

Dyn-FO: A Parallel, Dynamic Complexity Class*

Sushant Patnaik[†]

*Computer Science Dept.
University of Massachusetts
Amherst, MA 01003
patnaik@bear.com*

Neil Immerman

*Computer Science Dept.
University of Massachusetts
Amherst, MA 01003
immerman@cs.umass.edu*

Abstract

Traditionally, computational complexity has considered only static problems. Classical Complexity Classes such as NC, P, and NP are defined in terms of the complexity of checking – upon presentation of an entire input – whether the input satisfies a certain property.

For many applications of computers it is more appropriate to model the process as a dynamic one. There is a fairly large object being worked on over a period of time. The object is repeatedly modified by users and computations are performed.

We develop a theory of Dynamic Complexity. We study the new complexity class, Dynamic First-Order Logic (Dyn-FO). This is the set of properties that can be maintained and queried in first-order logic, i.e. relational calculus, on a relational database. We show that many interesting properties are in Dyn-FO including multiplication, graph connectivity, bipartiteness, and the computation of minimum spanning trees. Note that none of these problems is in static FO, and this fact has been used to justify increasing the power of query languages beyond first-order. It is thus striking that these problems are indeed dynamic first-order, and thus, were computable in first-order database languages all along.

We also define “bounded-expansion reductions” which honor dynamic complexity classes. We prove that certain standard complete problems for static complexity classes, such as AGAP for P, remain complete via these new reductions. On the other hand, we prove that other such problems including GAP for NL and 1GAP for L are no longer complete via bounded-expansion reductions. Furthermore, we show that a version of AGAP, called AGAP⁺, is not in Dyn-FO unless all of P is contained in parallel linear time.

1 Introduction

Traditional complexity classes are not completely appropriate for database systems. Unfortunately, appropriate Database Complexity Classes have not yet been defined. This paper makes a step towards correcting this situation.

*Research of both authors supported by NSF grant CCR-9207797.

[†]Current address: Bear Stearns, 245 Park Avenue, New York, NY 10167.

In our view, the main two differences between database complexity and traditional complexity are:

1. Databases are **dynamic**. The work to be done consists of a long sequence of small updates and queries to a large database. Each update and query should be performed very quickly in comparison to the size of the database.
2. Computations on databases are for the most part **disk access bound**. The cost of computing a request is usually tied closely to the number of disk pages that must be read or written to fulfill the request.

Of course, a significant percentage of all uses of computers have the above two features. In this paper we focus on the first issue. Dynamic complexity is quite relevant in most day to day tasks. For example: texting a file, compiling a program, processing a visual scene, performing a complicated calculation in Mathematica, etc. Yet an adequate theory of dynamic complexity is lacking. (Recently, there have been some significant contributions in this direction, e.g. [MSV94]. Note that dynamic complexity is different although somewhat related to on-line complexity which is receiving a great deal of attention lately.)

We will define the complexity class Dyn-FO to be the set of dynamic problems that can be expressed in first-order logic. What this means is that we maintain a database of relevant information so that the action invoked by each insert, delete, and query is first-order expressible. This is very natural in the database setting. In fact, Dyn-FO is really the set of queries that are computable in a traditional first-order query language.

Many interesting queries such as connectivity for undirected graphs are not first-order when considered as static queries. This has led to much work on database query languages such as Datalog that are more expressive than first-order logic.

We show the surprising fact that a wealth of problems, including connectivity, are in Dyn-FO. Thus, considered as dynamic problems – and that is what database problems are – these problems are already first-order computable. The problems we show to be in Dyn-FO include: reachability in undirected graphs, maintaining minimum spanning forest, k -edge connectivity and bipartiteness. All regular languages are shown to be in Dyn-FO. In [P94] it is shown that some NP-complete problems admit Dyn-FO approximation algorithms. Dong and Su [DS93] showed that reachability in directed, acyclic graphs and in function graphs is in Dyn-FO. The static versions of all these problems are not first-order.

Related work on dynamic complexity appears in [MSV94]. In [DST93] first-order incremental evaluation system (FOIES) are defined. This is a monotone version of Dyn-FO. In [TY79], Tarjan and Yao propose a dynamic model whose complexity measure is the number of probes into a data structure and any other computation is free. A $\log n / \log \log n$ lower bound on a dynamic prefix multiplication problem was proved in [FS]. Other lower bounds [M2], [R94] have been proved using these methods.

Other work on dynamic complexity for databases includes the theory of maintaining materialized views upon updates ([J92], [GMS93], [Io85]), and in integrity constraint simplification ([LST87], [N82]). The design of dynamic algorithms is an active field. See, for example, [E*92], [E2*92], [R94], [CT91], [F85], [F91] among many others. There is also a large amount of work in the programming language community on incremental computation, see for example [RR93, LT94].

This paper is organized as follows. In Section 2, we begin with some background on Descriptive Complexity. In Section 3, for any static complexity class \mathcal{C} , we define the corresponding dynamic class, Dyn- \mathcal{C} . The class Dyn-FO is the case we emphasize. In Section 4, we present the above mentioned Dyn-FO algorithms. In Section 5, we describe and investigate reductions honoring dynamic complexity. Finally, we suggest some future directions for the study of dynamic complexity.

2 Descriptive Complexity: Background and Definitions

In this section we recall the notation of Descriptive Complexity. See [I89] for a survey and [IL94] for an extensive study of first-order reductions.

In the development of descriptive complexity it has turned out that “natural” complexity classes have “natural” descriptive characterizations. For example, space corresponds to number of variables; and parallel time is linearly related to quantifier-depth. Sequential time on the other hand does not seem to have a natural descriptive characterization. We like to think of this as yet another indication that sequential time is not a natural notion, simply an artifact of the so-called “Von-Neumann bottleneck”. As another example, the class P, consisting of those problems that can be performed in a polynomial amount of work, has an extremely natural characterization as (FO + LFP) – first-order logic closed under the ability to make inductive definitions.

It is reassuring that our notions of naturalness in logic correspond so nicely with naturalness in complexity theory. In the present work we venture into the terrain of dynamic complexity. What is natural is not yet clear. We use the intuitions gained from the descriptive approach to aid us in our search.

We will code all inputs as finite logical structures, i.e., relational databases. A *vocabulary* $\tau = \langle R_1^{a_1}, \dots, R_r^{a_r}, c_1, \dots, c_s \rangle$ is a tuple of input relation and constant symbols. We define a complexity theoretic *problem* to be any subset $S \subseteq \text{STRUC}[\tau]$ for some τ .

For any vocabulary τ there is a corresponding first-order language $\mathcal{L}(\tau)$ built up from the symbols of τ and the numeric relation symbols, =, \leq , and BIT, and numeric constants *min*, *max* using logical connectives: \wedge, \vee, \neg , variables: x, y, z, \dots , and quantifiers: \forall, \exists . (\leq represents a total ordering on the universe which may be identified with the set $\{0, 1, \dots, n-1\}$. The constants *min*, *max* represent the minimum and maximum elements in this ordering. BIT(x, y) means that when x is coded as a log n bit number, the y^{th} bit of this encoding is a one.)

In static complexity, the entire input structure \mathcal{A} is fixed and we are interested in deciding whether $\mathcal{A} \in S$ for a relevant property, S . In the dynamic case, the structure changes over time. The actions we have in mind are a sequence of insertions and deletions of tuples in the input relations. We will usually think of our dynamic structure, $\mathcal{A} = \langle \{0, 1, \dots, n-1\}, R_1, \dots, R_r, c_1, \dots, c_s \rangle$, as having a fixed size potential universe, $|\mathcal{A}| = \{0, 1, \dots, n-1\}$, and a unary relation R_1 , specifying the elements in the active domain. (We use the notation $\|\mathcal{A}\|$ to denote the cardinality of the universe of \mathcal{A} .) The initial structure of size n for this vocabulary will be taken to be $\mathcal{A}_0^n = \langle \{0, 1, \dots, n-1\}, \{0\}, \emptyset, \dots, \emptyset, 0, \dots, 0 \rangle$, having $R_1 = \{0\}$ indicating that the single element 0 is in the active domain and all other relations are empty.

2.1 First-Order Reductions

Here we briefly describe first-order reductions, a natural way of reducing one problem to another in the descriptive context. First-order reductions are used in Section 5 to build new reductions that honor dynamic complexity. Furthermore, reductions are used in Section 3 as a motivation for the definition of Dynamic Complexity. More information about first-order reductions can be found in [IL94]. “First-order reduction” is another name for a first-order query, i.e. a first-order definable mapping from structures of one vocabulary to structures of another. We give an example and then the formal definition.

Example 2.1 Let graph reachability denote the following problem: given a graph, G , and vertices s, t , determine if there is a path from s to t in G . We shall use GAP to denote graph reachability on directed graphs. Let UGAP be the restriction of the GAP problem to undirected graphs. Let 1GAP be the restriction of the GAP problem in which we only allow deterministic paths, i.e., if the edge (u, v) is on the path, then this must be the unique edge leaving u . Notice that the 1GAP problem is reducible to the UGAP problem as follows: Given a directed graph, G , let G' be the undirected graph that results from G by the following steps:

1. Remove all edges out of t .
2. Remove all multiple edges leaving any vertex.
3. Make each remaining edge undirected.

Observe that there is a deterministic path in G from s to t iff there is a path from s to t in G' .

The following first-order formula $\varphi_{1\text{GAP-UGAP}}$ accomplishes these three steps and is thus a first-order reduction from 1GAP to UGAP. More precisely, the first-order reduction is the expression $I_{1\text{GAP-UGAP}} = \lambda_{xy}(\varphi_{1\text{GAP-UGAP}}, s, t)$ whose meaning is, “Make the new edge relation $\{(x, y) \mid \varphi_{1\text{GAP-UGAP}}\}$, and map s to s and t to t .”

$$\begin{aligned} \alpha(x, y) &\equiv E(x, y) \wedge x \neq t \wedge (\forall z)(E(x, z) \rightarrow z = y) \\ \varphi_{1\text{GAP-UGAP}}(x, y) &\equiv \alpha(x, y) \vee \alpha(y, x) \end{aligned}$$

□

$I_{1\text{GAP-UGAP}}$ is a unary first-order reduction: it leaves the size of the universe unchanged. Now we define k -ary first-order reductions. These map structures with universe size n to structures with universe size n^k .

Definition 2.2 (first-order reductions) Let σ and τ be two vocabularies, with $\tau = \langle R_1^{a_1}, \dots, R_r^{a_r}, c_1, \dots, c_s \rangle$. Let $S \subseteq \text{STRUC}[\sigma]$, $T \subseteq \text{STRUC}[\tau]$ be two problems. Let k be a positive integer. Suppose we are given an r -tuple of formulas $\varphi_i \in \mathcal{L}(\sigma)$, $i = 1, \dots, r$, where the free variables of φ_i are a subset of $\{x_1, \dots, x_{k \cdot a_i}\}$. Finally, suppose we are given an s -tuple, $\overline{t_1}, \dots, \overline{t_s}$, where each $\overline{t_j}$ is a k -tuple of constant symbols from $\mathcal{L}(\sigma)$. Let $I = \lambda_{x_1 \dots x_d} \langle \varphi_1, \dots, \varphi_r, \overline{t_1}, \dots, \overline{t_s} \rangle$ be a tuple of these formulas and constants. (Here $d = \max_i(k a_i)$.) Then I induces a mapping also called I from $\text{STRUC}[\sigma]$ to $\text{STRUC}[\tau]$ as follows. Let $\mathcal{A} \in \text{STRUC}[\sigma]$ and let $n = \|\mathcal{A}\|$.

$$I(\mathcal{A}) = \langle \{0, \dots, n^k - 1\}, R_1, \dots, R_r, \overline{t_1}, \dots, \overline{t_s} \rangle$$

Here each c_j is given by the corresponding k -tuple of constants. The relation R_i is determined by the formula φ_i , for $i = 1, \dots, r$. More precisely, let the function $\langle \cdot, \dots, \cdot \rangle : |\mathcal{A}|^k \rightarrow |I(\mathcal{A})|$ be given by

$$\langle u_1, u_2, \dots, u_k \rangle = u_k + u_{k-1}n + \dots + u_1n^{k-1}$$

Then,

$$R_i = \{(\langle u_1, \dots, u_k \rangle, \dots, \langle u_{1+k(a_i-1)}, \dots, u_{ka_i} \rangle) \mid \mathcal{A} \models \varphi_i(u_1, \dots, u_{ka_i})\}$$

Suppose that I is a many-one reduction from S to T , i.e., for all \mathcal{A} in $\text{STRUC}[\sigma]$,

$$\mathcal{A} \in S \iff I(\mathcal{A}) \in T$$

Then we say that I is a k -ary *first-order reduction* of S to T . □

3 Dynamic Complexity Classes

We think of an implementation of a problem $S \subseteq \text{STRUC}[\sigma]$ as a mapping, I , from $\text{STRUC}[\sigma]$ to $\text{STRUC}[\tau]$ where $T \subseteq \text{STRUC}[\tau]$ is an easier problem. The map I should be a many-one reduction from S to T meaning that any structure \mathcal{A} has the property S iff $I(\mathcal{A})$ has the property T . (Actually, in our definition below, the mapping I will map a sequence of inserts, deletes, and changes \bar{r} to a structure. In the interesting special case when $I(\bar{r})$ depends only on the corresponding structure \mathcal{A} and not which sequence of requests created it, we call I *memoryless*.)

The idea is that $I(\mathcal{A})$ is the data structure – our internal representation of \mathcal{A} . We are thinking and talking about a structure $\mathcal{A} \in \text{STRUC}[\sigma]$, but the structure that we actually have in memory and disk and are manipulating is $I(\mathcal{A}) \in \text{STRUC}[\tau]$. In this way, each request to \mathcal{A} is interpreted as a corresponding series of actions on $I(\mathcal{A})$. The fact that I is a many-one reduction insures that the query asking whether $\mathcal{A} \in S$ can be answered according as whether $I(\mathcal{A}) \in T$. In order for this to be useful, T must be a problem of low dynamic complexity.

Next we give the formal definition of dynamic complexity classes. The issue is that the structure $I(\mathcal{A})$ can be updated efficiently in response to any request to \mathcal{A} . In particular, if $T \in \text{FO}$ and all such requests are first-order computable, then $S \in \text{Dyn-FO}$.

3.1 Definition of Dyn- \mathcal{C}

For any complexity class \mathcal{C} we define its dynamic version, $\text{Dyn-}\mathcal{C}$, as follows. Let $\sigma = \langle R_1^{a_1}, \dots, R_r^{a_r}, c_1, \dots, c_s \rangle$ be a vocabulary and let $S \subseteq \text{STRUC}[\sigma]$ be any problem. Let

$$\mathcal{R}_{n,\sigma} = \{\text{ins}(i, \bar{a}), \text{del}(i, \bar{a}), \text{set}(j, a) \mid 1 \leq i \leq r, \bar{a} \in \{0, \dots, n-1\}^{a_i}, 1 \leq j \leq s\} \quad (3.1)$$

be the set of possible requests to insert tuple \bar{a} into the relation R_i , delete tuple \bar{a} from relation R_i , or set constant c_j to a . Let $\mathcal{R}_{n,\sigma}^*$ be the set of finite sequences from $\mathcal{R}_{n,\sigma}$.

Let $\text{eval}_{n,\sigma} : \mathcal{R}_{n,\sigma}^* \rightarrow \text{STRUC}[\sigma]$ be the naturally defined evaluation of a sequence of requests, initialized by $\text{eval}_{n,\sigma}(\emptyset) = \mathcal{A}_0^n$.

Define, $S \in \text{Dyn-}\mathcal{C}$ iff there exists another problem $T \subseteq \text{STRUC}[\tau]$ such that $T \in \mathcal{C}$ and there exist maps,

$$f_n : \mathcal{R}_{n,\sigma}^* \rightarrow \text{STRUC}[\tau]; \quad g_n : \text{STRUC}[\tau] \times \mathcal{R}_{n,\sigma} \rightarrow \text{STRUC}[\tau]$$

satisfying the following properties.

1. For all $\bar{r} \in \mathcal{R}_{n,\sigma}^*$, $(\text{eval}_{n,\sigma}(\bar{r}) \in S) \Leftrightarrow (f_n(\bar{r}) \in T)$
2. For all $s \in \mathcal{R}_{n,\sigma}$, and $\bar{r} \in \mathcal{R}_{n,\sigma}^*$, $f_n(\text{eval}_{n,\sigma}(\bar{r}s)) = g_n(f_n(\text{eval}_{n,\sigma}(\bar{r})), s)$
3. $\|f_n(\bar{r})\| = \|\text{eval}_{n,\sigma}(\bar{r})\|^{O(1)}$ ¹
4. The functions g_n and the initial structure $f_n(\emptyset)$ are uniformly computable in complexity \mathcal{C} (as a function of n).

Note that the condition (4) says that each single request $s \in \mathcal{R}_{n,\sigma}$ can be responded to quickly, i.e., with complexity \mathcal{C} . According to the definition of $\mathcal{R}_{n,\sigma}$, the kind of updates allowed are single inserts or deletes of tuples, and assignments to constants. Of course it would be interesting to consider other possible sets of updates.

We will say that the above map f is *memoryless* if the value of $f(\bar{r})$ depends only on $\text{eval}_{n,\sigma}(\bar{r})$.

In the above, if no deletes are allowed then we get the class $\text{Dyn}_s\text{-}\mathcal{C}$, the semi-dynamic version of \mathcal{C} . One can also consider amortized versions of these two classes. Furthermore, there are some cases where we would like extra, but polynomial, precomputation to compute the initial structure $f(\emptyset)$. If we relax condition (4) in this way, then the resulting class is called $\text{Dyn-}\mathcal{C}^+$ – $\text{Dyn-}\mathcal{C}$ with polynomial precomputation.

We have thus defined the dynamic complexity classes $\text{Dyn-}\mathcal{C}$ for any static class, \mathcal{C} . Two particularly interesting examples are Dyn-FO and $\text{Dyn-TIME}[t(n)]$ for $t(n)$ less than n , where the latter is the set of problems computable dynamically on a RAM (with word size $O(\log n)$) in time $t(n)$.

Example 3.2 Consider the simple boolean query: Parity, which is true iff the input binary string has an odd number of one's. This is well known not to be in static FO [A83, FSS84]. The dynamic algorithm for Parity maintains a bit b which is toggled after any change to the string. We also remember the input string so that we can tell if a request has actually changed the string.

The vocabulary of the Parity problem is $\sigma = \langle M \rangle$ consisting of a single monadic relation symbol. Let \mathcal{A}_w be the structure coding the binary string w . Then $\mathcal{A}_w \models M(i)$ iff the i^{th} bit of w is a one. Let $\tau = \langle M, b \rangle$ where M is monadic and b is a boolean constant symbol. The problem T , obviously in FO, is given as follows:

$$T = \{ \mathcal{A} \in \text{STRUC}[\tau] \mid \mathcal{A} \models b \}$$

¹This expects that the complexity class \mathcal{C} is closed under polynomial increases in the input size. For more restricted classes \mathcal{C} , such as linear time, we insist that $\|f_n(\bar{r})\| = O(\|\text{eval}_{n,\sigma}(\bar{r})\|)$.

The initial data structure $f_n(\emptyset) = \langle \{0, 1, \dots, n-1\}, \emptyset, false \rangle$ consists of a string of all 0's, with the boolean b initialized to false.

To show that Parity is in Dyn-FO we need to give the first-order formulas that compute $g_n(\mathcal{B}, s)$ for any request $s \in \mathcal{R}_{n,\sigma}$. The cases are the setting of a bit of w to 0 or 1. Let M, b denote the relations in the data structure \mathcal{B} before the request, and M', b' are their values afterwards.

ins(a, M):

$$\begin{aligned} M' &\equiv M(x) \vee x = a \\ b' &\equiv (b \wedge M(a)) \vee (\neg b \wedge \neg M(a)) \end{aligned}$$

del(a, M):

$$\begin{aligned} M' &\equiv M(x) \wedge x \neq a \\ b' &\equiv (b \wedge \neg M(a)) \vee (\neg b \wedge M(a)) \end{aligned}$$

□

Note 3.3 In the definition of Dyn- \mathcal{C} we assumed that our problem S had only the basic requests $\mathcal{R}_{n,\sigma}$ (Equation 3.1) defined. The definition of Dyn- \mathcal{C} remains valid when we allow an arbitrary set of operations $\mathcal{O}_{n,\sigma}$ to be performed on the input structures. It thus makes sense to ask whether any data structure with a given set of operations is in Dyn- \mathcal{C} .

4 Problems in Dyn-FO

It is well known that the graph reachability problem is not first-order expressible and this has often been used as a justification for using database query languages more powerful than FO [CH82]. Thus, the following two theorems are striking.

Theorem 4.1 *UGAP is in Dyn-FO.*

Proof We maintain a spanning forest of the underlying graph via relations, $F(x, y)$ and $PV(x, y, u)$ and the input relation, E . $F(x, y)$ means that the edge (x, y) is in the current spanning forest. $PV(x, y, u)$ means that there is a (unique) path in the forest from x to y via vertex u . The vertex, u , may be one of the endpoints. So, for example, if $F(x, y)$ is true, then so are $PV(x, y, x)$ and $PV(x, y, y)$. We maintain the undirected nature of the graph by interpreting $\text{insert}(E, a, b)$ or $\text{delete}(E, a, b)$ to do the operation on both (a, b) and (b, a) .

Insert(E, a, b): We denote the updated relations as E', F' and PV' . In the sequel, we shall use $P(x, y)$ to abbreviate $(x = y \vee PV(x, y, x))$, and $\text{Eq}(x, y, c, d)$ to abbreviate the formula,

$$((x = c \wedge y = d) \vee (x = d \wedge y = c)).$$

Maintaining the input edge relation is trivial:

$$E'(x, y) \equiv E(x, y) \vee \text{Eq}(x, y, a, b)$$

The edges in the forest remain unchanged if vertices a and b were already in the same connected component. Otherwise, the only new forest edge is (a, b) .

$$F'(x, y) \equiv F(x, y) \vee (\text{Eq}(x, y, a, b) \wedge \neg P(a, b))$$

Now all that remains is to compute PV' . The latter changes iff edge (a, b) connects two formerly disconnected trees. In this case, all new tuples (x, y, z) have x coming from one of the trees containing a and b , and y coming from the other.

$$\begin{aligned} PV'(x, y, z) \equiv & PV(x, y, z) \vee (\exists uv)[\text{Eq}(u, v, a, b) \wedge P(x, u) \wedge P(v, y) \\ & \wedge (PV(x, u, z) \vee PV(v, y, z))] \end{aligned}$$

Delete(\mathbf{E}, a, b): If edge (a, b) is not in the forest ($\neg F(a, b)$), then the updated relations are unchanged, except that $E'(a, b)$ is set to false. Otherwise, we first identify the vertices of the two trees in the forest created by the deletion, and then we pick an edge, say e , out of all the edges (if any) that run between the two trees and *insert* e into the forest, updating the relations, PV and F , appropriately.

We define a temporary relation T to denote the PV relation after (a, b) is deleted, before the new edge, e , is inserted.

$$T(x, y, z) \equiv PV(x, y, z) \wedge \neg(PV(x, y, a) \wedge PV(x, y, b))$$

Using T , we then pick the new edge that must be added to the spanning forest. $\text{New}(x, y)$ is true if and only if edge (x, y) is the minimum² edge that connects the two disconnected components:

$$\begin{aligned} \text{New}(x, y) \equiv & E(x, y) \wedge T(a, x, a) \wedge T(b, y, b) \quad \wedge \\ & (\forall uv)[(E(u, v) \wedge T(a, u, a) \wedge T(b, v, b)) \rightarrow (x < u \vee (x = u \wedge y \leq v))] \end{aligned}$$

E' , F' and PV' are then defined as follows:

$$E'(x, y) \equiv E(x, y) \wedge \neg \text{Eq}(x, y, a, b)$$

We remove (a, b) from the forest and add the new edge.

$$F'(x, y) \equiv (F(x, y) \wedge \neg \text{Eq}(x, y, a, b)) \vee \text{New}(x, y) \vee \text{New}(y, x)$$

The paths in the forest, from x to y via z , that did not pass through a and b , are valid. Also, new paths have to be added as a result of the insertion of a new edge in the forest.

$$\begin{aligned} PV'(x, y, z) \equiv & T(x, y, z) \vee [(\exists u, v)(\text{New}(u, v) \vee \text{New}(v, u)) \wedge T(x, u, x) \\ & \wedge T(y, v, y) \wedge (T(x, u, z) \vee T(y, v, z))] \end{aligned}$$

²Note that this uses an ordering on the vertices. If no such ordering is given, then we can order edges by their order of insertion. In the presence of an ordering relation, we can modify the construction to always maintain the minimum spanning forest as in Theorem 4.4. This is then memoryless.

□

In [PI94] we asked if the proof of Theorem 4.1 could be carried out using auxiliary relations of arity two instead of three. Quite recently Dong and Su have shown that the answer is yes [DS95]. They show that the arity three construction of PV can be replaced by a directed version of F and its transitive closure. They also use Ehrenfeucht-Fraïssé games to show that arity one does not suffice. We next give a new Dyn-FO algorithm for the following result of Dong and Su. By GAP (acyclic) we mean the GAP problem restricted to queries in which the input graph is acyclic during its entire history.

Theorem 4.2 ([DS93]) *1GAP and GAP (acyclic) are in Dyn-FO.*

Proof That 1GAP is in Dyn-FO follows from Theorem 4.1 together with the fact that 1GAP is reducible to UGAP (Example 2.1). We defer the details of this until the proof of Proposition 5.3.

For the GAP (acyclic) case, the inserts are assumed to always preserve acyclicity. We maintain the path relation $P(x, y)$ which means that there is a path from x to y in the graph.

Insert(\mathbf{E}, a, b):

$$P'(x, y) \equiv P(x, y) \vee (P(x, a) \wedge P(b, y))$$

Delete(\mathbf{E}, a, b):

$$P'(x, y) \equiv P(x, y) \wedge [\neg P(x, a) \vee \neg P(b, y) \vee (\exists uv)(P(x, u) \wedge P(u, a) \wedge E(u, v) \wedge \neg P(v, a) \wedge P(v, y) \wedge (v \neq b \vee u \neq a))]$$

In the case where there is a path from x to y using the edge (a, b) , consider any path not using this edge. Let u be the last vertex along this path from which a is reachable. Note that $u \neq y$ because the graph was acyclic before the deletion of edge (a, b) . Thus, the edge (u, v) described in the above formula must exist and acyclicity insures that the path $x \rightarrow u \rightarrow v \rightarrow y$ does not involve the edge (a, b) . □

For G , a directed acyclic graph, the *Transitive Reduction*, $\text{TR}(G)$, is the minimal subgraph of G having the same transitive closure as G .

Corollary 4.3 *Transitive Reduction for directed acyclic graphs is in memoryless Dyn-FO.*

Proof We maintain the path relation, P , in a way that is quite similar to the proof of Theorem 4.2.

Insert($\mathbf{E}, (a, b)$): If $P(a, b)$ already holds, then there is no change. Otherwise, we may have to remove some edges from TR.

$$\text{TR}'(x, y) \equiv (\neg P(a, b) \wedge x = a \wedge y = b) \vee [\text{TR}(x, y) \wedge \neg(P(x, a) \wedge P(b, y))]$$

Delete($\mathbf{E}, (a, b)$): We have to determine the new edges that might be added in TR. For an edge (x, y) , $\text{New}(x, y)$ holds if there was a path from x to y via (a, b) and no path of length greater than one remains in the graph when (a, b) is deleted.

$$\begin{aligned} \text{New}(x, y) \equiv & E(x, y) \wedge \neg \text{TR}(x, y) \wedge P(x, a) \wedge P(b, y) \\ & \wedge (\forall uv) \neg (P(x, u) \wedge P(u, a) \wedge E(u, v) \wedge \neg P(v, a) \wedge P(v, y) \wedge (v \neq b \vee u \neq a)) \end{aligned}$$

$$\text{TR}'(x, y) \equiv (\text{TR}(x, y) \wedge \neg(x = a \wedge y = b)) \vee \text{New}(x, y) \quad \square$$

Theorem 4.4 *Minimum Spanning Forests can be computed in Dyn-FO.*

Proof The general idea is to maintain the forest edges and non-forest edges dynamically and to maintain the relations $\text{PV}(x, y, e)$ and $\text{F}(x, y)$ as in the case of UGAP. Let $W(a, b)$ denote the weight of edge (a, b) . The difference from UGAP is that we have to maintain the minimum weighted forest. That changes our update procedures in the following way.

Deletion of the edge (a, b) is handled as follows. We determine using PV all the vertices that can be reached from a in the tree and all those that can be reached from b . These give the vertices in the two trees that the original tree splits into. Then, instead of choosing the lexicographically first non-forest edge that reconnects the two pieces, we choose the *minimum weight* such edge, and insert it. If there is more than one such minimum edge, then we break the tie with the ordering. PV is updated accordingly to reflect the merging of two disconnected trees into one.

When the edge (a, b) is inserted, we determine if there exists a path between a and b . If there is no path, then (a, b) merges two trees into one, and PV is updated as before for UGAP. Otherwise, using PV, we can determine the forest-edges that appear in the unique path in the forest between b and a , and check to see if the weight of the new edge, (a, b) , is less than the weight of any of these edges. If not, then (a, b) is not a forest edge and nothing changes. Otherwise, let (c, d) be the maximum weight edge on the path from a to b . We make $\text{F}'(c, d)$ false and $\text{F}'(a, b)$ true and update PV accordingly. It is easy to see that if the weights are all distinct, or in the presence of an ordering on the edges, this construction is memoryless. \square

We next show that Dyn-FO algorithms exist for Bipartiteness, Edge Connectivity, Lowest Common Ancestors in directed forests, and Maximal Matching in bounded degree graphs. This last problem has no known sub-linear-time fully dynamic algorithm.

Theorem 4.5 *Dyn-FO algorithms exist for the following problems:*

1. *Bipartiteness,*
2. *k-Edge Connectivity, for a fixed constant k*
3. *Maximal Matching in undirected graphs,*
4. *Lowest Common Ancestor in directed forests.*

Proof

1. Bipartiteness: We maintain bipartiteness in undirected graphs by maintaining relations $\text{PV}(x, y, z)$ and $\text{F}(x, y)$, as for UGAP, and also, $\text{Odd}(x, y)$, which means that there

exists a path of odd length from x to y in the spanning forest. The graph is bipartite iff $(\forall xy)E(x, y) \rightarrow \text{Odd}(x, y)$. We show how to update Odd in Dyn-FO.

Insert(E, a, b): If the new edge, (a, b) becomes a forest-edge, we determine for all newly connected vertices x and y whether the new path is odd. If (a, b) does not become a forest edge, then Odd is unchanged.

$$\begin{aligned} \text{Odd}' \equiv & \text{Odd}(x, y) \vee \left[\neg \text{PV}(a, b, a) \wedge (\exists uv) \text{Eq}(u, v, a, b) \wedge \text{PV}(x, u, x) \wedge \text{PV}(y, v, y) \right. \\ & \left. \wedge ((\text{Odd}(x, u) \wedge \text{Odd}(y, v)) \vee (\neg \text{Odd}(x, u) \wedge \neg \text{Odd}(y, v))) \right] \end{aligned}$$

Delete(E, a, b): If (a, b) is a non-forest edge, Odd is unchanged. Otherwise, for all vertices x, y , which are in the two disconnected trees that result from deletion of (a, b) , make $\text{Odd}(x, y)$ and $\text{PV}(x, y, z)$ false. Then, we select some edge (if any) that spans the disconnected components and insert it in and update Odd and PV exactly as for the insertion case.

2. k -Edge Connectivity: As before, we maintain the relations, E,F and PV. Insertions and deletions are handled as for UGAP. The query is handled as follows. Since k is constant, we universally quantify over k edges, say, $(x_1, y_1), \dots, (x_k, y_k)$, and then, for every pair of vertices, x and y , check for a path between x and y in the graph that is obtained after deletion of edges, $(x_1, y_1), \dots, (x_k, y_k)$, by composing the Dyn-FO formula (for a single deletion) k times.

3. Maximal Matching: We maintain a maximal matching in Dyn-FO by maintaining, under insertions and deletions of edges, a relation, $\text{Match}(x, y)$ which means that the edge (x, y) is in the matching. Initially, for the empty graph, $\text{Match}(x, y)$ is false for all x and y . As usual, all relations are symmetric. We shall use $\text{MP}(x)$ to abbreviate the formula $(\exists z)\text{Match}(x, z)$.

Insert(E, a, b): The matching remains unchanged except that the edge (a, b) is checked to see whether it can be added.

$$\text{Match}'(x, y) \equiv \text{Match}(x, y) \vee (\text{Eq}(x, y, a, b) \wedge \neg \text{MP}(a) \wedge \neg \text{MP}(b))$$

Delete(E, a, b): If (a, b) is not in the matching, then Match is unchanged. Otherwise, we remove (a, b) from the matching. We pick the minimum unmatched vertex adjacent to a , if any, and match it with a . Then we do the same for b .

4. Lowest Common Ancestor: In a directed forest we maintain the relation P exactly as in Theorem 4.2. Vertex a is the lowest common ancestor of x and y iff

$$P(a, x) \wedge P(a, y) \wedge (\forall z)((P(z, x) \wedge P(z, y)) \rightarrow P(z, a)) \quad \square$$

We have shown that Dyn-FO contains some interesting problems that are not static first-order. In particular, Dyn-FO contains natural complete problems for L and NL. As we will see in the next section, these problems do not remain complete via the reductions that honor dynamic complexity. Thus it does not necessarily follow that Dyn-FO contains the classes L or NL. We can prove the following:

Theorem 4.6 *Every regular language is in Dyn-FO.*

Proof We are given a deterministic finite automaton $D = \langle \Sigma, Q, \delta, s, F \rangle$ and an input size n . Let $\Sigma = \{\sigma_1, \sigma_2, \dots, \sigma_t\}$. Let $w \in \Sigma^*$ be a string length $n = |w|$. Let \mathcal{A}_w be the logical structure coding w :

$$\mathcal{A}_w = \langle \{0, 1, \dots, n-1\}, R_1, \dots, R_t \rangle$$

The universe of \mathcal{A}_w consists of the positions of the n letters in w . $\mathcal{A}_w \models R_i(j)$ iff the j^{th} character of w is σ_i . With this encoding, the allowable operations are to insert or delete a character into any position in the string. (As usual, there is a numeric predicate \leq giving the usual total ordering on $\{0, 1, 2, \dots, n-1\}$.) The deletion of a character at position i , is equivalent to setting the character at that position to the empty string.

A natural dynamic algorithm for the regular language $L(D)$, is to maintain a complete binary tree with leaves at the input positions $0, 1, \dots, n-1$. At the leaf $l(i)$ at position i we store the transition function of D on reading input symbol σ_i . That is we store a table for $f_{l(i)} = \delta(\cdot, \sigma_i) : Q \rightarrow Q$. At each internal node of the tree we store the composition of the functions of its two children. Thus, at every node v of the tree, we have stored the transition function $f_v = \delta^*(\cdot, w_v)$ where w_v is the subword of w that is sitting below v 's subtree. In particular, the current string is in $L(D)$ iff $f_r(s) \in F$, where f_r is the mapping stored at the root.

Since D is a finite state machine, these mappings consist of a bounded number of bits each. When we change the symbol σ_i , this affects the $\log n$ nodes on the path from $l(i)$ to r . We can thus guess the $O(\log n)$ bits that change in the tree by existentially quantifying $O(1)$ variables. (Remember that the value of a variable is a number between 0 and $n-1$, i.e. $\log n$ bits, and we have the BIT predicate available for decoding.) We then universally assert that each of the $\log n$ positions has been updated correctly. \square

We conclude this section with two other low-level problems that are in Dyn-FO:

Proposition 4.7 *Multiplication is in Dyn-FO.*

Proof Given two n -bit numbers, x, y , their addition can be expressed in $\text{FO} \subseteq \text{Dyn-FO}$. We maintain the product in a bit array, P . Suppose the update operation is $\text{Change}(x, i, b)$. ($\text{Change}(y, i, b)$ is analogous.) There are two cases:

If the bit is changed from 0 to 1, then P' is given by shifting y by i bits to the right and then adding it to P . It is easily accomplished by a first-order formula.

If the bit is changed from 1 to 0, then P' is given by shifting y by i bits to the right and then adding the 2's complement of the resulting number to P . Again this is easily accomplished by a first-order formula. \square

Proposition 4.8 *For any constant, k , D^k , the Dyck language on k parentheses is in Dyn-FO.*

Proof This is similar to the proof in [BC89] that D^k is in ThC^0 – the class of problems accepted by bounded-depth, polynomial-size threshold circuits. D^k can be parsed using the

level trick: assign a level to each parenthesis starting at one and ignoring the differences in parenthesis type. The *level* of a parenthesis equals the number of left parentheses to its left (including it) minus the number of right parentheses strictly to its left. A right parenthesis matches a left one if it is the closest parenthesis to the right on the same level. A string is in D^k iff all parentheses have a positive level and each left parenthesis has a matching right parenthesis of the same type.

The level of each position can be maintained in Dyn-FO. For example, the insertion of a left parenthesis at position p causes a one to be added to the level of each position $q \geq p$. Once the levels are maintained, the fact that every left parenthesis has a matching right one of the same type is first-order expressible. \square

5 Dynamic Reductions

Now we define appropriate reductions for comparing dynamic complexity classes. For these classes first order reductions are too powerful. We restrict them by imposing the following *expansion* property, cf. [MSV94] for a similar restriction.

Definition 5.1 *Bounded expansion, first-order reductions* (bfo) are first-order reductions (Definition 2.2) such that each tuple in a relation and each constant of the input structure affects at most a constant number of tuples and constants in the output structure. This dependency is oblivious, i.e., only depending on the numeric predicates: $\leq, =, \text{BIT}$ and not on the input predicates. (This is similar to the definition of first-order projections [IL94] in which each output bit must depend on at most one input bit.) Furthermore, a bfo reduction is required to map the initial structure, $\mathcal{A}_{0,n}$, to a structure with only a bounded number of tuples present. If this condition is relaxed we get bounded expansion, first-order reductions with precomputation (bfo⁺). If S is reducible to T via bounded-expansion, first-order reductions (with precomputation), we write $S \leq_{\text{bfo}} T$ ($S \leq_{\text{bfo}^+} T$). \square

As an example consider the first-order reduction $I_{\text{1GAP-UGAP}}$ from Example 2.1. Observe that this is bounded expansion because each insertion or deletion of an edge (a, b) from the graph G can cause at most two edges to be inserted or deleted in $G' = I_{\text{1GAP-UGAP}}(G)$.

Since the composition of bfo or bfo⁺ reductions are bfo, bfo⁺, respectively, it follows that:

Proposition 5.2 *The relations \leq_{bfo} and \leq_{bfo^+} are transitive.*

As desired, bfo reductions preserve dynamic complexity:

Proposition 5.3 *If $T \in \text{Dyn-FO}$ and $S \leq_{\text{bfo}} T$, then $S \in \text{Dyn-FO}$.*

Proof We are given a bfo reduction, I , from S to T . Any change to an input, \mathcal{A} , for S corresponds to a bounded number of changes to $I(\mathcal{A})$. Given a request r to \mathcal{A} , we know the number, k , of possibly affected tuples and constants in $I(\mathcal{A})$. Since I is first-order, we can existentially quantify all the changed tuples and constants. Then we can respond to the k or fewer changes using the Dyn-FO algorithm for T . Since I is a many-one reduction, every answer of a query to T will also be the correct answer for the given query to S . \square

A bfo^+ reduction allows $I(\mathcal{A}_0^n)$ to be quite different from the standard initial structure, \mathcal{B}_0^m . This corresponds to some precomputation to handle $I(\mathcal{A}_0^n)$ as an initial structure. Thus we have:

Corollary 5.4 *If $T \in \text{Dyn-FO}^+$ and $S \leq_{\text{bfo}^+} T$, then $S \in \text{Dyn-FO}^+$.*

Bounded-expansion reductions with precomputation are an appropriate reduction for comparing the dynamic complexity of problems. We also note that most natural reductions for P-complete and NP-complete problems are either bounded expansion, or can be easily modified to be so (see [P94], [MI94]). We give one such example here:

Proposition 5.5 *AGAP and CVAL are complete for P via bfo^+ reductions.*

Proof AGAP is the reachability problem for alternating graphs. It is equivalent to CVAL – the circuit value problem. In [I81], it is shown that AGAP is complete for $\text{ASPACE}[\log n]$ via first-order reductions. Recall that $\text{ASPACE}[\log n] = \text{P}$. The proof depends on the fact that AGAP is the natural complete problem for $\text{ASPACE}[\log n]$. An alternating machine can put off looking at its input until the last step of its computation. Thus, each input bit is copied only once and the first-order reductions from [I81] become bounded expansion. \square

The reason that Proposition 5.5 requires bfo^+ reductions is that $I(\mathcal{A}_0^n)$ contains more than a bounded number of edges and vertices marked “ \forall ” (or equivalently for CVAL, more than a bounded number of wires and nodes marked “and”). In fact, $I(\mathcal{A}_0^n)$ represents the entire computation tree of an $\text{ASPACE}[\log n]$ machine on input all zero’s. Clearly there is a related problem which we call AGAP^+ which knows by heart the first-order describable graph $I(\mathcal{A}_0^n)$ and starts its input there instead of at \mathcal{B}_0^m . In general we have

Proposition 5.6 *For any problem S that is hard for a complexity class \mathcal{C} via bfo^+ reductions, there is a related problem S^+ that is hard for \mathcal{C} via bfo reductions. In particular, AGAP^+ is complete for P via bfo reductions.*

The following corollary indicates that AGAP^+ is probably not in Dyn-FO . Let $\text{CRAM}[t(n)]$ be the set of problems computable by uniform CRCW-PRAMS using polynomially much hardware. It is known that $\text{FO} = \text{CRAM}[1]$ [I89b]. Let $\text{CRAM}^+[t(n)]$ be $\text{CRAM}[t(n)]$ with polynomial precomputation. Then

Corollary 5.7 *$\text{Dyn-FO} \subseteq \text{CRAM}[n]$ and $\text{Dyn-FO}^+ \subseteq \text{CRAM}^+[n]$.*

If $\text{AGAP} \in \text{Dyn-FO}^+$, then $\text{P} = \text{Dyn-FO}^+$, and thus $\text{P} = \text{CRAM}^+[n]$.

If $\text{AGAP}^+ \in \text{Dyn-FO}$, then $\text{P} = \text{Dyn-FO}$, and thus $\text{P} = \text{CRAM}[n]$.

Things are more complicated for the lower complexity classes L and NL. The reason is that bfo^+ reductions preserve the number of times that the input is read. This does not matter for P and larger classes as we may simply copy the input, reading it once. For appropriate complexity classes, \mathcal{C} , define $\text{read-}O(1)\text{-times } \mathcal{C}$ to be the set of problems $S \in \mathcal{C}$ such that there exists a constant k , such that any accepting computation on an input w for S reads no bit of w more than k times.

Proposition 5.8 *Let S, T be problems. Let \mathcal{C} be any complexity class that is closed under first-order reductions. If $S \leq_{\text{bfo}^+} T$ and $T \in \text{read-}O(1)\text{-times } \mathcal{C}$ Then $S \in \text{read-}O(1)\text{-times } \mathcal{C}$.*

Proof To test if input \mathcal{A} is in S , we run the $\text{read-}O(1)$ times \mathcal{C} algorithm to test if $I(\mathcal{A}) \in T$. Each time the algorithm needs to read a bit b of $I(\mathcal{A})$, it examines the relevant bits of \mathcal{A} . By the definition of bfo^+ reductions, these can be determined in a first-order way. Furthermore, the bounded-expansion property insures that each bit of \mathcal{A} is thus queried only a bounded number of times. \square

Now, clearly 1GAP is in read-once L and GAP is in read-once NL , since no accepting computation need check the existence of a particular edge more than once. Thus if these problems remained complete for their respective classes via bfo^+ reductions, then it would follow that L would be equal to $\text{read-}O(1)\text{-times L}$ and NL would be equal to $\text{read-}O(1)\text{-times NL}$. These latter facts are false:

Fact 5.9 ([BRS91]) *There is a problem in L that is not in $\text{read-}O(1)\text{-times NL}$.*

It follows that

Corollary 5.10 *1GAP is not complete for L via bfo^+ reductions and GAP is not complete for NL via bfo^+ reductions.*

The standard proof that 1GAP and GAP are complete for L and NL map each input w to a computation graph of a fixed Turing machine on input w . The reason this mapping is not bounded expansion is that many different nodes in the computation tree might read a particular bit of w and thus have an edge to a next node according to the value of this bit. A variant of GAP called COLOR-GAP is invented in [MSV94] to finesse this difficulty. An input to COLOR-GAP consists of a directed graph with outdegree at most two with the outgoing edges labelled zero and one. There is a given partition of the vertices $V = V_0 \cup V_1 \cup \dots \cup V_r$. Furthermore, there is a bit-vector $C[1 \dots r]$. C tells us for each vertex $v \notin V_0$ whether to follow the one-edge or the zero-edge out of v . Thus, by setting a single bit $C[i]$, we are deciding on edges out of all the vertices in V_i . By letting V_i be the set of nodes that would query bit i , we have transformed the GAP problem to one where the standard reduction is now bounded expansion:

Fact 5.11 ([MSV94]) *COLOR-GAP is complete for NL via bfo^+ reductions.*

It is not hard to see that one can “colorize” most complete problems via first-order reductions to get a version that is complete via bfo^+ reductions. Here are two examples: Let COLOR-1GAP be the subproblem of COLOR-GAP in which V_0 is empty and so with the color vector C given, every vertex has out-degree one. Let COLOR- $\amalg(S_5)$ be the colored version of the problem $\amalg(S_5)$ – iterated multiplication of elements of the symmetric group on five objects [IL94, B89]. This means that for each position there is a pair of group elements, σ_0 and σ_1 . The positions are partitioned into classes P_1, \dots, P_r and bit $C[i]$ tells us whether to pick σ_0 or σ_1 for all the positions in class P_i .

Corollary 5.12 *COLOR-1GAP is complete for L via bfo^+ reductions, and COLOR- $\Pi(S_5)$ is complete for NC^1 via bfo^+ reductions.*

Another idea from [MSV94] indicates that dynamic complexity classes may not be as robust under changes to the form of the input as static classes are:

Definition 5.13 ([MSV94]) For any problem S , define the padded form of S as follows:

$$PAD(S) = \{w_1, w_2, \dots, w_n \mid n \in \mathbf{N}, |w_1| = n, w_1 = w_2 = \dots = w_n, w_1 \in S\} \quad \square$$

Clearly, $PAD(S)$ is computationally equivalent to S . However, changing a single bit of the input to S requires n changes to the input to $PAD(S)$. Thus a Dyn-FO algorithm for $PAD(AGAP)$ has n first-order steps to respond to any real change to the input to S . Recall that $AGAP$ is complete for P via first-order reductions. Thus, so is $PAD(AGAP)$. Also, it is easy to see that $AGAP$ is in $FO[n]$ [I87]. It follows that a complete problem for P is in Dyn-FO:

Theorem 5.14 *$PAD(AGAP)$ is in Dyn-FO.*

6 Conclusions

We have defined dynamic complexity classes, and their reductions. In particular, we have begun an investigation of the rich dynamic complexity class Dyn-FO. Much work remains to be done. We point toward a few of the many directions.

1. Does Dyn-FO contain any of the complexity classes: ThC^0 , NC^1 , L, NL? We conjecture that $ThC^0 \subseteq Dyn-FO$ and that $AGAP \notin Dyn-FO$ and thus $Dyn-FO \subsetneq P$.
2. Is GAP in Dyn-FO? We conjecture that it is.
3. Further work needs to be done concerning the role and value of precomputation in the dynamic setting.
4. In this paper, we have only considered problems that have the basic set of requests as defined in Equation 3.1. As pointed out in Note 3.3, one can consider different sets of possible requests. For example, an interesting subset of Dyn-FO results if we require first-order response to any first-order change to the input. This and other changes to the request set should be investigated.
5. Find natural complete problems for Dynamic Classes.
6. Find natural descriptive languages that capture Dynamic Classes.
7. Determine an automatic way to look at a query and figure out what to save in a data structure so that the query will have low dynamic complexity.

Acknowledgements: Thanks to Ron Fagin, Paris Kanellakis, and Peter Miltersen for useful discussions.

References

- [A83] M. Ajtai (1983), " Σ_1^1 Formulae on Finite Structures," *Annals of Pure and Applied Logic* **24**, 1-48.
- [B89] D.M. Barrington, "Bounded-Width Polynomial-Size Branching Programs Recognize Exactly Those Languages in NC^1 ," *J. Comput. Sys. Sci.* , 38 (1989), 150-164.
- [BC89] D.M. Barrington, J.C. Corbett, "On the Relative Complexity of some Languages in NC^1 ," *Technical Report 89 - 22*, Department of Computer Science, University of Massachusetts, Amherst (1989).
- [BRS91] A. Borodin, R. Razborov, S. Smolensky. "On Lower Bounds for Read- k Times Branching Programs," Preprint, (1991).
- [CH82] A. Chandra and D. Harel, "Structure and Complexity of Relational Queries," *J. Comput. Sys. Sci.* , 25 (1982), 99-128.
- [CT91] R. Cohen, R. Tamassia, "Dynamic Expression Trees and their applications," *Proceedings of the 2nd Annual ACM-SIAM SODA*, (1991).
- [DS95] G. Dong, J. Su, "Space-Bounded FOIES," *ACM Symp. Principles Database Systems* (1995), 139 -150.
- [DS93] G. Dong, J. Su, "Incremental and Decremental Evaluation of Transitive Closure by First-Order Queries," *Information and Computation* , 120(1) (1995), 101-106.
- [DST93] G. Dong, J. Su, R. Topor. "First-Order Incremental Evaluation of Datalog Queries", *Proceedings of the 1992 International Conference on Database Theory*, LNCS 646, Springer Verlag.
- [E*92] D. Eppstein, Z. Galil, G. F. Italiano, A. Nissenzweig. "Sparsification - A technique for speeding up dynamic graph algorithms," *IEEE Found. of Comp. Sci. Symp.* (1992), 60-69.
- [E2*92] D. Eppstein, Z. Galil, G. F. Italiano, A. Nissenzweig. "Sparsification and Planarity Testing," *Proceedings of ACM Symposium on Theory of Computing* (1992).
- [Fa74] Ron Fagin, "Generalized First-Order Spectra and Polynomial-Time Recognizable Sets," in *Complexity of Computation*, (ed. R. Karp), *SIAM-AMS Proc.* 7 (1974), 27-41.
- [F85] G.F. Frederickson, "Data structures for on-line updating of minimum spanning trees," *SIAM J. Comput.* , 14 (1985), 781-798.
- [F91] G.F. Frederickson, "Ambivalent data structures for dynamic 2-edge connectivity and k smallest spanning trees," *IEEE Found. of Comp. Sci. Symp.* (1991), 632-641.
- [FS] M. Fredman and M. Saks, "The Cell Probe Complexity of Dynamic Data Structures," *21st ACM STOC Symp.* (1989), 345-354.
- [FSS84] M. Furst, J.B. Saxe, and M. Sipser, "Parity, Circuits, and the Polynomial-Time Hierarchy," *Math. Systems Theory* **17** (1984), 13-27.
- [GMS93] A. Gupta, I. S. Mumick, V. S. Subrahmanian, "Maintaining Views Incrementally," *Proceedings of the ACM SIGMOD* (1993), 157-166.

- [I89] N. Immerman, "Descriptive and Computational Complexity," in *Computational Complexity Theory*, ed. J. Hartmanis, *Proc. Symp. in Applied Math.*, 38, American Mathematical Society (1989), 75-91.
- [I89b] Neil Immerman, "Expressibility and Parallel Complexity," *SIAM J. of Comput.*, 18 (1989), 625-638.
- [I87] N. Immerman, "Languages That Capture Complexity Classes," *SIAM J. Comput.*, (16:4) (1987), 760-778.
- [I81] N. Immerman, "Number of Quantifiers is Better than Number of Tape Cells," *J. Comput. Sys. Sci.*, (22:3) (1981), 65-72.
- [IL94] N. Immerman, S. Landau, "The Complexity of Iterated Multiplication," *Information and Computation*, (116:1) (1995), 103-116.
- [Io85] Y. Ionanadis, "A Time Bound on the Materialization of Some Recursively Defined Views," *Proceedings of International Conference on Very Large Data Bases*, 1985.
- [J92] H. Jakobsson, "On Materializing Views and On-line Queries," *Proceedings of International Conference on Database Theory*, Berlin, (1992), 407-420.
- [LT94] Y. Liu and T. Teitelbaum, "Systematic Derivation of Incremental Programs," *Science of Computer Programming* 24 (1995), 1-39.
- [LST87] J. W. Lloyd, E. A. Sonenberg and R. W. Topor, "Integrity Constraint Checking in Stratified Databases," *Journal of Logic Programming*, 4(4):334-343, 1987.
- [MI94] J.A. Medina and N. Immerman, "A Syntactic Characterization of NP-Completeness," *IEEE Symp. Logic In Comput. Sci.* (1994), 241-250.
- [M2] P. B. Miltersen. "The Bit Probe Complexity Measure Revisited," *Proceedings of the Tenth Symposium on Theoretical Aspects of Computer Science* (1993).
- [MSV94] Peter Bro Miltersen, Sairam Subramanian, Jeffrey Scott Vitter, and Roberto Tamassia, "Complexity Models for Incremental Computation," *Theoret. Comp. Sci.* (130:1) (1994), 203-236.
- [N82] J-M. Nicolas, "Logic for Improving Integrity Checking in Relational Databases," *Acta Informatica*, (18:3) (1982), 227-253.
- [P94] S. Patnaik, "The Dynamic Complexity of Approximation," Manuscript, Computer Science Dept., University of Massachusetts (1994).
- [PI94] S. Patnaik and N. Immerman, "Dyn-FO: A Parallel, Dynamic Complexity Class," *ACM Symp. Principles Database Systems* (1994), 210-221.
- [R94] M. Rauch. "Improved Data Structures for Fully Dynamic Biconnectivity," *ACM Symp. Theory Of Comput.* (1994), 686-695.
- [TY79] R. Tarjan and A. Yao, "Storing a Sparse Table," *Communications of the ACM* 22:11 (1979), 606-611.
- [RR93] G. Ramalingam and T. Reps, "A Categorized Bibliography on Incremental Computation," *ACM Symp. Principles of Programming Languages* (1993), 502-510.