its inverse q^{-1} so that the directions are not really important for the base predicates, but they are for the IDB predicates. A program P consisting of chain rules is a chain program. This is a restricted, but important class of programs. Taking the join of the predicates in the body of a rule in some order, requires merging predicates to form some acyclic hypergraph (where the edges may not be binary any more) [GS]. Chain rules represent the simplest type of acyclic bodies, where the join tree is just a path. Some query transformation methods that apply to special classes of recursion treat groups of EDB predicates in the bodies of the rules as black-box units, and in the process, often abstract them so that they look basically like chain rules except that the q_i's are conjunctions of predicates, and X, Y etc. are tuples of variables. These programs generalize the usual paradigms of the 'ancestor' query (i.e., transitive closure) in its different forms, and of the 'same generation' query on a graph with arcs labelled "flat", "up" and "down".

$sg(X,Y) :- up(X,Z)sg(Z,W)down(W,Y)$
$sg(X,Y) :- flat(X,Y)$

A chain program corresponds to a context-free grammar [UV]. The terminal and nonterminal symbols of the grammar correspond to the EDB and IDB predicates respectively, the initial symbol corresponds to the goal predicate and the productions of the grammar are obtained from the rules of the program by ignoring the variables. For example, the above rule corresponds to the production $p \rightarrow q_0 q_1 \cdots q_k$. Let L be the context-free language defined by this grammar, and let G be the database graph. That is, for every tuple (u,v) of an EDB predicate a, the graph G has an arc (u,v) labelled a. Then a tuple (x,y) satisfies the goal predicate if and only if there is an L-path from x to y in G.

A simple case is when L is a regular language [BKBR, MW, BKV]. Some examples that fall into this category are:
1. The ordinary transitive closure problem: all edges have the same label a and $L = a^*$ or a^+.
2. Even-length paths: $L = (aa)^*$.
3. Common ancestor problem: if a is a child-parent relationship and b its inverse, then two nodes x and y have a common ancestor if there is a path of the form a^*b^* from x to y.

Regular languages can be generated by left-linear grammars (or right-linear grammars), and thus regular path problems can be expressed by chain programs in Datalog that are one-sided (all rules are left-linear or all are right-linear). As shown in [BKBR], if L is regular (and only if), one can take full advantage of a selection in either one of the arguments of the goal predicate to rewrite the program so that the recursive predicates are monadic.

A problem that is equivalent to L-transitive closure with L regular, is the problem of evaluating an expression E that is built from a given set of binary relations (the EDB predicates) using the operators $\cup$ (union), $\cdot$ (composition), * (transitive closure) [HSU, SU]; E is the regular expression for L, with $\cdot$ being concatenation and * the Kleene star. One can include also the operator $^{-1}$ for taking the inverse of a relation; the inverse can be always pushed down to the base relations [SU].

The regular path problem can be reduced to ordinary transitive closure [HSU, MW]. Let M be a (deterministic or nondeterministic) finite automaton for the language L. For any pair of nodes x, y of the database graph G, the labels of the paths from x to y in the graph G form a regular language $P_{x,y}$; this is the language accepted if we regard M as a finite automaton with initial state x and accepting state y. There is an L-path in G from x to y if and only if the intersection of L with $P_{x,y}$ is nonempty. An automaton for the intersection can be constructed by taking the product of the automata for the two languages. That is, we construct a graph H whose nodes are pairs (s,u) consisting of a state s of M and a node u of G, and which has an arc labelled a from a node (s,u) to another node (t,v) if M has a transition on letter a from state s to state t and G has an arc from u to v. (Actually, the labels in the product graph are not important.) Let s_0 be the initial state of M and F its set of accepting states. Then, the database graph G contains an L-path from a node x to a node y iff (s_0,x) can reach in the product graph a state (t,y) with $t \in F$.

If F has more than one states, we can modify the graph so that a single-source single-sink L-path query on G translates

*For undirected graphs, properties of this type can be tested in polynomial time by the work of Robertson and Seymour [RS], but except for the simplest cases (patterns with very few nodes), the constants are extremely large.

to one ordinary path query of the same type on the new graph (instead of $|F|$ queries). Add to the product graph a node (s_f,u) for every node u of G, where s_f is a 'dummy', not an actual state of M, and add unlabelled arcs from all nodes (t,u) with $t \in F$ to (s_f,u). (This is equivalent to adding ε transitions in the automaton from the old final states to the new final state s_f.) Let H be the resulting graph. An L-path of G from x to y corresponds to a path of H from (s_0,x) to (s_f,y).

Suppose that M has k states, and G has n nodes and e labelled arcs. Then H has at most $n(k+1)$ nodes and ek^2+n arcs; the second term n accounts for the new dummy nodes and the arcs into them. If M is deterministic, then H has at most $ek+n$ arcs: every labelled arc of G, say arc (u,v) with label a, gives rise to k arcs in H; for every state s of M there is a unique arc $(s,u) \rightarrow (t,v)$ in the product since t is uniquely determined from s and a. For any fixed language, k is a constant, and thus the L-transitive closure problems have the same complexity as the corresponding ordinary TC problems: the single-source L-TC problem can be solved in $O(e)$ time, and the all-pairs problem in $O(ne)$ time.

Given a source node x or sink node y of G, we can view the search of H forward from the corresponding source (s_0,x), or backward from the corresponding sink (s_f,y), as the application of Horn rules that infer which nodes of G are reached in which combinations with states of M. We can interpret then the transformation from G to H into a transformation on logical (Datalog) rules rather than one on graphs. Introduce a unary IDB predicate s for every state s of M, including the dummy final state s_f; the fact $s(u)$ means that the node (s,u) of H is reached in the search forward from the source or backwards from the sink. In the single-source L-TC problem with node x as source, the seed is $s_0(x)$ and the goal is $s_f(Y)$. The rules reflect in the obvious way the construction of H and the fact that the search goes forward. For each transition $s \rightarrow t$ of M on letter a, there is a rule $t(V) :- s(U)a(U,V)$. The transitions $t \rightarrow s_f$ into the dummy final state from the states $t \in F$ give rise to the rules $s_f(V) :- t(V)$. In the case of the single sink problem, the roles are reversed in the natural way; for example the rule corresponnding to the transition from s to t on letter a is $s(U) :- a(U,V)t(V)$.

Example. Consider the following program of mixed linear rules, with query goal $p(x,Y)$, where x is a given constant.

$p(X,Y) :- a(X,Z),p(Z,Y)$
$p(X,Y) :- p(X,Z),c(Z,Y)$
$p(X,Y) :- b(X,Y)$

The arguments X and Y of p above need not be single variables, but could be tuples of variables, and could even have different arities. Also, a, b, c could really be conjunctions of EDB predicates which may also depend on other nondistinguished variables that do not appear in p. Note however, that if say a is a chain of EDB predicates, we do not want to actually compute their join a, because this may turn a sparse graph into a dense one.

The language generated by the grammar corresponding to this program is a^*bc^*. The corresponding automaton is shown below. For clarity, we did not use subscripts; the initial state is s, and the accepting state is t.

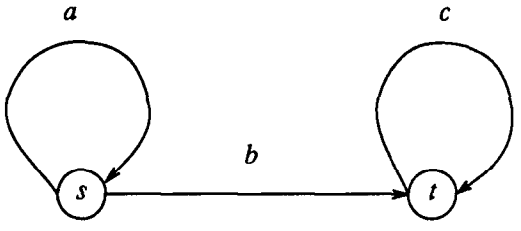

Figure 1

The new program is as follows (this is basically the same as the transformation of Algorithm 15.3 in [U2]).

$p(x,Y) :- t(Y)$
$t(V) :- t(U), c(U,V)$
$t(V) :- s(U), b(U,V)$
$s(V) :- s(U), a(U,V)$
$s(x)$
□

Let us consider now the case that the language L is linear (i.e., generated by a linear grammar), but not regular. A typical example is the same generation query where the language is $\{(up)^i flat(down)^i \mid i \geq 0\}$. The *counting* method [BMSU, HN, SZ] is inspired by this example. The same generation language is recognized by a (one-way) machine with one counter. If we regard the graph H above of the regular case as a version of the graph of "configurations" of the accepting automaton, the counting method is basically the corresponding natural extension, where the configuration of the machine includes now also the value of the counter. If the database graph is acyclic, the counter need not exceed n, the number of nodes, and thus the counting method yields a $O(ne)$ solution in this case. Extending the counting method to cyclic graphs is highly nontrivial. Haddad and Naughton developed an elegant and deep analysis of the lengths of paths in cyclic graphs, and used it to solve the same generation problem on arbitrary graphs with the same complexity bound $O(ne)$ [HN]. (Earlier papers had proposed other, less efficient solutions.)

Extending the counting method to programs with multiple recursive rules presents dificulties: one counter suffices for a language that corresponds to a program with a single linear recursive rule, but for two or more rules one may need a full pushdown store, and the number of different contents (configurations) becomes exponential.

There is a simple reduction to ordinary transitive closure which yields the same complexity as the same generation query,

and works for linear rules and cyclic graphs.

Theorem: Let L be a linear language. The all-pairs L-transitive closure problem on a graph with n nodes and e arcs can be solved in time $O(ne)$.

Proof: Let L be a linear language and G the database graph. We will construct a graph H with $O(n^2)$ nodes, $O(ne)$ edges, and a distinguished source node s, such that finding the nodes of H that are reachable from s answers the all-pairs transitive closure problem in G. A simple reduction described in [U2] yields a graph with $O(n^2)$ nodes and $O(n^4)$ edges in the worst case. The only trick is to rewrite the program before carrying out the reduction.

Obtain a linear grammar for L, so that in every production, the right-hand side has at most two symbols, a terminal and a nonterminal. Equivalently, transform in the obvious way the Datalog program so that the body of every rule has one EDB predicate and possibly one IDB predicate. For example, in the same generation query we split the recursive rule into two rules, a left-linear and a right-linear.

$sg(X,Y) :- up(X,Z),new(Z,Y)$
$new(Z,Y) :- sg(Z,W),down(W,Y)$
$sg(X,Y) :- flat(X,Y)$

The graph H has one node $p(x,y)$ for every IDB predicate p of the new program and every pair of nodes (x,y), and contains an additional source node s. Suppose that $p(X,Y) :- a(X,Y)$ is a nonrecursive rule. Then, for every a-arc $x \rightarrow y$ of G, the graph G contains one arc from s to node $p(x,y)$. Suppose that $p(X,Y) :- a(X,Z),q(Z,Y)$ is a recursive rule. Then, every a-arc $x \rightarrow z$ of G gives rise to n arcs in H of the form $q(z,y) \rightarrow p(x,y)$ for all values of y. Similarly, if $p(X,Y) :- q(X,Z),a(Z,Y)$ is a left-linear recursive rule, every a-arc $z \rightarrow y$ of G gives rise to n arcs in H of the form $q(x,z) \rightarrow p(x,y)$. Since the program has constant size, the graph H has $O(n^2)$ nodes and $O(ne)$ arcs.

Searching H from the source node s is the same as computing the new program bottom-up. □

Suppose that L is a general context-free language. The recognition problem for L corresponds to the special case of the single-source, single-sink L-Transitive Closure problem where the graph is a directed acyclic path: to test if a word w in in L, just let the graph G_w consist of a path that spells w. (Clearly, this is true for any language L, not only context-free.) There are various algorithms for the recognition problem for context free grammars. In particular, this problem can be solved in time $O(n^3)$ by the Cocke-Younger-Kasami dynamic programming algorithm (see eg [ASU]), which can be stated as follows: Take a grammar for L in Chomsky normal form, construct the corresponding Datalog chain program and evaluate it on G_w using bottom-up semi-naive. (Indeed, bottom-up evaluation is a form of dynamic programming.) The C-Y-K algorithm uses a particular, convenient order of evaluation. The same complexity, $O(n^3)$, suffices to carry out the bottom-up semi-naive computation on any graph, and thus to solve the all-pairs L-Transitive Closure problem.

Valiant has shown that the context-free recognition problem can be reduced to matrix multiplication [V]. This technique does not seem to generalize to arbitrary database graphs, though it does for acyclic graphs. Also note that, for parallel complexity, general graphs can be apparently more difficult: Context-free language recognition is in NC, but L-reachability for some context-free languages L is P-complete [UV, AP].

The following table summarizes the complexity of the L-Transitive Closure problems.

	single source	all pairs
regular	e	ne
linear	ne	ne
context-free	n^3	n^3

From the table we see that the regular case is as easy as ordinary transitive closure. We can take advantage of both sparsity and a single-source restriction. In the context-free case, in general we do not take advantage of either. Recall however, that the recognition problem corresponds to a graph that is essentially as sparse as it can get (n arcs) and with both source and sink specified. The linear case is in-between. It is as easy as ordinary TC for the all-pairs problem, i.e., if we want to materialize the whole relation. However, according to these bounds, no advantage is taken of having a bounded argument, a single source or single sink restriction. Of course, one could do the obvious optimizations, for example consider only reachable nodes, but this does not decrease the worst-case complexity. We conjecture that it should be possible to do better. A good place to start is the special case of the same generation query on acyclic graphs.

Finally, we note that the bounds can be extended to all binary programs with acyclic bodies. Suppose that we have a Datalog program where all the EDB and IDB predicates are binary (we may also have some unary). For each rule, form an *undirected* graph whose nodes are the variables, and the edges correspond to the literals in the body of the rule. Say that the rule is acyclic if its graph is acyclic, with self loops allowed.

Theorem: A binary acyclic program can be evaluated (on general database graphs) in time $O(ne)$ if the program is linear, and time $O(n^3)$ in general. □

The case of cyclic bodies is harder even in the absence of recursion. For example, given a cyclic join of binary relations, we do not know how to test whether it is empty in polynomial time $O(n^c)$ with c independent of the number of relations; if we consider the query as part of the input, the problem is NP-complete.

6. ON-LINE PROCESSING

As in traditional relational systems, if a database view is queried much more often than updated, we may want to keep data structures that allow us to answer queries faster than if we were to compute them from scratch. There has been relatively little work done on developing algorithms for graphs that change dynamically with edges and nodes being added or deleted. In particular, the problem of updating dynamically a directed graph so that we can compute reachability information fast is an especially difficult one.

Ibaraki and Katoh proposed an algorithm which maintains the transitive closure of a graph with complexity $O(n^3)$ for any number of insertions, and $O(n^2(e+n))$ for any number of deletions [IK]. Italiano studied the incremental problem, where there are only edge insertions [I1]. He proposed a data structure (we will describe it in somewhat different terms below) with the following time bounds:

1. Edge insertion takes $O(n)$ *amortized* time; that is, although an individual insertion may require more time than that, a sequence of r insertions takes time $O(rn)$.
2. Answering a reachability query for a given source, sink pair takes constant (worst-case) time. If there is a path from the source to the sink, then such a path can be returned in time $O(l)$, where l is the length of the path found.

A rationale from the database pont of view for considering the incremental problem is that databases often tend to grow, that is, insertions may be much more frequent than deletions. Buchsbaum, Kanellakis and Vitter extend these results on the incremental problem to the L-transitive closure problem when L is regular [BKV]. The complexity bounds are the same, with the constants depending on the language: the data structure supports arc insertions in $O(n)$ amortized time, and single-source, single-sink queries in $O(1)$ time (again the path can be constructed in $O(l)$ time). They also give partial extensions to some other cases, and handle the same generation problem on acyclic graphs with $O(n^2)$ amortized time per insertion and constant time per query.

The transformation of the previous section can be used to solve the incremental problem for a linear language L. First, consider the ordinary incremental transitive closure problem, and suppose that we know that all the queries that we will be asked have the same source node s.

Lemma: The incremental single-source TC problem can be solved so that arc insertions take constant time in the amortized sense, and queries in the worst-case sense.

It suffices to apply a standard search algorithm from the source node s and use an array, indexed by the nodes to indicate whether each node can be reached from s. The array need not be initialized because of the trick in an exercise of [AHU]. When an arc $u \rightarrow v$ is inserted, we add it to the adjacency list of u, and we examine whether u and v were already reachable. If v was already reachable or u was not, then we do nothing. Otherwise, we search out of v in the straightforward way. In the algorithm below, Q is an initially empty set (it can be implemented for example as a queue or stack), and *parent* gives the parent of each reachable node in a tree rooted at s so that we can recover a path; $parent(u) = nil$ signifies that u cannot be reached from s.

```
INSERT(u,v)
if parent(u) ≠ nil and parent(v) = nil then begin
    parent(v) := u; Q := {v};
    while Q ≠ ∅ do begin
        x := delete an element of Q;
        for each successor y of x do
            if par(y) = nil then begin
                insert y into Q;
                parent(y) := x
            end
    end
end
```

Every edge is traversed at most once, thus the amortized complexity is $O(1)$ per insertion. We can tell if a node is reachable, and find a path if it is, by following the *parent* pointers. Note that even if we search in a breadth-first-order in the INSERT procedure (i.e., Q is a queue), the tree is not necessarily a BFS tree, and thus the path from s to a node u that is returned may not be a shortest path.

If we want to answer queries for all possible sources, then we simply have a different *parent* array for every source node. That is, we use a two-dimensional array, so that $parent(s,u)$ is *nil* if s cannot reach u, and otherwise it is the parent of u in a tree rooted at s. The amortized insertion cost becomes $O(n)$. This is basically a simplified version of the data structure of [I1], where he uses also the following optimization: Suppose that an arc $u \rightarrow v$ is added, the head u is reachable from a source node s and the tail v is not. When we search out of v to discover the new descendants of s, instead of using the edges of the whole graph, we can just use the tree rooted at v. That is, in the above INSERT procedure, we compute also a *children* list for each node of the tree, and instead of examining the successors y of x, we examine its children in the tree rooted at v. Obviously, this can only help since the tree is smaller. Furthermore, it guarrantees that no insertion takes more than $O(n^2)$ time in the worst case *.

We return now to the L-Transitive Closure problem.

Theorem: Let L be a linear language. The incremental L-transitive closure problem can be solved with $O(n)$ amortized complexity for insertion, and $O(1)$ worst-case complexity for single-source, single-sink reachability query.

Proof: Recall the construction of the graph H in the previous

* Inserting a single arc in a graph may change the reachability status of $O(n^2)$ pairs. Thus, if an algorithm keeps track of the transitive closure at all times then it must have $O(n^2)$ worst-case insertion time. It is not clear whether one can avoid this while keeping the query time constant. In the undirected case, this is possible using a Union-Find data structure.

section. We maintain a data structure to answer ordinary reachability queries on H only about the distinguished source s. A query whether the database contains an L-path from a node x to a node y is translated into the query whether s can reach node $q(x,y)$, where q is the goal predicate. Suppose that the (rewritten) program corresponding to L has r rules. An insertion of a tuple in the database (insertion of a labelled arc in the graph G) is translated into at most rn arc-insertions into H, n for each rule. Thus, the amortized complexity of insertion is $O(rn)$, which is $O(n)$ since r is a constant. □

Clearly, the data structure can be easily extended so that we can answer single source (or single sink) queries in time $O(t)$, where t is the number of nodes reachable by L-paths (t could be much less than n).

Note that an incremental algorithm can be used to solve the all-pairs transitive closure problem by first inserting all the arcs one by one, and then querying all pairs of nodes. If the query time is constant, then the second phase takes $O(n^2)$ time. Since at present the algorithms for all-pairs TC take $O(ne)$ time, we cannot expect to spend less than $O(n)$ amortized time per insertion, at least with current techniques. It would be nice however, to improve the worst case complexity of insertion.

The same bounds hold for linear, binary acyclic programs. In the case of context-free languages L, and nonlinear acyclic programs, the amortized time for insertion becomes $O(n^2)$, with the total being no more than $O(n^3)$ for arbitrary number of insertions.

Handling deletions is more difficult. Italiano extended his data structure in [I2] to maintain the transitive closure of an *acyclic* graph under deletions only (no insertions). The complexity of his algorithm is $O(ne)$ time for any number of deletions, and $O(1)$ for a reachability query ($O(l)$ for reporting a path, where l is the path length). It can be shown that this algorithm too can be extended to the L-transitive closure problem when L is linear, with the same complexity bounds.

Maintaining a dynamic acyclic graph under insertions and deletions is a problem that has to be solved in other contexts, for example in deadlock prevention and detection, or if one wants to use a concurrency control algorithm based on a serialization graph. In the case of a general DAG, we do not know of any dynamic techniques that can guarrantee better complexity than the obvious bounds.

In the case of deadlock prevention with exclusive locks, the problem is easier. In this case the DAG, usually, called the "wait-for" graph, is a collection of trees, assuming that every transaction requests and releases its locks sequentially. There is one tree for each running transaction, which is the root of the tree. When a transaction T requests a lock on an item x, we look up if another transaction T' holds a lock on x. If this is the case, then T' is unique, and we test whether there is a path from T' to T, i.e., whether T is the root of the tree containing T'. If it is, then the request is denied, otherwise, T becomes a child of T'. When T' releases the lock, then the edge (T,T') is deleted. In the straightforward solution (no other data structures), the insertion time is proportional to the depth of T', which in general is $O(n)$, and the deletion time is constant. It is possible to achieve logarithmic worst-case complexity for both insertions and deletions. Can one do better?

7. CONCLUSIONS

Graphs and hypergraphs are important tools in the theory of databases. We have focused on transitive closure and its variants, and have examined algorithmic aspects in differents kinds of frameworks. Although transitive closure is a fundamental problem in this and other areas and has been the subject of many works, as we have seen, there are still many questions remaining, and there is a lot of room for improvement.

Some of the main concerns in query processing reflect and generalize similar concerns in graph algorithms. There is an obvious analogy between the objective of pushing selections in queries with the goal of taking advantage of single-source restriction in graph searching, and between the efforts to avoid Cartesian products with taking advantage of sparsity in graph algorithms.

Having recursion in the query language makes the task of finding good general methods for query optimization a very difficult and important task. Essentially, it amounts to developing general-purpose methods for generating automatically good algorithms for interesting classes of problems. One can view a logic program both as the specification of a problem and as a specification of an algorithm for solving the problem, if we use say, bottom-up semi-naive evaluation. The syntactic restrictions determine what class of problems we want to solve, and what is the class of possible algorithms that is being considered. A single logical query can be the specification of a challenging combinatorial problem. Transforming the program, as in magic sets, counting, etc, means choosing a particular algorithm for the given problem. There is some work on characterising the kind of problems that can be specified in logic languages, for example Datalog with or without $\neq$. It would be interesting to understand better these languages also in their algorithm-specification capacity. For example, how much do we loose if we only consider program transformation methods? What other methods are there that are manageable and at the same time have some generality?

REFERENCES

[AP] F. Afrati, and C. H. Papadimitriou, "The Parallel Complexity of Simple Chain Queries", *Proc. 6th ACM Symp. on Principles of Database Systems*, pp. 210-214, 1987.

[AAK] A. Aggarwal, R. J. Anderson, and M.-Y. Kao, "Parallel Depth-First Search in General Directed Graphs", *Proc. 21st ACM Symp. on Theory of*

Computing, pp. 297-308.

[ACS] A. Aggarwal, A. K. Chandra, and M. Snir, "Hierarchical Memory with Block Transfer", *Proc. 28th IEEE Symp. on Foundations of Computer Science*, pp. 204-216, 1987.

[A] R. Agrawal, "Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries", *Proc. 3rd Intl. Conf. on Data Engineering*, pp. 580-590, 1987.

[AJ] R. Agrawal, and H. V. Jagadish, "Direct Algorithms for Computing the Transitive Closure of Database Relations", *Proc. Intl. Conf. on Very Large Data Bases*, pp. 255-266, 1987.

[AHU] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[ASU] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[AU] A. V. Aho, and J. D. Ullman, "Universality of Data Retrieval Languages", *Proc. 6th ACM Symp. on Principles of Programming Languages*, pp. 110-120, 1979.

[A+] R. Aleliunas, R. M. Karp, R. J. Lipton, L. Lovasz, and C. Rackoff, "Random Walks, Universal Traversal Sequences, and the Complexity of Maze Problems", *Proc. 20th IEEE Symp. on Foundations of Computer Science*, pp. 218-223, 1979.

[BMSU] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman, "Magic Sets and Other Strange Ways to Implement Logic Programs", *Proc. 6th ACM Symp. on Principles of Database Systems*, pp. 1-15, 1986.

[BR] F. Bancilhon, and R. Ramakrishnan, "An Amateur's Introduction to Recursive Query Processing Strategies", *Proc. ACM SIGMOD Conf. on Management of Data*, pp. 16-52, 1986.

[BKBR] C. Beeri, P. C. Kanellakis, F. Bancilhon, and R. Ramakrishnan, "Bounds on the Propagation of Selection into Logic Programs", *Proc. 6th ACM Symp. on Principles of Database Systems*, pp. 214-226, 1987.

[BR] C. Beeri and R. Ramakrishnan, "On the Power of Magic", *Proc. 6th ACM Symp. on Principles of Database Systems*, pp. 269-283, 1987.

[BS] J. Biskup, and H. Stiefeling, "Transitive Closure Algorithms for Very Large Databases", TR, Hochschule Hildesheim, 1988.

[BFM] P. A. Bloniarz, M. J. Fischer, and A. R. Meyer, "A Note on the Average Time to Compute Transitive Closure", *Proc. Intl. Coll. on Automata, Languages, and Programming*, 1976.

[BKV] A. L. Buchsbaum, P. C. Kanellakis, and J. S. Vitter, "A Data Structure for Arc Insertion and Regular Path Finding", *Proc. ACM-SIAM Symp. on Discrete Algorithms*, to appear, 1990.

[C] B. Carre, *Graphs and Networks*, Oxford University Press, 1979.

[CW] D. Coppersmith, and S. Winograd, "Matrix Multiplication via Arithmetic Progressions", *Proc. 19th ACM Symp. on Theory of Computing*, pp. 1-6, 1987.

[CK] S. Cosmadakis, and P. C. Kanellakis, "Parallel Evaluation of Recursive Rule Queries", *Proc. 5th ACM Symp. on Principle of Database Systems*, pp. 280-293, 1986.

[CMW] I. F. Cruz, A. O. Mendelzon, and P. T. Wood, "A Graphical Query Language Supporting Recursion", *Proc. ACM SIGMOD Conf. on Management of Data*, pp.323-330, 1987.

[DKS] C. Dwork, P. C. Kanellakis, and L. Stockmeyer, "Parallel Algorithms for Term Matching", *SIAM J. on Computing*, 1988.

[FHW] S. Fortune, J. Hopcroft, and J. Wyllie, "The Directed Subgraph Homeomorphism Problem", *Theoretical Computer Science*, 10, pp. 11-121, 1980.

[GM] H. Gazit, and G. L. Miller, "An Improved Parallel Algorithm that Computes the BFS Numbering of a Directed Graph", *Information Processing Letters*, 28(1), pp. 61-65, 1988.

[GS] N. Goodman, and O. Shmueli, "The Tree Property is Fundamental for Query Processing", *Proc. 1st ACM Symp. on Principles of Database Systems*, pp. 40-48, 1982.

[GSS] G. Grahne, S. Sippu, and E. Soisalon-Soininen, "Efficient Evaluation for a Subset of Recursive Queries", *Proc. 6th ACM Symp. on Principles of Database Systems*, pp. 284-293, 1987.

[HaN] R. W. Haddad, and J. F. Naughton, "Counting Methods for Cyclic Relations", *Proc. 7th ACM Symp. on Principles of Database Systems*, pp. 333-340, 1988.

[HeN] L. J. Henschen, and S. A. Naqvi, "On Compiling Queries in First-Order Databases", *J. ACM*, 31(1), pp. 47-85, 1984.

[Ho] G. J. Holzman, "An Improved Protocol Reachability Analysis Technique", *Software-Practice and Experience*, 18(2), pp. 137-161, 1988.

[HK] J. W. Hong, and H. T. Kung, "The Red-Blue Pebble Game", *Proc. 13th ACM Symp. on Theory of Computing*, pp. 326-33, 1980.

[HSU] H. B. Hunt III, T. G. Szymanski, and J. D. Ullman, "Operations on Sparse Relations", *C. ACM*, 20(3),

pp. 171-176, 1977.

[IK] T. Ibaraki, and N. Katoh, "On-line Computation of Transitive Closure of Graphs", *Information Processing Letters*, 16(9), pp. 5-7, 1983.

[I] Y. E. Ioannides, "On the Computation of the Transitive Closure of Relational Operators", *Proc. Intl. Conf. on Very Large Data Bases*, pp. 403-411, 1986.

[IR] Y. E. Ioannidis, and R. Ramakrishnan, "Efficient Transitive Closure Algorithms", *Proc. 14th Intl. Conf. on Very Large Data Bases*, pp. 382-394, 1988.

[It1] G. F. Italiano, "Amortized Efficiency of a Path Retrieval Data Structure", *Theoretical Computer Science*, 48(2), pp. 73-81, 1986.

[It2] G. F. Italiano, "Finding Paths and Deleting Edges in Directed Acyclic Graphs", *Information Processing Letters*, 28(1), pp. 5-11, 1988.

[Im] N. Immerman, "Nondeterministic Space is Closed Under Complementation", *SIAM J. Computing*, 17(5), pp. 935-938, 1988.

[JAN] H. V. Jagadish, V. R. Agrawal, and L. Ness, "A Study of Transitive Closure as a Recursion Mechanism", *Proc. ACM SIGMOD Conf. on Management of Data*, pp. 331-344, 1987.

[K] P. C. Kanellakis, "Logic Programming and Parallel Complexity", in *Foundations of Deductive Databases and Logic Programming*, J. Minker, ed., Morgan Kaufmann Publishers, 1987.

[KS] M.-Y. Kao, and G. E. Shannon, "Local Orientation, Global Order and Planar Topology", *Proc. 21st ACM Symp. on Theory of Computing*, pp. 286-296, 1989.

[LN] R. J. Lipton, and J. F. Naughton, "Estimating the Size of Generalized Transitive Closure", *Proc. 15th Intl. Conf. on Very Large Data Bases*, pp. 165-1172, 1989.

[L] H. Lu, "New Strategies for Computing the Transitive Closure of Database Relations", *Proc. 13th Intl. Conf. on Very Large Databases*, pp. 267-274, 1987.

[MPS] A. Marchetti-Spaccamela, A. Pelaggi, and D. Sacca, "Worst-Case Complexity Analysis of Methods for Logic Query Implementation", *Proc. 6th ACM Symp. on Principles of Database Systems*, pp. 294-301, 1987.

[MC] A. C. McKellar, and E. G. Coffman, Jr., "Organizing Matrices and Matrix Operations for Paged Memory Systems", *C. ACM*, 12(3), pp. 153-165, 1969.

[MW] A. O. Mendelzon, and P. T. Wood, "Finding Regular Simple Paths in Graph Databases", *Proc. 15th Intl. Conf. on Very Large Data Bases*, pp. 185-194, 1989.

[N] J. F. Naughton, "One-Sided Recursions", *Proc. 6th ACM Symp. on Principles of Database Systems*, pp. 340-348, 1987.

[NRSU] J. F. Naughton, R. Ramakrishnan, Y. Sagiv, and J. D. Ullman, "Efficient Evaluation of Right-, Left-, and Multi-Linear Rules", *Proc. ACM SIGMOD Conf. on Management of Data*, pp. 235-242, 1989.

[RS] N. Robertson, and P. D. Seymour, "Graph Minors XIII: The Disjoint Paths Problem", manuscript, 1986.

[SZ1] D. Sacca, and C. Zaniolo, "Magic Counting Methods", *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pp. 49-59, 1987.

[SZ2] D. Sacca, and C. Zaniolo, "The Generalized Counting Method for Recursive Logic Queries", *Theoretical Computer Science*, 62, pp. 187-220, 1988.

[S] C. P. Schnorr, "An Algorithm for Transitive Closure with Linear Expected Time", *SIAM J. Computing*, 7(1), pp. 127-133, 1978.

[SeV] R. Sedgewick, and J. S. Vitter, "Shortest Paths in Euclidean Graphs", *Proc. 25th IEEE Symp. on Foundations of Computer Science*, pp. 417-424, 1984.

[SV] Y. Shiloah, and U. Vishkin, "An $O(\log n)$ parallel connectivity algorithm", *J. Algorithms*, 3(1), pp. 57-63, 1982.

[Sh] O. Shmueli, "Dynamic Cycle Detection", *Information Processing Letters*, 17, pp. 185-188, 1983.

[Si] K. Simon, "An Improved Algorithm for Transitive Closure on Acyclic Digraphs", *Proc. Intl. Conf. on Automata, Languages, and Programming*, pp. 376-386, 1986.

[SS1] S. Sippu and E. Soisalon-Soininen, "An Optimization Strategy for Recursive Queries in Logic Data Bases", *Proc. 4th Intl. Conf. on Data Engineering*, pp. 470-477, 1988.

[SS2] S. Sippu and E. Soisalon-Soininen, "A Generalized Transitive Closure for Relational Queries", *Proc. 7th ACM Symp. on Principles of Database Systems*, pp. 325-332, 1988.

[Sz] R. Szelepcsenyi, "The Mehtod of Forcing for Nondeterministic Automata", *Bulletin EATCS*, 33, pp. 96-100, 1987.

[SU] T. G. Szymanski, and J. D. Ullman, "Evaluating Relational Expressions with Dense and Sparse Arguments", *SIAM J. on Computing*, 6(1), pp. 109-122, 1977.

[T1] R. E. Tarjan, "A Unified Approach to Path Problems", *J. ACM*, 28(3), pp. 577-593, 1981.

[T2] R. E. Tarjan, "Fast Algorithms for Solving Path Problems", *J. ACM*, 28(3), pp.594-614, 1981.

[TY] R. E. Tarján, and M. Yannakakis, "Simple Linear-Time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hepergraphs, and Selectively Reduce

Acyclic Hypergraphs", *SIAM J. on Computing*, 13(3), pp. 566-579, 1984.

[To] M. Tompa, "Two Familiar Transitive Closure Algorithms which Admit no Polynomial Time, Sublinear Space Implementations", *Proc. 12th ACM Symp. on Theory of Computing*, pp. 333-338, 1980.

[U1] J. D. Ullman, *Principles of Database and Knowledge-Base Systems*, Vol. I: *Classical Database Systems*, Computer Science Press, 1988.

[U2] J. D. Ullman, *Principles of Database and Knowledge-Base Systems*, Vol. II: *The New Technologies*, Computer Science Press, 1989.

[UV] J. D. Ullman, and A. Van Gelder, "Parallel Complexity of Logical Query Programs", *Proc. 27th IEEE Symp. on Foundation of Computer Science*, pp. 438-454, 1986.

[UY1] J. D. Ullman, and M. Yannakakis, "The Input/Output Complexity of Transitive Closure", manuscript, 1989.

[UY2] J. D. Ullman, and M. Yannakakis, "High-Probability Parallel Transitive Closure Algorithms", manuscript, 1989.

[VK] P. Valduriez, and S. Khoshafian, "Parallel Evaluation of the Transitive Closure of a Database Relation", *Intl. J. of Parallel Programming*, pp. 19-42, 1988.

[V] L. G. Valiant, "General Context-Free Recognition in Less than Cubic Time", *J. Computer and System Sc.*, 10, pp. 308-315, 1975.

[W1] H. S. Warren, "A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations", *C. ACM*, 18(4), pp. 218-220, 1975.

[W2] S. Warhall, "A Theorem on Boolean Matrices", *J. ACM*, 9(1), pp. 11-12, 1962.

[Z] M. Zloof, "Query-by-example: Operations on the Transitive Closure", IBM RC 5526, 1976.