Theory Comput. Systems **40**, 355–377 (2007)
DOI: 10.1007/s00224-006-1312-0

Theory of Computing Systems
© 2007 Springer Science+Business Media, Inc.

Dynamic Complexity Theory Revisited

Volker Weber and Thomas Schwentick

Fachbereich Informatik, Universität Dortmund,
D-44227 Dortmund, Germany
{Volker.Weber,Thomas.Schwentick}@udo.edu

Abstract. Dynamic complexity investigates the required effort to maintain knowledge about a property of a structure under changing operations. This article introduces a refined notion of dynamic problems which takes the initial structure into account. It develops the basic structural complexity notions accordingly. It also shows that the dynamic version of the LOGCFL-complete problem D_2LREACH(acyclic) can be maintained with first-order updates.

1. Introduction

For a set S, the *static decision problem* asks whether a given input I is an element of S. Classical *static complexity theory* studies the inherent computational effort to answer this question. However, I often undergoes small changes, and information about its membership in S should be available after each change. As an example let us consider the problem REACH consisting of all triples (G, s, t), where G is a graph and s and t are nodes such that t is reachable from s in G. Changes could be insertion and deletion of edges. Another typical example is a view in a database with tuple insertions and deletions to the base relations. Obviously, there are many other situations of this kind.

Of course, in many cases one can expect that if I' results from I by applying a small change, whether $I' \in S$ might be closely related to whether $I \in S$. In particular, it should often be simpler to *maintain* information about membership in S under small changes than to recompute it from scratch for each new instance. This might involve auxiliary data structures that are updated accordingly.

These considerations are the starting point of *dynamic complexity theory* which was initiated in [MSVT] and [PI]. This theory has been focusing on two lines of research, structural results about complexity classes of dynamic problems, including suitable reduction concepts and complete problems, and upper bounds for concrete problems. Both

lines consider problems with a very low update complexity, especially with updates expressible in first-order logic (or, equivalently, by uniform families of constant-depth, polynomial-size circuits with unbounded fan-in, aka AC^0). This class is called DynFO in [PI].

In [HI], low-level reductions for dynamic problems are defined and a complete problem (CSSCV) for DynFO is presented. Although this was an important step, there remained some steps to be done. First, as the authors pointed out, it is still a challenge to find *natural* complete problems for dynamic classes. Second, in the construction of the complete problem a technical difficulty arose that was caused by the role of the initial structure of a dynamic problem in the setting of [PI] and [HI]. Further, the reductions seem hard to use for more natural problems. Also, the class DynFO itself does not distinguish between problems of quite different static complexity, e.g., between LOGSPACE and PTIME.

Concerning upper bounds, the dynamic versions of many problems inside NL have been shown to be in DynFO [PI], e.g., reachability in directed, acyclic graphs or in undirected graphs. The technically most demanding result was that reachability in directed (possibly cyclic) graphs is in DynTC^0, i.e., the dynamic class with updates computable by constant-depth threshold circuits [H].

It is natural to ask for the highest complexity of a (static) problem, such that its dynamic version allows for first-order updates. The answer to this question has two facets. It was shown in [MSVT] and [PI] that there are even P-complete problems in DynFO. Nevertheless, these problems are very artificial, containing structures that are highly redundant.[1] Concerning non-redundant problems the complexity-wise highest problems in DynFO known so far are complete for NL.

Contributions. This article contributes to both lines of research mentioned above. First, by taking the complexity of the initial instance I into account in a very simple fashion, we define more refined complexity classes and corresponding reduction concepts. More precisely, our classes are of the form $\text{Dyn}(\mathcal{C}, \mathcal{C}')$, where $\mathcal{C}$ is the complexity of computing the auxiliary data structure for the initial input I and $\mathcal{C}'$ is the complexity of the updates.

We show that these classes and reductions behave nicely and that the results of [PI], [H], and [HI] translate in a straightforward way. The new classes allow a more precise classification of problems. We show that most of the problems mentioned above are in the respective class $\text{Dyn}(\mathcal{C}, \text{FO})$, where $\mathcal{C}$ is the complexity of the underlying static problem. Nevertheless, optimality with respect to the initial input complexity is not automatic, e.g., it is not clear whether the dynamic reachability problem is in $\text{Dyn}(\text{NL}, \text{TC}^0)$.

The technically most difficult result of this article contributes to the other line of research. It presents a (non-redundant) LOGCFL-complete problem with first-order updates, more precisely, in Dyn(LOGCFL, FO).

Related work. In a series of papers (e.g., [DS1]–[DS4] and [DLW]) *first-order incremental evaluation systems* have been studied, which are basically the analogue of DynFO for database queries. In [LW1] and [LW2] SQL was used as update language.

[1] Basically, this redundancy implies that a change to a structure requires a series of identical changes, thus allowing to compute the updates in a series of simple subcomputations. A more precise account can be found in Section 7.

There is a huge body of work on algorithms for dynamic problems, e.g., [HdLT] and [RZ] and in *Online Algorithms* [FW].

Organization. In Section 2 we give the basic definitions of dynamic problems and complexity classes. Some precise upper bounds are provided in Section 3. Reductions between dynamic problems are addressed in Section 4, complete problems in Section 5. Connections to static complexity theory are the focus of Section 6. In Section 7 we exhibit a LOGCFL-complete problem that is in Dyn(LOGCFL, FO). We conclude in Section 8.

2. Definitions

In this section we define our notions of dynamic problems and dynamic complexity classes. They depart considerably from [PI]. We discuss the relationship between the definitions at the end of this section.

Intuitively, in our view, a dynamic problem D is induced by a static problem S, i.e., a set of structures and a set of operations. A pair (A, w) consisting of a structure A and a sequence w of operations is in D if the structure resulting from A after applying w is in S. As an example, in dynREACH, the dynamic version of REACH, A is a triple (G, s, t) and w is a sequence of instructions of the form $\texttt{insert}(i, j)$ and $\texttt{delete}(i, j)$, where i and j are elements of G.

We turn to the formal definitions. We write $\text{STRUC}_n(\tau)$ for the class of structures with n elements over vocabulary τ. For example, if $\tau = \{E\}$ and E is a binary (edge) relation symbol then $\text{STRUC}_5(E)$ denotes all graphs with 5 vertices. For simplicity, we assume the universe of a structure always to be $[n] := \{0, \ldots, n-1\}$. We only consider vocabularies with relation and constant symbols.

In general, we use *operation symbols* σ from a finite set Σ with an associated arity arity(σ). Thus, for dynREACH, arity($\texttt{insert}$) = arity($\texttt{delete}$) = 2. An *operation* on a structure $\mathcal{A} \in \text{STRUC}_n(\tau)$ is simply a tuple $\sigma(a_1, \ldots, a_m)$ with $a_i \in [n]$, for $i \leq m$ and $m = \text{arity}(\sigma)$. We denote the set of operations with symbols from Σ over structures from $\text{STRUC}_n(\tau)$ by Σ_n.

The semantics of operations is given by an *update function* g which maps a pair $(A, \sigma(a_1, \ldots, a_m))$ from $\text{STRUC}_n(\tau) \times \Sigma_n$ to a structure $\sigma^g(a_1, \ldots, a_m)(A)$ from $\text{STRUC}_n(\tau)$. We usually write $\sigma(a_1, \ldots, a_m)(A)$ for this structure. For a string $w = w_1 \cdots w_m$ of operations and a structure A we write $w(A)$ for $w_m(\cdots(w_1(A)\cdots)$.

Definition 2.1. Let τ be a vocabulary, S a set of τ-structures, Σ a set of operation symbols, and g an update function. The *dynamic problem* $D(S, \Sigma, g)$ associated with S, Σ, and g is the set of pairs (A, w), where, for some $n > 0$, $A \in \text{STRUC}_n(\tau)$, $w \in \Sigma_n^*$, and $w(A) \in S$. We call S the *underlying static problem of* D.

To simplify notation, we often do not mention g explicitly and denote dynamic problems simply by $D(S, \Sigma)$.

The computations we want to model are of the following kind: First, from the input structure A an *auxiliary data structure* B is computed. Afterward, for each operation

w_i this auxiliary structure is updated in order to reflect the changes of A. The auxiliary structure can be used to compute whether $w_1 \cdots w_i(A)$ is in S. In our framework for dynamic complexity classes we consider the costs for the initial computation and the updates separately. There is a trade-off between these costs, and the interesting case is, of course, where the costs for updates are very low. In this case the costs of the initial computation cannot be better than the complexity of S itself. More precisely, $\mathrm{Dyn}(\mathcal{C}, \mathcal{C}')$ is the class of problems for which the initial computation can be done within complexity class $\mathcal{C}$ and the updates within $\mathcal{C}'$.

Definition 2.2. Let τ be a fixed vocabulary and let $\mathcal{C}$ and $\mathcal{C}'$ be complexity classes. $\mathrm{Dyn}(\mathcal{C}, \mathcal{C}')$ is the class of all dynamic problems $D = D(S, \Sigma, g)$ satisfying the following conditions:

- There is a vocabulary ρ, a set S' of ρ-structures, and a $\mathcal{C}$-computable function $f\colon \mathrm{STRUC}[\tau] \to \mathrm{STRUC}[\rho]$ such that
  - $f(A) \in \mathrm{STRUC}_n[\rho]$ for all $A \in \mathrm{STRUC}_n[\tau]$;
  - $f(A) \in S'$ if and only if $A \in S$.
- There is a $\mathcal{C}'$-computable function f' mapping tuples $(B, \sigma, a_1, \dots, a_{\mathrm{arity}(\sigma)})$, where, for some n, B is from $\mathrm{STRUC}_n[\rho]$ and $\sigma(a_1, \dots, a_{\mathrm{arity}(\sigma)}) \in \Sigma_n$, to structures in $\mathrm{STRUC}_n[\rho]$ such that for each n, each $A \in \mathrm{STRUC}_n[\tau]$ and each operation sequence $w \in \Sigma_n^*$:

  $$w(A) \in S \quad \Longleftrightarrow \quad f'(f(A), w) \in S',$$

  where f' is extended to sequences of operations in the obvious way.
- $S' \in \mathcal{C}'$.

We are mainly interested in the case where $\mathcal{C}'$ is the very weak complexity class AC^0 [V]. As AC^0 contains exactly the problems that can be characterized by first-order formulas with built-in arithmetic[2] and as in the context of mathematical structures logical formulas are the natural means to express function f', we usually refer to $\mathcal{C}'$ by its corresponding logic [I2], [L], e.g., in $\mathrm{Dyn}(\mathcal{C}, \mathrm{FO})$.

Remarks 2.3.

- We do not pose any restriction on the update function g. Therefore, it is in general not possible to infer the complexity of the dynamic problem from the complexity of the underlying static problem. This situation changes when we consider canonical dynamic problems as in Section 6.

  On the other hand, of course all problems in a class Dyn(C,C') are based on static problems in $\mathcal{C}$.
- Viewed more abstractly, Definition 2.2 imposes a kind of reduction from a dynamic problem D to a dynamic problem D'. Actually, it is a 1-bounded $(\mathcal{C}, \mathcal{C}')$-reduction in the notation below.

[2] Throughout this article, circuit classes AC^0 and TC^0 are always DLOGTIME-uniform (see [BIS] and [V]).

- In general, the notion of $\mathcal{C}$-computable functions is not unambiguously defined. However, for the cases of interest in this paper, the connection between function and language classes is obvious, the class of P-computable functions is FP, the class of L-computable functions is FL, and the class of NL-computable functions is $\mathrm{FNL} = \mathrm{FL}^{\mathrm{NL}}$.
- We follow [PI] in that we do not allow operations that delete or insert *elements* into structures. On the other hand, that the auxiliary structure $f(A)$ has the same size as A is not a severe restriction, as all auxiliary structures of polynomial size $p(n)$ can be encoded over the universe of A.

The main difference between our definitions and the setting of [PI] and [HI] is that our definition includes structures and arbitrary operations, whereas [PI] talks about static problems extended by a restricted set of operations (corresponding to the canonical dynamic problems of Section 6) and [HI] defines dynamic problems by sequences of operations only.

To be more precise, in [HI] a dynamic problem is an infinite sequence $D = \{D_n \subseteq \Sigma_n^* \mid n = 1, 2, \ldots\}$ of regular languages, where the elements of Σ_n are the operations with parameters from $\{0, \ldots, n-1\}$. Dynamic complexity classes are defined via *dynamic machines*, which are uniform sequences of deterministic finite automata. The class Dyn$\mathcal{C}$ consists of all problems accepted by a dynamic machine whose initial state, transition function, and final state set are $\mathcal{C}$-computable. The class where a polynomial-time initial computation is allowed is called Dyn$\mathcal{C}^+$.

Nevertheless, there is obviously a close correspondence between dynamic problems in their and our setting. Consider a dynamic problem D in our model. If we fix a uniform sequence $I = (I_n)_{n\geq 0}$ of initial structures and consider those sequences of operations that lead to acceptance when applied to a structure from I, we get a dynamic problem in the sense of [HI].[3] We call this problem $D|_I$. Therefore, we might state that

$$\mathrm{Dyn}(\mathcal{C}, \mathcal{C}) \subseteq \mathrm{Dyn}\mathcal{C} \quad \text{and} \quad \mathrm{Dyn}(\mathrm{P}, \mathcal{C}) \subseteq \mathrm{Dyn}\mathcal{C}^+,$$

because these inclusions are independent of the choice of the sequence of initial structures, which only has to be $\mathcal{C}$-uniform or P-uniform, respectively.

For the latter inclusion we can even prove equality if we pose an additional restriction on the dynamic problems. We call a dynamic problem *polynomially connected* if for each pair A, A' of structures of size n there is a polynomial-size sequence w of operations such that $w(A) = A'$ and w can be computed from A and A' in time polynomial in n.

Proposition 2.4. *Let* $\mathcal{C} \subseteq \mathrm{P}$ *be a static complexity class. If a dynamic problem* D *is polynomially connected and* $D|_I$ *is in* Dyn$\mathcal{C}^+$ *for some sequence of initial structures* I, *then* D *is in* Dyn(P, $\mathcal{C}$).

Proof. Consider a dynamic machine showing that $D|_I$ is in Dyn$\mathcal{C}^+$. The sequence of initial states of this machine is the sequence of auxiliary data structures for I, computable in polynomial time by assumption and because I is P-uniform. The transition function tells us how to update these data structures with complexity $\mathcal{C}$.

[3] In fact, all dynamic problems presented in [HI] are defined in this way.

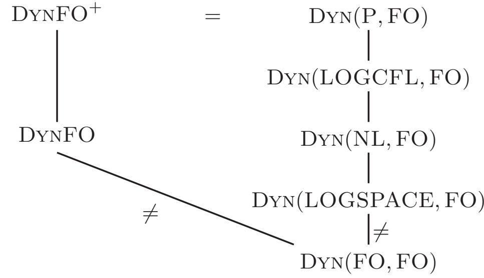

Fig. 1. The relationship of our dynamic complexity classes to that of [PI], [HI]. Note that these classes are based on a different definition of dynamic problems and that the equality only holds for polynomially connected problems.

It only remains to show how to compute the auxiliary data structure for an arbitrary input structure A of size n in polynomial time. This can be done by first computing the auxiliary data structure X_n for I_n. Because D is polynomially connected, we can compute a polynomial length sequence of operations w such that $A = w(I_n)$. By applying the transformations induced by w to X_n according to w we obtain an initial auxiliary structure for A. All steps can be carried out in polynomial time by assumption.

Altogether, we have proven that D is in Dyn(P, $\mathcal{C}$). □

The restriction to polynomially connected problems certainly rules out some dynamic problems. On the other hand, all dynamic problems mentioned in this article are polynomially connected.

We summarize all inclusions stated above, in the case of first-order updates, in Figure 1.

3. Some Precise Upper Bounds

In this section, we show membership in Dyn($\mathcal{C}$, FO) for various dynamic problems, where the underlying static problem is complete for $\mathcal{C}$. In the following, all dynamic graph problems have insertion and deletion of edges as operations.

Theorem 3.1.

(a) *The dynamic* PARITY *problem*[4] dynPARITY, *with operations that set or unset one bit*, *is in* Dyn(ACC^0[2], FO).
(b) *The dynamic deterministic reachability problem* dynREACH$_d$, *which asks for a path on which every node has outdegree* 1, *is in* Dyn(LOGSPACE, FO).
(c) dynREACH *for undirected graphs is in* Dyn(LOGSPACE, FO).
(d) *Dynamic* 2*-colorability for undirected graphs is in* Dyn(LOGSPACE, FO).
(e) dynREACH *for acyclic graphs is in* Dyn(NL, FO).

[4] For concreteness, a string is in PARITY if its number of 1-symbols is even.

Proof (*Sketch*). (a) requires only one auxiliary bit, the parity bit itself. (c) and (d) follow directly by analyzing the proofs in [PI]. They require the construction of a spanning tree, but SL = LOGSPACE [R] allows the contruction of the lexicographically least spanning tree within logarithmic space. (b) and (e) can be concluded from [DS2]. □

From [H] we can conclude dynREACH $\in$ Dyn(P, TC^0); whether this is optimal remains open. From this result one also gets dyn2SAT $\in$ Dyn(P, TC^0), where the allowed operations are insertion and deletion of clauses.

4. Reductions

Reductions are the key tool of computational complexity. In this section we introduce dynamic reductions that suit our dynamic complexity classes.

Given an instance (A, w) of a dynamic problem D, a reduction from D to D' has to map A to an initial structure A'. Further, every operation w_i has to be mapped to one or more operations of D'. In principle, one could allow the image of w_i to depend on the previous operations $w_1, \ldots, w_{i-1}$ and on A. However, as we are interested in a weak notion of reductions, we follow [HI] and require that each w_i is mapped independently.

Definition 4.1. Let $D = D(S, \Sigma)$ and $D' = D(S', \Sigma')$ be dynamic problems over vocabularies τ and τ', respectively. A *reduction* from D to D' is a pair (f, h) of mappings with the following properties:

- For each $n \in \mathbb{N}$, f maps τ-structures of size n to τ'-structures of size n', where $n' = p(n)$ for some polynomial p.
- For each $n \in \mathbb{N}$, h is a string homomorphism from Σ_n^* to $\Sigma_{n'}^{\prime *}$.
- For each τ-structure A, and each sequence w of operations on A,

$$(f(A), h(w)) \in D' \iff (A, w) \in D.$$

If f and h can be computed in complexity $\mathcal{C}$ and $\mathcal{C}'$, respectively, we say that the reduction is a $(\mathcal{C}, \mathcal{C}')$-reduction and write $D \leq_{\mathcal{C},\mathcal{C}'} D'$.

Note that if (f, h) is a reduction from $D(S, \Sigma)$ to $D' = D(S', \Sigma')$ then, in particular, f is a reduction from S to S'.

Note also that although we require that $h(\sigma)(f(A)) \in D' \iff \sigma(A) \in D$, this does not imply $h(\sigma)(f(A)) = f(\sigma(A))$. In fact, if the latter holds, for each A and w, (f, h) is called a *homomorphism*.[5]

A reduction is k-*bounded* if $|h(s)| \leq k$ for each $s \in \bigcup_{i=1}^{\infty} \Sigma_i$. It is *bounded* if it is k-bounded for some k. Observe that $|h(s)|$ refers to the number of operations, not to the length of their encoding. We write $\leq^b$ for bounded reductions.

We are mainly interested in the case where $\mathcal{C}$ is LOGSPACE or FO and $\mathcal{C}' =$ FO.

[5] In [HI] the term *homomorphism* was used for what we call a *reduction*.

Example 4.2. dynPARITY $\leq^b_{\mathrm{FO,FO}}$ dynREACH$_d$ [HI]: Here, f maps a string $x = x_1 \cdots x_n$ to the graph with nodes (i, j), where $i \in [n+1]$ and $j \in \{0, 1\}$. If $x_i = 0$ there are edges from $(i-1, 0)$ to $(i, 0)$ and from $(i-1, 1)$ to $(i, 1)$. Otherwise, there are edges from $(i-1, 0)$ to $(i, 1)$ and from $(i-1, 1)$ to $(i, 0)$. Clearly, x is in PARITY iff there is a path from $(0, 0)$ to $(n, 0)$. Each operation on the string is mapped to at most four operations in the graph in a straightforward manner.

The following propositions, which are easy to prove, show that our dynamic reductions and dynamic complexity classes fit together.

Proposition 4.3. *The relations* $\leq^b_{\mathrm{LOGSPACE,FO}}$ *and* $\leq^b_{\mathrm{FO,FO}}$ *are transitive.*

Proposition 4.4. *Let* $\mathcal{C}$ *and* $\mathcal{C}'$ *be closed under functional composition*, FO $\subseteq \mathcal{C}$ *and* FO $\subseteq \mathcal{C}'$. *Then* Dyn$(\mathcal{C}, \mathcal{C}')$ *is closed under bounded* (FO, FO)*-reductions.*

Proposition 4.4 also holds for bounded (LOGSPACE, FO)-reductions if $\mathcal{C}$ is required to be closed under logspace-reductions.

5. Complete Problems

In [HI] a complete problem for DynFO under bounded first-order homomorphisms was established. We show next that the problem can be translated into our setting and, furthermore, it can be adapted to obtain complete problems for classes of the type Dyn$(\mathcal{C}$, FO).

The dynamic problem *single step circuit value* (SSCV) of (possibly cyclic) circuits is defined in [HI]. It has operations to change the type of a gate—“and”, “or”, or “nand”—to add and delete wires between gates, and to set the current value of a gate. There is also an operation that propagates values one step through the circuit. A sequence of such operations belongs to SSCV if its application to a circuit of unconnected “and” gates of value “0” results in a circuit with gate 0 having value “1”.

To translate SSCV to our setting, we have to add an underlying static problem. The most natural choice would be the set of all such circuits whose gate 0 has value “1”. That is, SSCV is the set of all pairs consisting of a circuit and a sequence of operations, such that gate 0 of the resulting circuit has value “1”. However, this problem has very low complexity. More precisely, it is in Dyn(FO,FO).

Proposition 5.1. *For every class* $\mathcal{C}$ *containing* FO, SSCV *is complete for* Dyn(C,FO) *under bounded* ($\mathcal{C}$, FO)*-reductions.*

Proof. First-order updates to SSCV are possible with an auxiliary data structure containing only the relations needed to describe the circuit [HI], i.e., no computation is needed to obtain the auxiliary data structure.

We can describe a bounded $(\mathcal{C}, \mathrm{FO})$-reduction (f, h) from an arbitrary dynamic problem D in Dyn(C,FO) to SSCV based on the proof of Theorem 7.4 in [HI]. They describe a first-order definable circuit whose main components are an array of latches,

meant to hold the auxiliary data structure for D, and circuits for all operations of D that use this array of latches to get their input from, as well as to write their output to. The latter circuits exist because there are first-order updates to D by assumption. There are also some additional gates used to control the computation. For more details we refer to [HI].

We use f to construct this circuit as well as to compute the auxiliary data structure corresponding to the given instance of D and to store it in the array of latches. This can be done because this auxiliary data structure is $\mathcal{C}$-computable as D is in Dyn(C,FO). The operations can be mapped by h as in [HI], i.e., the circuit corresponding to an operation is selected and stepped through, writing the updated auxiliary data structure to the array of latches. □

To obtain a dynamic problem complete for a larger class, e.g., Dyn(P,FO), under weaker reductions, such as bounded (FO, FO)-reductions, we have to choose a static problem that is complete for P. We use a modification of the generic complete problem for P. To this end, let S be the set of all tuples $(c, x, 1^m)$ such that c encodes a Turing machine (TM) that on input x computes within m steps a circuit whose gate 0 evaluates to "1". The dynamic problem $\mathrm{SSCV_P}$ is defined by adding the operations from above to S, i.e., we substitute the circuits of SSCV by instructions to build a circuit. It should be stressed that the operations change the circuit, not the TM.

Theorem 5.2. $\mathrm{SSCV_P}$ *is complete for* Dyn(P,FO) *under bounded* (FO, FO)*-reductions*.

Proof. Let $D = D(S, \Sigma)$ be a problem in Dyn(P,FO). We describe a bounded (FO, FO)-reduction (f, h) from D to $\mathrm{SSCV_P}$.

An instance (A, w) of D is mapped by f to a coding of a TM M and an input x for M. M is basically a combination of two other TMs, one to compute the auxiliary data structure to the given instance A of D and one that outputs the circuit described in the proof of Theorem 7.4 in [HI], which we described above. The latter one is first-order definable, and the first one is fixed since A can be given as input x. Therefore, we can describe in first order a TM M that has the circuit of [HI] with respect to the operations of D as output, with the auxiliary data structure for A stored in the array of latches.

The operations can be mapped as before. □

An analogous result can be obtained for other dynamic complexity classes like Dyn(NL,FO) and Dyn(LOGSPACE,FO) as well as to classes of the form Dyn($\mathcal{C}$, TC^0).

6. Connections to Static Complexity

Now, we establish some connections between static and dynamic problems.

To transfer properties from static problems to dynamic problems, the dynamic problems need to depend in a uniform way on their underlying static problems.

Most of the problems studied in earlier work (e.g., [PI], [DS2], [E], and [H]) have essentially the same operations: insertion and deletion of tuples and setting constants.

Therefore, we now study dynamic problems with these operations. For a static problem $S \subseteq \text{STRUC}[\tau]$ we define the set $\Sigma_{\text{can}}(S)$ of canonical operations by

$$\Sigma_{\text{can}}(S) = \{\texttt{insert}_R, \texttt{delete}_R \mid \text{for all relation symbols } R \in \tau\}$$
$$\cup \{\texttt{set}_c \mid \text{for all constant symbols } c \in \tau\}.$$

Thus, an operation is of one of the forms $\texttt{insert}_R(u, v)$, $\texttt{delete}_R(u, v)$ or $\texttt{set}_c(u)$ where the letter has the effect of assigning u to c.

We call $D_{\text{can}}(S) := D(S, \Sigma_{\text{can}}(S), g_{\text{can}})$ the *canonical dynamic problem* of S, where g_{can} is the update function corresponding to the intended meaning of the operation symbols from Σ_{can}.

Obviously, there is a close correspondence between static and dynamic complexity classes in the case of canonical dynamic problems. If S is a static problem and C a static complexity class, we have

$$S \in C \quad \Longleftrightarrow \quad D_{\text{can}}(S) \in \text{Dyn}(\mathcal{C}, \mathcal{C}).$$

An interesting question is the dynamic complexity of NP-problems. One might assume that if some NP-complete problem has polynomial dynamic complexity (i.e., is in $\text{Dyn}(\mathcal{C}, \text{P})$ for some class $\mathcal{C}$) then $\text{P} = \text{NP}$. Of course, this holds for $\mathcal{C} = \text{P}$, but we cannot prove it for more powerful classes $\mathcal{C}$. Nevertheless, we can show the following result which draws a simple but interesting connection between dynamic and non-uniform complexity.

Theorem 6.1. *Let S be* NP*-complete. Then* $\text{NP} \subseteq \text{P/poly}$ *if and only if there is a class $\mathcal{F}$ of functions such that* $D_{\text{can}}(S) \in \text{Dyn}(\mathcal{F}, \text{P})$.

Proof. For the *if* direction, as P/poly is closed under reductions, it suffices to show that S is in P/poly if its canonical dynamic version is in $\text{Dyn}(\mathcal{F}, \text{P})$. For each n, let E_n denote the τ-structure of size n with empty relations and all constants set to 0.

The advice B_n for size-n inputs is just the auxiliary structure for E_n. Whether a structure A of size n is in S can be tested by applying, for each $t \in R$ an operation $\texttt{insert}_R(t)$ to E_n and for each constant $c = u$, an operation $\texttt{set}_c(u)$. The order of these applications is arbitrary. Whether $A \in S$ can be derived by another polynomial-time computation from the final result of this process.

For the *only if* direction, assuming $S \in \text{NP} \subseteq \text{P/poly}$, let C_n denote the polynomial advice for S for inputs of size n. The auxiliary structure for a structure A basically is (A, C_n). The update operations only change the A part of the auxiliary structure. Clearly, updates can be done in polynomial time and checking membership of (A, C_n) is in polynomial time by assumption. □

The definition of reductions between dynamic problems already requires that there is a reduction between the underlying static problems. The following result shows that

for a homomorphism between dynamic problems one can say more. In a *bounded* FO-*reduction* (bfo) each tuple of the source structure affects only a bounded number of tuples in the target structure, i.e., if a tuple in the source structure is changed only a bounded number of tuples in the target structured needs to be changed to get the image of the new source structure [PI].

Theorem 6.2. *Let S and T be static problems. For every bounded* (FO, FO)*-homomorphism (f, h) from $D_{\text{can}}(S)$ to $D_{\text{can}}(T)$, f is a bfo-reduction from S to T.*

Proof. By definition, f is an FO-reduction from S to T. We have to show that f is bounded in the abovementioned sense.

Let τ be the signature of S and let A and A' be two τ-structures that differ only in a single tuple t, say, $A' = \texttt{insert}(t)(A)$. Because (f, h) is a homomorphism, $f(A)$ and $f(A')$ differ only in the tuples and constants affected by the operations in $h(\texttt{insert}(t))$. The number of these operations is bounded and each operation affects only one tuple or constant. Since h does not depend on A and A', we can conclude that f is bounded and therefore a bfo-reduction. □

This theorem enables us to draw some conclusions concerning the completeness of canonical dynamic problems for dynamic complexity classes. As stated in [HI], canonical dynamic problems cannot be complete for dynamic complexity classes in the general setting. However, this no longer holds if we restrict to classes of canonical dynamic problems.

By Theorem 6.2, problems complete for a class of canonical dynamic problems under bounded (FO, FO)-homomorphisms must be based on problems complete under bfo-reductions. However, we know from [PI] that REACH is not complete for NL under bfo-reductions. Their proof shows that there is no bfo-reduction from CLIQUE-ONLY, the set of graphs that are the union of a co-clique (a graph without edges) and a clique with the same number of nodes [BRS], to REACH. The canonical dynamic problem for CLIQUE-ONLY is easily seen to be in Dyn(TC^0, FO). Therefore, we get the following corollary, where $\text{Dyn}_{\text{can}}(\mathcal{C}, \mathcal{C}')$ denotes the restriction of $\text{Dyn}(\mathcal{C}, \mathcal{C}')$ to canonical dynamic problems.

Corollary 6.3. *D_{can}(REACH) is not $\text{Dyn}_{\text{can}}(\text{TC}^0, \text{FO})$-hard under bounded* (FO, FO)-*homomorphisms.*

Therefore, D_{can}(REACH) cannot be complete for Dyn_{can}(NL, FO) under bounded (FO, FO)-homomorphisms. This result can be extended to other problems and classes, e.g., $D_{\text{can}}(\text{REACH}_d)$ and Dyn_{can}(LOGSPACE, FO).

On the other hand, there are some results on completeness under bfo-reductions in [PI]: REACH_a and CIRVAL are complete for P, COLOR-REACH for NL and COLOR-REACH_d for LOGSPACE. They carry over to canonical dynamic problems, e.g., D_{can}(CIRVAL) is complete for Dyn_{can}(P, P) and D_{can}(COLOR-REACH) is complete for Dyn_{can}(NL, NL) under bounded (FO, FO)-homomorphisms. Unfortunately, no efficient updates are known for these problems.

7. A LOGCFL-Complete Problem with First-Order Dynamic Complexity

In this section we turn to the question of the maximum possible static complexity of a problem with first-order updates. In dealing with this question one has to distinguish between "redundant" and "non-redundant" problems.

It was observed in [MSVT] and [PI] that by blowing up the encoding of a problem its dynamic complexity can be decreased and that each problem in Phas a *padded version* in Dyn(P, FO). To be more precise, let, for a (string) problem S, PAD(S) be its *padded version* $\text{PAD}(S) = \{w^{|w|} \mid w \in S\}$. If it is a set of graphs then instances of PAD(S) consist of n disjoint n-vertex graphs. It was shown in [PI] that PAD(REACH_a), the padded version of the alternating reachability problem is in Dyn(P, FO). Note that, for graphs with n nodes it needs n operations to make a significant change, i.e., this result is basically a restatement of the fact that REACH_a is in FO[n] [I1].

As noted by the authors in [MSVT], the latter result depends on the redundant encoding. They defined a notion of non-redundant problems by completeness under a certain kind of reduction. However, this notion does not seem to be applicable to complexity classes below P.

We are interested in first-order updates to non-redundant problems. The complexity-wise highest problems known so far to have such updates are complete for NL, e.g., the reachability problem on acyclic graphs.

In this section we improve this result by establishing first-order updates for a non-redundant problem complete for LOGCFL.

The class LOGCFLconsists of all problems that are logspace-reducible to a context-free language and is placed between NL and AC^1. More on LOGCFL can be found in [GLS].

The problem we consider is the canonical dynamic problem for a reachability problem on labeled, acyclic graphs: D_2LREACH(acyclic). The labels are symbols drawn from $\Sigma = \{a, b, \bar{a}, \bar{b}\}$. Each edge has a unique label and there is at most one edge from a node u to a node v. The problems asks for a path between two nodes s and t that is labeled by a string in D_2, the Dyck language with two types of parentheses defined by the following context-free grammar:

$$D_2\text{:} \quad S \to aS\bar{a}S \mid bS\bar{b}S \mid \varepsilon.$$

Proposition 7.1. D_2LREACH(acyclic) *is complete for* LOGCFL.

Proof. LOGCFLcan be characterized by non-deterministic *auxiliary pushdown automata* (NAuxPDA) first studied by Cook [C]. An auxiliary pushdown automaton is basically a TM with read-only input tape and several work tapes. One of these work tapes can only be used as a pushdown store but it is not affected by space constraints on the other work tapes. Sudborough proved that the class of languages accepted by non-deterministic AuxPDA in polynomial time and logarithmic space on the work tapes is LOGCFL [S].

We can easily construct a NAuxPDA for D_2LREACH(acyclic) working basically like a non-deterministic logspace TM for the reachability problem, i.e., it guesses a path from s to t. When using an edge labeled with a or b, this symbol is put on the

pushdown store. Whenever an edge labeled $\bar{a}$ or $\bar{b}$ is used, it checks the top symbol on the pushdown store and removes it if it fits. Otherwise it rejects immediately. It accepts only if t is reached with empty pushdown store.

Completeness can be shown by a reduction similar to the one in [GHR] for the P-completeness of D_2LREACH. Let L be a language in LOGCFLand M a NAuxPDA for L using logarithmic space on its work tapes and accepting in polynomial time. Without loss of generality we may assume that M uses just the symbols a and b on its pushdown store and has a single accepting configuration. Furthermore, M accepts only if the pushdown store is empty.

We will now use M to reduce L to D_2LREACH(acyclic) by constructing a labeled graph G_x, where $x = x_1 \cdots x_n$ is an input to M. The nodes of G_x are the configurations of M with an additional time stamp but without considering the pushdown store. Each node can be represented by a tuple (p, i, k, t), where p is a state of M, i is a position on M's input tape, and k is a representation of the work tapes of M. The time stamp t is needed to obtain an acyclic graph. Observe that there is just a polynomial number of configurations.

We add an edge from node (p, i, k, t) to node $(p', i', k', t+1)$ labeled with $\alpha \in \{a, b, \bar{a}, \bar{b}\}^+$ if and only if M can make a step from configuration (p, i, k) to (p', i', k') removing $\sigma \in \{a, b\}$ from the pushdown store and pushing $\beta \in \{a, b\}^*$, where $\alpha = \bar{\sigma}\beta$. Whenever an edge is labeled by a string α of length more than one, new nodes are added and α is distributed over several edges.

Finally, let s be the initial configuration and t the unique accepting configuration of M. Thus, G_x is in D_2LREACH(acyclic) if and only if x is accepted by M, i.e., we have reduced L to D_2LREACH(acyclic).

The construction above can be carried out by a logspace transducer. Therefore, we have proved that D_2LREACH(acyclic) is complete for LOGCFLunder logspace reductions. □

We represent D_2LREACH(acyclic) as a set of structures over vocabulary $\langle R^2_a, R^2_{\bar{a}}, R^2_b, R^2_{\bar{b}}, s, t\rangle$, i.e., for each symbol $\sigma \in \Sigma$ there is a binary relation R^2_σ and we allow insertion and deletion of tuples in these relations as well as setting the constants.

We have to be a bit careful with the insertion operation, as we want the graph to remain acyclic and every edge to carry a unique label. Hence, the semantics of a insertion operation is to insert an edge, if it is not already present (with another label) and does not cause a cycle. Both conditions can be checked by a first-order formula, therefore we will ignore them in the following to simplify presentation. We call the resulting dynamic problem dynD_2LREACH(acyclic). It should be noted that, besides the restriction on insertions, dynD_2LREACH(acyclic) is the canonical dynamic problem of D_2LREACH(acyclic).

We can now state the main theorem of this section.

Theorem 7.2. dynD_2LREACH(acyclic) *is in* Dyn(LOGCFL, FO).

We sketch the proof in the remainder of this section. First, we introduce the auxiliary data structure we are going to use. Insertion and deletion of edges is considered afterward.

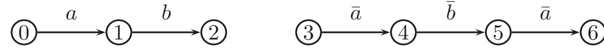

Fig. 2. Example of a labeled graph.

7.1. *An Auxiliary Data Structure*

One might be tempted to use as auxiliary structure the set of pairs (u, v) such that there is a D_2-labeled path from u to v. As the example in Figure 2 indicates, this information is not sufficient. In this example, the data structure would contain only the tuples (u, u) for $u \in \{0, \ldots, 6\}$. This does not help to recognize new paths, e.g., the path from 0 to 6 after insertion of an edge $(2, 3)$ labeled a.

A more helpful information is that the concatenation of the labels on the paths from 0 to 2 and from 4 to 6 is in D_2. This is exactly the kind of information we are going to store. More precisely, we maintain a relation P of arity four which contains the tuples (u_1, v_1, u_2, v_2) such that there are paths from u_1 to v_1 labeled s_1 and from u_2 to v_2 labeled s_2 with $s_1 s_2 \in D_2$.

We also maintain the transitive closure T of the graph, ignoring the labels. Therefore, our auxiliary data structure is over vocabulary

$$\tau = \langle R_a^2, R_b^2, R_{\bar{a}}^2, R_{\bar{b}}^2, P^4, T^2, s, t\rangle.$$

The relation P stores information about concatenations of labels of two paths. To update P during insertion and deletion of edges, we will need the corresponding information for three or even four paths. Fortunately, this information can be extracted from P by a first-order formula.

Lemma 7.3. *For every $k \geq 1$, there is a first-order formula π_k over vocabulary τ, such that $\pi_k(u_1, v_1, \ldots, u_k, v_k)$ holds if and only if there are paths from u_i to v_i labeled with strings s_i for $i \leq k$, respectively, and $s_1 \cdots s_k \in D_2$.*

Proof. The proof is by induction on k. Of course, we can take $\pi_1(u_1, v_1) = P(u_1, v_1, u_1, u_1)$ and $\pi_2(u_1, v_1, u_2, v_2) = P(u_1, v_1, u_2, v_2)$.

Let $p_1, \ldots, p_k$ be paths in a graph G such that the concatenation $s_1 \cdots s_k$ of their label sequences is in D_2. To construct π_k for $k > 2$ we will distinguish two cases. Either for every a and b in s_1 the corresponding $\bar{a}$ or $\bar{b}$ is in s_1 or in s_2 or there is a symbol in s_1 with corresponding closing symbol in some s_i, $i > 2$.

In the first case, s_2 can be split into $w_1 w_2$ such that w_1 ends[6] with the last closing symbols corresponding to a symbol in s_1 and $s_1 w_1$ and $w_2 s_3 \cdots s_k$ are in D_2. This can be checked by the following formula using π_{k-1}:

$$\exists x (\pi_2(u_1, v_1, u_2, x) \wedge \pi_{k-1}(x, v_2, u_3, v_3, \ldots, u_k, v_k)). \tag{1}$$

In the second case, let σ be the last opening symbol in s_1 whose corresponding symbol is in a substring s_i with $i > 2$. Hence, $s_1 = w_1 \sigma w_2$ and $s_i = w_3 \bar{\sigma} w_4$ and we

[6] Observe that $w_1 = \varepsilon$ covers the case where all closing symbols for s_1 are already in s_1.

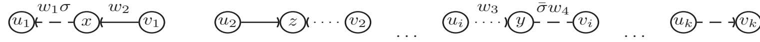

Fig. 3. The paths considered by formula (2).

must have $w_1\sigma\bar{\sigma}w_4s_{i+1}\cdots s_k \in D_2$ and $w_2s_2\cdots s_{i-1}w_3 \in D_2$. The first property can be checked by π_{k-i+2} where $k-i+2 < k$. However, the second expression might have k factors, so we have to split it again. By the choice of σ, for every a and b in w_2 the corresponding $\bar{a}$ or $\bar{b}$ is either in w_2 or in s_2. Thus, we can treat $w_2s_2\cdots s_{i-1}w_3$ like in the first case resulting in the following formula illustrated in Figure 3:

$$\exists x, y, z(\pi_{k-i+2}(u_1, x, y, v_i, \dots, u_k, v_k) \wedge \pi_2(x, v_1, u_2, z) \wedge \pi_{i-1}(z, v_2, \dots, u_i, y)). \tag{2}$$

We observe that by setting i to k and assigning u_1 to x and v_k to y in formula (2), we get formula (1). Consequently, we use only formula (2) to construct π_k:

$$\begin{aligned}
&\pi_k(u_1, v_1, \dots, u_k, v_k)\\
&\quad\equiv \exists x, y, z(\pi_{k-1}(u_1, x, y, v_3, \dots, u_k, v_k)\\
&\qquad\qquad \wedge \pi_2(x, v_1, u_2, z) \wedge \pi_2(z, v_2, u_3, y))\\
&\quad\vee \exists x, y, z(\pi_{k-2}(u_1, x, y, v_4, \dots, u_k, v_k)\\
&\qquad\qquad \wedge \pi_2(x, v_1, u_2, z) \wedge \pi_2(z, v_2, \dots, u_4, y))\\
&\quad\vdots\\
&\quad\vee \exists x, y, z(\pi_3(u_1, x, y, v_{k-1}, u_k, v_k) \wedge \pi_2(x, v_1, u_2, z)\\
&\qquad\qquad \wedge \pi_{k-2}(z, v_2, \dots, u_{k-1}, y))\\
&\quad\vee \exists x, y, z(\pi_2(u_1, x, y, v_k) \wedge \pi_2(x, v_1, u_2, z) \wedge \pi_{k-1}(z, v_2, \dots, u_k, y)).
\end{aligned}$$

It can be easily verified that π_k also fulfills the “only if” part of the lemma. □

7.2. *Inserting Edges*

After insertion of an edge (x, y) tuples might have to be added to P. Of course any new tuple must depend on a path through (x, y). As such a tuple involves two paths, there are three cases that are covered by the following lemma: the edge (x, y) may be used in only the first, in only the second, or in both paths.

Lemma 7.4. *For each $\sigma \in \Sigma$ there are* FO*-formulas $\varphi_{\sigma 1}$, $\varphi_{\sigma 2}$, and φ_σ, such that*

- *$\varphi_{\sigma 1}(u_1, v_1, u_2, v_2, x, y)$ is true iff there are paths from u_1 to x labeled s_1, from y to v_1 labeled s_2, and from u_2 to v_2 labeled s_3 such that $s_1\sigma s_2s_3$ is in D_2.*
- *$\varphi_{\sigma 2}(u_1, v_1, u_2, v_2, x, y)$ is true iff there are paths from u_1 to v_1 labeled s_1, from u_2 to x labeled s_2, and from y to v_2 labeled s_3 such that $s_1s_2\sigma s_3$ is in D_2.*

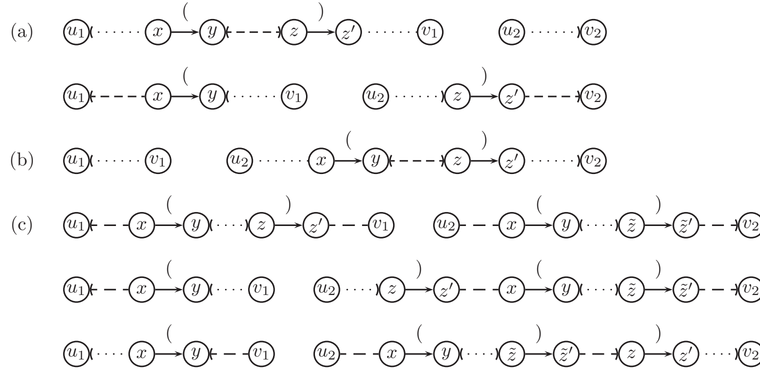

Fig. 4. The different situations occurring at the insertion of edges labeled by opening parentheses.

- *$\varphi_\sigma(u_1, v_1, u_2, v_2, x, y)$ is true iff there are paths from u_1 to x labeled s_1, from y to v_1 labeled s_2, from u_2 to x labeled s_3, and from y to v_2 labeled s_4 such that $s_1 \sigma s_2 s_3 \sigma s_4$ is in D_2.*

Proof. All three formulas are defined in a similar way: They guess the edge corresponding to (x, y) and check for paths such that the labels between and outside of these edges form words in D_2. Of course, each formula depends on whether σ is an opening or a closing symbol, but it is straightforward that the formulas are symmetric in that respect. Furthermore, whether $\sigma = a$ or $\sigma = b$ does not make a significant difference. Thus, we only consider the case $\sigma = a$.

The reasoning in constructing $\varphi_{a,1}$ and $\varphi_{a,2}$ is similar to that in Lemma 7.3. The formula expresses that there is an edge (z, z') labeled $\bar{a}$ corresponding to a in the concatenation of two paths from u_1 to v_1 and from u_2 to v_2 which combine to a string in D_2.

For $\varphi_{a,1}$, there are two cases illustrated in Figure 4(a): (z, z') might be on the path from u_1 to v_1 behind (x, y) or on the path from u_2 to v_2. These considerations lead to the following formula:

$$\begin{aligned}\varphi_{a,1}&(u_1, v_1, u_2, v_2, x, y)\\ &\equiv \exists z, z'(R_{\bar{a}}(z, z') \wedge [(\pi_3(u_1, x, z', v_1, u_2, v_2) \wedge \pi_1(y, z))\\ &\qquad\qquad \vee (P(u_1, x, z', v_2) \wedge P(y, v_1, u_2, z))]).\end{aligned}$$

As illustrated in Figure 4(b), $\varphi_{a,2}$ has to deal with only one possible place for (z, z'):

$$\varphi_{a,2}(u_1, v_1, u_2, v_2, x, y) \equiv \exists z, z'(R_{\bar{a}}(z, z') \wedge [\pi_3(u_1, v_1, u_2, x, z', v_2) \wedge \pi_1(y, z)]).$$

The construction of φ_a is a bit more complicated, as two edges (z, z') and $(\tilde{z}, \tilde{z}')$ are needed, corresponding to the two occurrences of (x, y). There are three possibilities to

order these edges (Figure 4(c)), resulting in the following formula:

$$\begin{aligned}
&\varphi_a(u_1, v_1, u_2, v_2, x, y)\\
&\quad\equiv \exists z, z', \tilde{z}, \tilde{z}'(R_{\bar{a}}(z, z') \wedge R_{\bar{a}}(\tilde{z}, \tilde{z}')\\
&\qquad\qquad \wedge [(\pi_1(y, z) \wedge \pi_1(y, \tilde{z}) \wedge \pi_4(u_1, x, z', v_1, u_2, x, \tilde{z}', v_2))\\
&\qquad\qquad \vee (\pi_1(y, \tilde{z}) \wedge P(y, v_1, u_2, z) \wedge \pi_3(u_1, x, z', x, \tilde{z}', v_2))\\
&\qquad\qquad \vee (\pi_1(y, \tilde{z}) \wedge \pi_3(y, v_1, u_2, x, \tilde{z}', z) \wedge P(u_1, x, z', v_2))]).
\end{aligned}$$

If (x, y) is labeled b, the formulas $\varphi_{b,1}$, $\varphi_{b,2}$, and φ_b can be obtained by replacing $R_{\bar{a}}$ with $R_{\bar{b}}$.

The edge (x, y) might also be labeled $\bar{a}$ or $\bar{b}$. The formulas for these cases are constructed similarly:

$$\begin{aligned}
&\varphi_{\bar{a},1}(u_1, v_1, u_2, v_2, x, y)\\
&\quad\equiv \exists x', y'(R_a(x', y') \wedge [\pi_1(y', x) \wedge \pi_3(u_1, x', y, v_1, u_2, v_2)]),
\end{aligned}$$

$$\begin{aligned}
&\varphi_{\bar{a},2}(u_1, v_1, u_2, v_2, x, y)\\
&\quad\equiv \exists x', y'(R_a(x', y')\\
&\qquad \wedge [(P(u_1, x', y, v_2) \wedge P(y', v_1, u_2, x)) \vee (\pi_1(y', x)\\
&\qquad\quad \wedge \pi_3(u_1, v_1, u_2, x', y, v_2))]),
\end{aligned}$$

$$\begin{aligned}
&\varphi_{\bar{a}}(u_1, v_1, u_2, v_2, x, y)\\
&\quad\equiv \exists z, z', \tilde{z}, \tilde{z}'(R_a(z, z') \wedge R_a(\tilde{z}, \tilde{z}')\\
&\qquad\qquad \wedge [(\pi_1(z', x) \wedge \pi_1(\tilde{z}', x) \wedge \pi_4(u_1, z, y, v_1, u_2, \tilde{z}, y, v_2))\\
&\qquad\qquad \vee (\pi_1(z', x) \wedge P(\tilde{z}', v_1, u_2, x) \wedge \pi_3(u_1, z, y, \tilde{z}, y, v_2))\\
&\qquad\qquad \vee (\pi_1(\tilde{z}', , x) \wedge \pi_3(z', \tilde{z}, y, v_1, u_2, x) \wedge P(u_1, z, y, v_2))]). \qquad \square
\end{aligned}$$

Altogether, a tuple is in P after inserting (x, y) if it was in P before or if one of the three formulas witnesses that it newly belongs to P. Thus, we get the update formulas for $\texttt{insert}_{R_\sigma}(x, y)$ as follows:

$$\begin{aligned}
P'(u_1, v_1, u_2, v_2) &\equiv P(u_1, v_1, u_2, v_2) \vee \varphi_{\sigma 1}(u_1, v_1, u_2, v_2, x, y)\\
&\qquad \vee \varphi_{\sigma 2}(u_1, v_1, u_2, v_2, x, y) \vee \varphi_\sigma(u_1, v_1, u_2, v_2, x, y)\\
T'(u, v) &\equiv \exists x, y(T(u, v) \vee [T(u, x) \wedge T(y, v)])
\end{aligned}$$

7.3. *Deleting Edges*

Maintaining P and T under deletion of edges is more complicated. Whenever the edge to be deleted is used on a path, we have to check whether there is a path that avoids the edges. We basically need the formulas described in the following lemma.

Lemma 7.5. *For each $\sigma \in \Sigma$ there are* FO*-formulas $\psi_{\sigma 1}$, $\psi_{\sigma 2}$, and ψ_σ, such that*

- $\psi_{\sigma 1}(u_1, v_1, u_2, v_2, x, y)$ *expresses the following implication: If there is an edge (x, y) labeled σ and $\varphi_{\sigma 1}(u_1, v_1, u_2, v_2, x, y)$ is true, then there is a path from u_1 to v_1 that does not use (x, y) and a path from u_2 to v_2 so that the concatenation of their labels is in D_2.*
- $\psi_{\sigma 2}(u_1, v_1, u_2, v_2, x, y)$ *expresses the following implication: If there is an edge (x, y) labeled σ and $\varphi_{\sigma 2}(u_1, v_1, u_2, v_2, x, y)$ is true, then there is a path from u_1 to v_1 and a path from u_2 to v_2 that does not use (x, y) so that the concatenation of their labels is in D_2.*
- $\psi_\sigma(u_1, v_1, u_2, v_2, x, y)$ *is the following implication: If there is an edge (x, y) labeled σ and $\varphi_{\sigma 1}(u_1, v_1, u_2, v_2, x, y) \wedge \varphi_{\sigma 2}(u_1, v_1, u_2, v_2, x, y)$ is true, then there are paths from u_1 to v_1 and from u_2 to v_2 that do not use (x, y) so that the concatenation of their labels is in D_2.*

Proof. Again, we only consider the case $\sigma = a$ as the other three cases are either almost identical or symmetric.

To build ψ_{a1}, we have to describe a path from u_1 to v_1 not using (x, y). To this end, we make use of a technique from [PI]. Such a path exists if there is an edge (z, z') different from (x, y), such that there are a path from u_1 to z, a path from z' to v_1, and a path from z to x but no path from z' to x. In our context, we also need that the concatenation of labels along the path from u_1 via (z, z') to v_1 and a path from u_2 to v_2 is in D_2. This can be done by $\varphi_{\rho 1}$, where ρ is the label of (z, z'). Since we do not know ρ, we have to consider all four possibilities:

$$\begin{aligned}
\psi_{a,1}(u_1, v_1, u_2, v_2, x, y) \equiv{} & (R_a(x, y) \wedge \varphi_{a,1}(u_1, v_1, u_2, v_2, x, y)) \\
& \rightarrow (\exists z, z'[T(u_1, z) \wedge T(z, x) \wedge \varphi_E(z, z') \wedge \neg T(z', x) \\
& \quad \wedge T(z', v_1) \wedge (z \neq x \vee z' \neq y) \\
& \quad \wedge ((R_a(z, z') \wedge \varphi_{a,1}(u_1, v_1, u_2, v_2, z, z')) \\
& \quad \vee (R_{\bar{a}}(z, z') \wedge \varphi_{\bar{a},1}(u_1, v_1, u_2, v_2, z, z')) \\
& \quad \vee (R_b(z, z') \wedge \varphi_{b,1}(u_1, v_1, u_2, v_2, z, z')) \\
& \quad \vee (R_{\bar{b}}(z, z') \wedge \varphi_{\bar{b},1}(u_1, v_1, u_2, v_2, z, z')))]).
\end{aligned}$$

Here, $\varphi_E(z, z')$ expresses that there is an edge from z to z'.

The formula $\psi_{a,2}$ is completely analogous:

$$\begin{aligned}
\psi_{a,2}(u_1, v_1, u_2, v_2, x, y) \equiv{} & (R_a(x, y) \wedge \varphi_{a,2}(u_1, v_1, u_2, v_2, x, y)) \\
& \rightarrow (\exists z, z'[T(u_2, z) \wedge T(z, x) \wedge \varphi_E(z, z') \wedge \neg T(z', x) \\
& \quad \wedge T(z', v_2) \wedge (z \neq x \vee z' \neq y) \\
& \quad \wedge ((R_a(z, z') \wedge \varphi_{a,2}(u_1, v_1, u_2, v_2, z, z')) \\
& \quad \vee (R_{\bar{a}}(z, z') \wedge \varphi_{\bar{a},2}(u_1, v_1, u_2, v_2, z, z')) \\
& \quad \vee (R_b(z, z') \wedge \varphi_{b,2}(u_1, v_1, u_2, v_2, z, z')) \\
& \quad \vee (R_{\bar{b}}(z, z') \wedge \varphi_{\bar{b},2}(u_1, v_1, u_2, v_2, z, z')))]).
\end{aligned}$$

Let us now turn to ψ_a. We have to avoid (x, y) on the path from u_1 to v_1 and between u_2 and v_2. This is done as before, but possibly by two different edges (z, z') and $(\tilde{z}, \tilde{z}')$ labeled ρ and $\tilde{\rho}$. We have to find two proper labeled paths using these edges. If this can be expressed by a formula ζ, we obtain ψ_a as before:

$$\begin{aligned}\psi_a(u_1, v_1, u_2, v_2, x, y) \equiv\ & (R_a(x, y) \wedge \varphi_{a,1}(u_1, v_1, u_2, v_2, x, y)\\ & \wedge \varphi_{a,2}(u_1, v_1, u_2, v_2, x, y))\\ & \to (\exists z, z'\tilde{z}, \tilde{z}'[T(u_1, z) \wedge \varphi_E(z, z') \wedge T(z', v_1) \wedge T(z, x)\\ & \wedge \neg T(z', x) \wedge (z \neq x \vee z' \neq y) \wedge T(u_2, \tilde{z})\\ & \wedge \varphi_E(\tilde{z}, \tilde{z}') \wedge T(\tilde{z}', v_2) \wedge T(\tilde{z}, x) \wedge \neg T(\tilde{z}', x)\\ & \wedge (\tilde{z} \neq x \vee \tilde{z}' \neq y)\\ & \wedge \zeta(u_1, v_1, u_2, v_2, z, z', \tilde{z}, \tilde{z}')]).\end{aligned}$$

To build ζ we distinguish the cases given by the possible labelings of (z, z') and $(\tilde{z}, \tilde{z}')$. We define a formula for every case and get ζ as their disjunction:

$$\begin{aligned}\zeta \equiv\ & \zeta_{aa} \vee \zeta_{a\bar{a}} \vee \zeta_{\bar{a}a} \vee \zeta_{\bar{a}\bar{a}} \vee \zeta_{ab} \vee \zeta_{a\bar{b}} \vee \zeta_{\bar{a}b} \vee \zeta_{\bar{a}\bar{b}} \vee \zeta_{ba} \vee \zeta_{b\bar{a}} \vee \zeta_{\bar{b}a} \vee \zeta_{\bar{b}\bar{a}}\\ & \vee \zeta_{bb} \vee \zeta_{b\bar{b}} \vee \zeta_{\bar{b}b} \vee \zeta_{\bar{b}\bar{b}}.\end{aligned}$$

For $\rho, \tilde{\rho} \in \{a, b, \bar{a}, \bar{b}\}$, we want $\zeta_{\rho\tilde{\rho}}(u_1, v_1, u_2, v_2, z, z', \tilde{z}, \tilde{z}')$ to hold if and only if there are edges (z, z') and $(\tilde{z}, \tilde{z}')$ labeled ρ and $\tilde{\rho}$ and paths from u_1 to z labeled s_1, from z' to v_1 labeled s_2, from u_2 to $\tilde{z}$ labeled s_3, and from $\tilde{z}'$ to v_2 labeled s_4 such that $s_1\rho s_2 s_3 \tilde{\rho} s_4 \in D_2$.

As in the case of inserting edges, the formulas guess the corresponding edges and check whether there are proper labeled paths.

We consider four cases. The first is that both edges, (z, z') and $(\tilde{z}, \tilde{z}')$, are labeled a or b. By the position of the corresponding edges we can distinguish the three cases shown in Figure 5(a). We give ζ_{aa} as an example, to obtain ζ_{ab}, ζ_{ba}, and ζ_{bb} only the respective relation symbols have to be replaced:

$$\begin{aligned}& \zeta_{aa}(u_1, v_1, u_2, v_2, z, z', \tilde{z}, \tilde{z}')\\ & \quad \equiv R_a(z, z') \wedge R_a(\tilde{z}, \tilde{z}')\\ & \quad\quad \wedge \exists c, d, e, f(R_{\bar{a}}(c, d) \wedge R_{\bar{a}}(e, f)\\ & \quad\quad \wedge [(\pi_1(z', c) \wedge \pi_1(\tilde{z}', e) \wedge \pi_4(u_1, z, d, v_1, u_2, \tilde{z}, f, v_2))\\ & \quad\quad \vee (P(z', v_1, u_2, c) \wedge \pi_1(\tilde{z}', e) \wedge \pi_3(u_1, z, d, \tilde{z}, f, v_2))\\ & \quad\quad \vee (\pi_1(\tilde{z}', e) \wedge P(u_1, z, d, v_2) \wedge \pi_3(z', v_1, u_2, \tilde{z}, f, c))]).\end{aligned}$$

The second case is $\zeta_{a\bar{a}}$. It can be built similarly. However, we have to distinguish more cases (Figure 5(b)). A special case is that (z, z') and $(\tilde{z}, \tilde{z}')$ might be corresponding

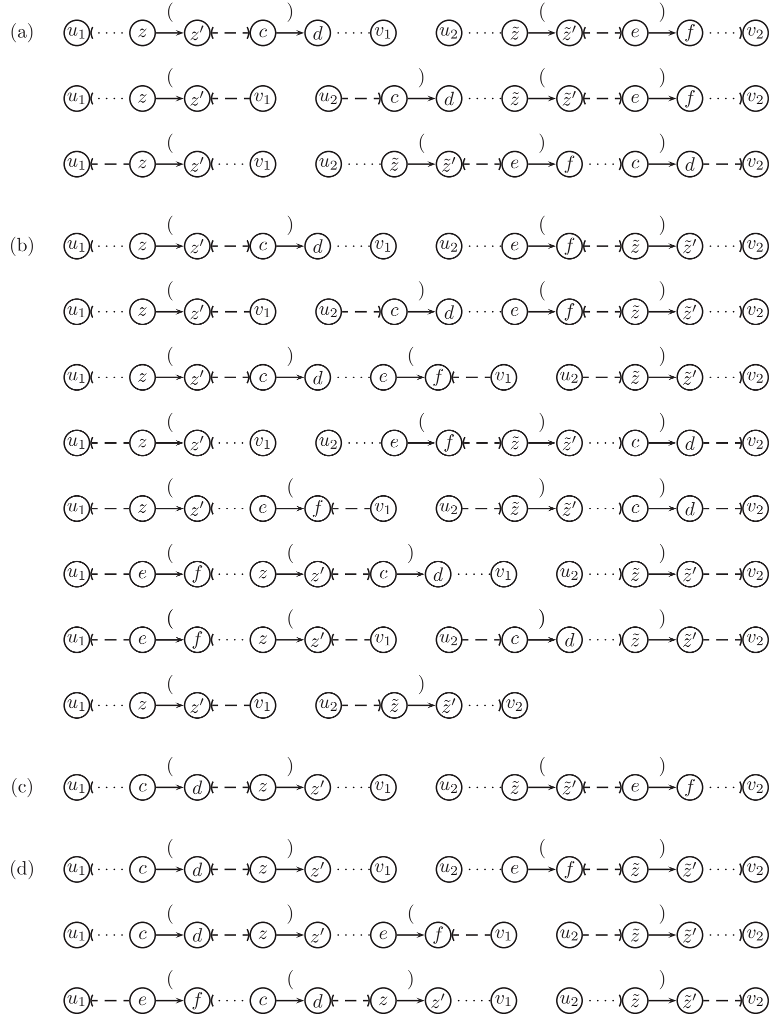

Fig. 5. The different possible positions of the edges to delete, (z, z') and $(\tilde{z}, \tilde{z}')$, and their corresponding edges as considered in the construction of ζ.

edges:

$$\begin{aligned}
&\zeta_{a\bar{a}}(u_1, v_1, u_2, v_2, z, z', \tilde{z}, \tilde{z}') \\
&\quad\equiv\ R_a(z, z') \wedge R_{\bar{a}}(\tilde{z}, \tilde{z}') \\
&\qquad \wedge \exists c, d, e, f\, (R_{\bar{a}}(c, d) \wedge R_a(e, f) \\
&\qquad \wedge [(\pi_1(z', c) \wedge \pi_1(f, \tilde{z}) \wedge \pi_4(u_1, z, d, v_1, u_2, e, \tilde{z}', v_2)) \\
&\qquad \vee (P(z', v_1, u_2, c) \wedge \pi_1(f, \tilde{z}) \wedge \pi_3(u_1, z, d, e, \tilde{z}', v_2)) \\
&\qquad \vee (\pi_1(z', c) \wedge P(f, v_1, u_2, \tilde{z}) \wedge \pi_3(u_1, z, d, e, \tilde{z}', v_2))
\end{aligned}$$

$$\vee (\pi_1(f, \tilde{z}) \wedge \pi_3(z', v_1, u_2, e, \tilde{z}', c) \wedge P(u_1, z, d, v_2))$$
$$\vee (P(f, v_1, u_2, \tilde{z}) \wedge P(z', e, \tilde{z}', c) \wedge P(u_1, z, d, v_2))$$
$$\vee (\pi_1(z', c) \wedge \pi_3(f, z, d, v_1, u_2, \tilde{z}) \wedge P(u_1, e, \tilde{z}', v_2))$$
$$\vee (P(z', v_1, u_2, c) \wedge P(f, z, d, \tilde{z}) \wedge P(u_1, e, \tilde{z}', v_2))$$
$$\vee (P(z', v_1, u_2, \tilde{z}) \wedge P(u_1, z, \tilde{z}', v_2))]).$$

$\zeta_{b\bar{b}}$ is again similar to $\zeta_{a\bar{a}}$, to build $\zeta_{a\bar{b}}$ and $\zeta_{b\bar{a}}$ we have to drop the last disjunct because the two edges cannot be corresponding.

For $\zeta_{\bar{a}a}$, and as well for $\zeta_{\bar{a}b}$, $\zeta_{\bar{b}a}$, and $\zeta_{\bar{b}b}$, there is only one case to consider, as illustrated in Figure 5(c):

$$\begin{aligned}\zeta_{\bar{a}a}(u_1, v_1, u_2, v_2, z, z', \tilde{z}, \tilde{z}') \equiv\ & R_{\bar{a}}(z, z') \wedge R_a(\tilde{z}, \tilde{z}') \\ & \wedge \exists c, d, e, f [R_a(c, d) \wedge R_{\bar{a}}(e, f) \\ & \wedge \pi_1(d, z) \wedge \pi_1(\tilde{z}', e) \\ & \wedge \pi_4(u_1, c, z', v_1, u_2, \tilde{z}, f, v_2)].\end{aligned}$$

Finally, both edges may be labeled $\bar{a}$ or $\bar{b}$. The four formulas are again similar, corresponding to the cases shown in Figure 5(d). We spell out $\zeta_{\bar{a}\bar{a}}$:

$$\begin{aligned}&\zeta_{\bar{a}\bar{a}}(u_1, v_1, u_2, v_2, z, z', \tilde{z}, \tilde{z}') \\ &\quad \equiv R_{\bar{a}}(z, z') \wedge R_{\bar{a}}(\tilde{z}, \tilde{z}') \wedge \exists c, d, e, f (R_a(c, d) \wedge R_a(e, f) \\ &\quad\quad \wedge [(\pi_1(d, z)) \wedge \pi_1(f, \tilde{z}) \wedge \pi_4(u_1, c, z', v_1, u_2, e, \tilde{z}', v_2)) \\ &\quad\quad \vee (\pi_1(d, z) \wedge P(f, v_1, u_2, \tilde{z}) \wedge \pi_3(u_1, c, z', e, \tilde{z}', v_2) \\ &\quad\quad \vee (\pi_1(d, z) \wedge \pi_3(f, c, z', v_1, u_2, \tilde{z}) \wedge P(u_1, e, \tilde{z}', v_2)]). \qquad \square\end{aligned}$$

Consequently, the updates necessary for an operation $\mathtt{delete}_{R_\sigma}(x, y)$ for $\sigma \in \Sigma$ can be described as follows, concluding the proof of Theorem 7.2:

$$\begin{aligned}P'(u_1, v_1, u_2, v_2) \equiv\ & P(u_1, v_1, u_2, v_2) \wedge \psi_{\sigma 1}(u_1, v_1, u_2, v_2, x, y) \\ & \wedge \psi_{\sigma 2}(u_1, v_1, u_2, v_2, x, y) \wedge \psi_\sigma(u_1, v_1, u_2, v_2, x, y) \\ T'(u, v) \equiv\ & T(u, v) \wedge (\neg T(u, x) \vee \neg T(y, v) \vee \exists z, z' [T(u, z) \wedge \varphi_E(z, z') \\ & \wedge T(z', v) \wedge T(z, x) \wedge \neg T(z', x) \wedge (z \neq x \vee z' \neq y)]).\end{aligned}$$

We end this section with the following corollary, which holds because the auxiliary data structure for the empty graph can be described in first order.

Corollary 7.6. dynD_2LREACH(acyclic) *is in* DynFO.

8. Conclusion

We have taken a step toward a dynamic complexity theory by presenting a more accurate notion of dynamic problems and complexity classes. This allowed us to characterize the complexity of several dynamic problems more precisely, thus clarifying the role of precomputation in dynamic complexity, which was an open problem in [PI]. We presented a useful kind of reductions and gave complete problems for dynamic complexity classes under these reductions. Finally, we presented first-order updates to a first "non-redundant" LOGCFL-complete problem.

We want to give some directions for further research:

- It remains open whether there is a non-redundant problem complete for Pthat allows efficient updates. D_2LREACH might be a candidate. Note that the result in [H] cannot be applied to D_2LREACH in a straightforward way since one has to consider paths of exponential length.
- As stated in [HI], canonical dynamic problems cannot be complete for dynamic complexity classes in general. Therefore, it might be interesting to look for complete problems for classes of canonical dynamic problems.
- A further issue is to establish connections between algorithmic results on dynamic problems and dynamic complexity theory.

References

[BIS] David A. Mix Barrington, Neil Immerman, and Howard Straubing. On uniformity within nc. *Journal of Computer and System Sciences*, 41(3):274–306, 1990.

[BRS] Allan Borodin, Alexander A. Razborov, and Roman Smolensky. On lower bounds for read-k-times branching programs. *Computational Complexity*, 3:1–18, 1993.

[C] Stephen A. Cook. Characterizations of pushdown machines in terms of time-bounded computers. *Journal of the ACM*, 18(1):4–18, 1971.

[DLW] Guozhu Dong, Leonid Libkin, and Limsoon Wong. Incremental recomputation in local languages. *Information and Computation*, 181(2):88–98, 2003.

[DS1] Guozhu Dong and Jianwen Su. First-order incremental evaluation of datalog queries. In *Proc. of DBPL-*4, Workshops in Computing, pages 295–308. Springer, Berlin, 1993.

[DS2] Guozhu Dong and Jianwen Su. Incremental and decremental evaluation of transitive closure by first-order queries. *Information and Computation*, 120(1):101–106, 1995.

[DS3] Guozhu Dong and Jianwen Su. Deterministic FOIES are strictly weaker. *Annals of Mathematics and Artificial Intelligence*, 19(1-2):127–146, 1997.

[DS4] Guozhu Dong and Jianwen Su. Arity bounds in first-order incremental evaluation and definition of polynomial time database queries. *Journal of Computer and System Sciences*, 557(3):289–308, 1998.

[E] Kousha Etessami. Dynamic tree isomorphism via first-order updates to a relational database. In *Proc. of the* 17*th PODS*, pages 235–243. ACM Press, New York, 1998.

[FW] Amos Fiat and Gerhard J. Woeginger, editors. *Online Algorithms*, *The State of the Art*, volume 1442 of LNCS. Springer, Berlin, 1998.

[GHR] Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. *Limits to Parallel Computation*: *P-Completeness Theory*. Oxford University Press, Oxford, 1995.

[GLS] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Computing LOGCFL certificates. *Theoretical Computer Science*, 270(1-2):761–777, 2002.

[H] William Hesse. The dynamic complexity of transitive closure is in $DynTC^0$. *Theoretical Computer Science*, 296(3):473–485, 2003.

[HdLT] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. In *Proc. of the* 30*th ACM STOC*, pages 79–89, 1998.

[HI] William Hesse and Neil Immerman. Complete problems for dynamic complexity classes. In *Proc. of the* 17*th LICS*, pages 313–322. IEEE Computer Society Press, Los Alamitos, CA, 2002.

[I1] Neil Immerman. Languages that capture complexity classes. *SIAM Journal on Computing*, 16(4):760–778, 1987.

[I2] Neil Immerman. *Descriptive Complexity*. Graduate Texts in Computer Science. Springer, New York, 1999.

[L] Leonid Libkin. *Elements of Finite Model Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, New York, 2004.

[LW1] Leonid Libkin and Limsoon Wong. Incremental recomputation of recursive queries with nested sets and aggregate functions. In *Proc. of DBPL*-6, volume 1369 of LNCS, pages 222–238. Springer, Berlin, 1998.

[LW2] Leonid Libkin and Limsoon Wong. On the power of incremental evaluation in SQL-like languages. In *Proc. of DBPL*-7, volume 1949 of LNCS, pages 17–30. Springer, Berlin, 2000.

[MSVT] Peter Bro Miltersen, Sairam Subramanian, Jeffrey Scott Vitter, and Roberto Tamassia. Complexity models for incremental computation. *Theoretical Computer Science*, 130(1):203–236, 1994.

[PI] Sushant Patnaik and Neil Immerman. Dyn-FO: A parallel, dynamic complexity class. *Journal of Computer and System Sciences*, 55(2):199–209, 1997.

[R] Omer Reingold. Undirected st-connectivity in log-space. In *Proc. of the* 37*th ACM STOC*, pages 376–385, 2005.

[RZ] Liam Roditty and Uri Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *Proc. of the* 36*th ACM STOC*, pages 184–191, 2004.

[S] Ivan Hal Sudborough. On the tape complexity of deterministic context-free languages. *Journal of the ACM*, 25(3):405–414, 1978.

[V] Heribert Vollmer. *Introduction to Circuit Complexity - A Uniform Approach*. Texts in Theoretical Computer Science. An EATCS Series. Springer, New York, 1999.

Received in final form March 21, 2006. *Online publication March* 1, 2007.