2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)

Dynamic Matrix Inverse: Improved Algorithms and Matching Conditional Lower Bounds

Jan van den Brand
KTH Royal Institute of Technology
Stockholm, Sweden

Danupon Nanongkai
KTH Royal Institute of Technology
Stockholm, Sweden

Thatchaphol Saranurak
Toyota Technological Institute at Chicago
Chicago, USA

***Abstract*—The dynamic matrix inverse problem is to maintain the inverse of a matrix undergoing element and column updates. It is the main subroutine behind the best algorithms for many dynamic problems whose complexity is not yet well-understood, such as maintaining the largest eigenvalue, rank and determinant of a matrix and maintaining reachability, distances, maximum matching size, and k-paths/cycles in a graph. Understanding the complexity of dynamic matrix inverse is a key to understand these problems.**

In this paper, we present (i) improved algorithms for dynamic matrix inverse and their extensions to some incremental/look-ahead variants, and (ii) variants of the Online Matrix-Vector conjecture [Henzinger et al. STOC'15] that, if true, imply that these algorithms are tight. Our algorithms automatically lead to faster dynamic algorithms for the aforementioned problems, some of which are also tight under our conjectures, e.g. reachability and maximum matching size (closing the gaps for these two problems was in fact asked by Abboud and V. Williams [FOCS'14]). Prior best bounds for most of these problems date back to more than a decade ago [Sankowski FOCS'04, COCOON'05, SODA'07; Kavitha FSTTCS'08; Mucha and Sankowski Algorithmica'10; Bosek et al. FOCS'14].

Our improvements stem mostly from the ability to use fast matrix multiplication "one more time", to maintain a certain transformation matrix which could be maintained only combinatorially previously (i.e. without fast matrix multiplication). Oddly, unlike other dynamic problems where this approach, once successful, could be repeated several times ("bootstrapping"), our conjectures imply that this is not the case for dynamic matrix inverse and some related problems. However, when a small additional "look-ahead" information is provided we can perform such repetition to drive the bounds down further.

***Keywords*-data structures, dynamic algorithms, dynamic matrix inverse, determinant, adjoint, reachability, matching**

I. Introduction

In the *dynamic matrix inverse* problem, we want to maintain the inverse of an $n \times n$ matrix A over any field, when A undergoes some updates. There were many variants of this problem considered [2]–[6]: Updates can be *element updates*, where we change the value of one element in A, or *column updates*, where we change the values of all elements in one column.[1] The inverse of A might be maintained explicitly or might be answered through an *element query* or a *row/column query*; the former returns the value of a specified element of the inverse, and the latter answers the values of all elements in a specified row/column of the inverse. The goal is to design algorithms with small *update time* and *query time*, denoting the time needed to handle each update and each query respectively. Time complexity is measured by the number of *field operations*.[2] Variants where elements are polynomials (e.g. [7]–[9]) and where some updates are known ahead of time (the *look-ahead* setting) were also considered (e.g. [10]–[13]).

Dynamic matrix inverse algorithms played a central role in designing algorithms for many dynamic problems such as maintaining matrix and graph properties. Its study can be traced back to the 1950 algorithm of Sherman and Morrison [14] which can be used to maintain the inverse explicitly in $O(n^2)$ time. The previous best bounds are due to Sankowski's FOCS'04 paper [2] and its follow-ups [3], [7], [10] (improving on, e.g., [15]). Their time guarantees depend on how fast we can multiply matrices. For example, with the state-of-the-art matrix multiplication algorithms [16], [17], Sankowski's algorithm [2] can handle an element update and answer an element query for matrix inverse in $O(n^{1.447})$ time. Consequently, the same update time[3] can be guaranteed for, e.g., maintaining largest eigenvalues, ranks and determinants of matrices undergoing entry updates and maintaining maximum matching sizes, reachability between two nodes (*st-reachability*), existences of a directed cycle, numbers of spanning trees, and numbers of paths in directed acyclic graphs (DAGs) undergoing edge insertions and deletions.[4] (Unless specified otherwise, all

The full version of this paper is available as [1] at https://arxiv.org/abs/1905.05067.

[1]There are other kinds of updates which we do not consider in this paper, such as rank-1 updates in [4]–[6].

[2]Later when we consider other kinds of dynamic problems, such as dynamic graphs, the time refer to the standard notion of time in the RAM model.

[3]See Footnote 2.

[4]We note that while the update and query time for the matrix inverse problem is defined to be the number of arithmetic operations, most of time the guarantees translate into the same running time in the RAM model. Exceptions are the numbers of spanning trees in a graph and numbers of paths in a DAG, where the output might be a very big number. In this case the running time is different from the number of arithmetic operations.

2575-8454/19/$31.00 ©2019 IEEE
DOI 10.1109/FOCS.2019.00036

IEEE computer society

mentioned update times are *worst-case*, as opposed to being amortized[5].) See Fig. 1 and 2 for lists of known results for dynamic matrix inverse and Fig. 5 for lists of applications. (There is a second figure with algebraic applications such as determinant, adjoint etc. in the full version.)

Is the $O(n^{1.447})$ bound the best possible for above problems? This kind of question exhibits the current *gap* between existing algorithmic and lower bound techniques and our limited understanding of the power of *algebraic techniques* in designing dynamic algorithms. First of all, despite many successes in the last decade in proving tight bounds for a host of dynamic problems (e.g. [18]–[20]), conditional lower bounds for most of these problems got stuck at $\Omega(n)$ in general. Even for a very special case where the preprocessing time is limited to $o(n^\omega)$ (which is too limited as discussed in Section I-C), the best known conditional lower bound of $\Omega(n^{\omega-1}) = \Omega(n^{1.3728})$ [19] is still not tight ([19] mentioned that "closing this gap is a very interesting open question"). Note that while the upper bounds might be improved in the future with improved rectangular matrix multiplication algorithms, there will still be big gaps even in the *best-possible* scenario: even if there is a *linear-time* rectangular matrix multiplication algorithm, the upper bounds will still be only $O(n^{1+1/3})$, while the lower bound will be $\Omega(n)$.

Secondly, it was shown that *algebraic techniques* – techniques based on fast matrix multiplication algorithms initiated by Strassen [21] – are *inherent* in any upper bound improvements for some of these problems: Assuming the *Combinatorial Boolean Matrix Multiplication (BMM) conjecture*, without algebraic techniques we cannot maintain, e.g., maximum matching size and st-reachability faster than $O(n^2)$ per edge insertion/deletion [19][6]. *Can algebraic techniques lead to faster algorithms that may ideally have update time linear in n? If not, how can we argue lower bounds that are superlinear in n and, more importantly, match upper bounds from algebraic algorithms?*

In this paper, we show that it is possible to improve some of the existing dynamic matrix inverse algorithms further and at the same time present conjectures that, if true, imply that they cannot be improved anymore.

A. Our Algorithmic Results (Details in Section IV and the full version)

Note that all our algorithms are randomized (Monte Carlo): Their update time guarantees hold with probability one, and their outputs are correct with high probability. Note that unlike typical randomized dynamic algorithms, our algorithms do *not* need the oblivious adversary assumption; i.e. updates can depend on the algorithms' prior outputs. All update times are worst-case.

Algorithms in the Standard Setting (Details in Section IV): We present two faster algorithms as summarized in Fig. 1. With known fast matrix multiplication algorithms [16], [17], our first algorithm requires $O(n^{1.407})$ time to handle each entry update and entry query, and the second requires $O(n^{1.529})$ time to handle each column update and row query.

The first algorithm improves over Sankowski's decade-old $O(n^{1.447})$ bound, and automatically implies improved algorithms for over 10 problems, such as maximum matching size, st-reachability and DAG path counting under edge updates (see upper bounds in blue in Fig. 5 and the second applications table in the full version).

The second bound leads to the first non-trivial upper bounds for maintaining the largest eigenvalue, rank, determinant under column updates, which consequently lead to new algorithms for dynamic graph problems, such as maintaining maximum matching size under insertions and deletions of nodes on one side of a bipartite graph (see upper bounds in red in Fig. 5 and the second applications table in the full version).

Note that the update time can be traded with the query time, but the trade-offs are slightly complicated. See Theorems 4.1 and 4.2 for these trade-offs.

Incremental/Look-Ahead Algorithms and Online Bipartite Matching (Details in the full version): We can speed up our algorithms further in a fairly general *look-ahead* setting, where we know ahead of time which columns will be updated. Previous algorithms ([10]–[13]) can only handle some special cases of this (e.g. when the update columns and the new values are known ahead of time). Our update time depends on how far ahead in the future we see. When we see n columns to be updated in the future, the update time is $O(n^{\omega-1})$. (See the full version for detailed bounds.) As a special case, we can handle the *column-incremental* setting, where we start from an empty (or identity) $n \times n$ matrix and insert the i^{th} column at the i^{th} update.

As an application, we can maintain the maximum matching size of a bipartite graph under the arrival of nodes on one side in $O(n^\omega)$ total time. This problem is known as the *online matching* problem. Our bound improves the $O(m\sqrt{n})$ bound in [22][7] (where m is the number of edges), when the graph is dense, additionally our result matches the bound in the static setting by [24] (see also [25]).

See Section I-C for further discussions on previous results.

Techniques (more in Section II): Our improvements are mostly due to our ability to exploit fast matrix multipli-

[5]Amortized time is not the focus of this paper, and we are not aware of any better amortized bounds for problems we consider in this paper

[6]More precisely, assuming BMM, no "combinatorial" algorithm can maintain maximum matching size and st-reachability in $O(n^{2-\epsilon})$ time, for any constant $\epsilon > 0$. Note that "combinatorial" a vague term usually used to refer as an algorithm that does not use subcubic-time matrix multiplication algorithms as initiated by Strassen [21]. We note that this statement only holds for algorithms with $O(n^{3-\epsilon})$ preprocessing time, which are the case for Sankowski's and our algorithms.

[7]Also see [23].

Variants	Known upper bound	Known lower bound	New upper bound	New lower bound	Corresponding conjectures
Element update Element query	$O(n^{1.447})$ $[O(n^{1+1/3})]$ $O(n^{1.447})$ $[O(n^{1+1/3})]$ [2]	$u+q=\Omega(n)$ via OMv [18]	$O(n^{1.407})$ $[O(n^{1+1/4})]$ $O(n^{1.407})$ $[O(n^{1+1/4})]$ Theorem 4.2	$u+q=\Omega(n^{1.406})$ $[\Omega(n^{1+1/4})]$ Corollary 5.15	uMv-hinted uMv (Conjecture 5.12)
same as above	$O(n^{1.529})$ $[O(n^{1.5})]$ $O(n^{0.529})$ $[O(n^{0.5})]$ [2]			$u=\Omega(n^{1.528})$ $[\Omega(n^{1.5})]$ or $q=\Omega(n^{0.528})$ $[\Omega(n^{0.5})]$ Corollary 5.9	Mv-hinted Mv (Conjecture 5.7)
Element update Row query	$O(n^{1.529})$ $[O(n^{1.5})]$ $O(n^{1.529})$ $[O(n^{1.5})]$ [2]	$u+q=\Omega(n)$ via OMv [18]	-	$u+q=\Omega(n^{1.528})$ $[\Omega(n^{1.5})]$ Corollary 5.5	Mv-hinted Mv (Conjecture 5.7)
Column update Row query	$O(n^2)$ $O(n)$ [trivial]	$u+q=$ $\Omega(n^{\omega-1})$ $[\Omega(n)]$ [trivial]	$O(n^{1.529})$ $O(n^{1.529})$ $[O(n^{1.5})]$ Theorem 4.1	$u+q=\Omega(n^{1.528})$ $[\Omega(n^{1.5})]$ Corollary 5.5	v-hinted Mv (Conjecture 5.2)
Column+Row update Element query	$O(n^2)$ $O(1)$ [trivial]	-	-	$u+q=\Omega(n^2)$ [1]	OMv conjecture [18]

Figure 1. Our new upper and conditional lower bounds (in colors) compared to the previous ones. All previous upper bounds were due to Sankowski [2]. Bounds in brackets $[\cdot]$ are for the ideal scenario, where there exists a linear-time rectangular matrix multiplication algorithm. Colors in the bounds are used to connect to applications in Fig. 5. The exponents in the upper and corresponding lower bounds are different because of the rounding. They are actually the same numbers.

cation more often than previous algorithms. In particular, Sankowski [2] shows that in order to maintain the matrix inverse, it suffices to maintain the inverse of another matrix that we call *transformation matrix*, which has a nicer structure than the input matrix. To keep this nice structure, we have to "reset" the transformation matrix to the identity from time to time; the reset process is where fast matrix multiplication algorithms are used. In more details, Sankowski writes the maintained matrix A as $A = A'T$, where A' is an older version of the matrix and T is a "transformation matrix". He then shows some methods to quickly maintain T^{-1} by exploiting its nice structures. The query about A^{-1} is then answered by computing necessary parts in $A^{-1} = T^{-1}(A')^{-1}$. From time to time, he "resets" the transformation matrix by assigning $A' \leftarrow A'T$, $A'^{-1} \leftarrow T^{-1}A'^{-1}$ and $T \leftarrow \mathbb{I}$ (the identity matrix).

A natural idea to speedup the above algorithm is to repeat the same idea again and again ("bootstrapping"), i.e. to write $T = T_1T_2$ (thus $A^{-1} = (A'T_1T_2)^{-1}$) and try to maintain T_2^{-1} quickly. Indeed, finding a clever way to repeat the same ideas several times is a key approach to significantly speed up many dynamic algorithms. (For a recent example, consider the spanning tree problem where [26] sped up the $n^{1/2-\epsilon}$ update time of [27], [28] to $n^{o(1)}$ by appropriately repeating the approach of [27], [28] for about $\sqrt{\log(n)}$ times. See, e.g., [29]–[31] for other examples.) The challenge is how to do it right. Arguably, this approach has already been taken in [2] where T^{-1} is maintained in the form $T^{-1} = (T_1T_2T_3T_4\ldots)^{-1}$.[8] However, we observe that this and other methods previously used to maintain T^{-1} do *not* exploit fast matrix multiplication, and in fact the same result can be obtained without writing T^{-1} in this long form. (See the discussion of Equation (5) in Section II.) An important question here is: *Can we use fast matrix multiplication to maintain T^{-1}?* An attempt to answer the above questions runs immediately to a barrier: while it is simple to maintain T explicitly after every update, maintaining T_2 *explicitly* already takes too much time!

In this paper, we show that one can get around the above barrier and repeat the approach one more time. To do this, we develop a dynamic matrix inverse algorithm that can handle updates that are *implicit* in a certain way. This algorithm allows us to maintain T_2 *implicitly*, thus avoid introducing a large running time needed to maintain T_2 explicitly. It also generalizes and simplifies one of Sankowski's algorithm, giving additional benefits in speeding up algorithms in the look-ahead setting and algorithms for some graph problems.

Further bootstrapping?: Typically once the approach can be repeated to speed up a dynamic algorithm, it can be repeated several times (e.g. [26], [29], [31]). Given this, it might be tempting to get further speed-ups by writing $A = A'T_1T_2T_3T_4\ldots$ instead of just $A = A'T_1T_2$. Interestingly, it does not seem to help even to write $A = A'T_1T_2T_3$. *Why are we stuck at $A = A'T_1T_2$?* On a technical level, since we have to develop a new, implicit, algorithm to maintain T_2 quickly, it is very unclear how to develop yet another algorithm to maintain T_3 quickly. On a conceptual

[8] [2] presents several dynamic matrix inverse algorithms. Algorithm "Dynamic Matrix Inverse: Simple Updates II" is the one with the $T^{-1} = (T_1T_2T_3\ldots)^{-1}$ structure.

level, this difficulty is captured by our conjectures below, which do not only explain the difficulties for the dynamic matrix inverse problem, but also for many other problems. Thus, these conjectures capture a phenomenon that we have not observed in other problems before. Interestingly, with a small "look-ahead" information, namely the columns to be updated, this approach can be taken further to reduce the update time to match existing conditional lower bounds.

B. Our Conditional Lower Bounds (Details in Section V)

We present conjectures that imply tight conditional lower bounds for many problems. We first present our conjectures and their implications here, and discuss existing conjectures and lower bounds in Section I-C. We emphasize that our goal is not to invent new lower bound techniques, but rather to find a simple, believable, explanation that our bounds are tight. Since the conjectures below are the only explanation we know of, they might be useful to understand other dynamic algebraic algorithms in the future.

Our Conjectures: We present variants of the OMv conjecture. To explain our conjectures, recall a special case of the OMv conjecture called *Matrix-Vector Multiplication (Mv)* [32]–[34]. The problem has two phases. In Phase 1, we are given a boolean matrix M, and in Phase 2 we are given a boolean vector v. Then, we have to output the product Mv. Another closely related problem is the *Vector-Matrix-Vector (uMv)* product problem where in the second phase we are given two vectors u and v and have to output the product $u^\top Mv$. A naive algorithm for these problems is to spend $O(|M|)$ time in Phase 2 to compute Mv and uMv, where $|M|$ is the number of entries in M. The OMv conjecture implies that we cannot beat this naive algorithm even when we can spend polynomial time in the first phase; i.e. there is *no* algorithm that spends polynomial time in Phase 1 and $O(|M|^{1-\epsilon})$ time in Phase 2 for any constant $\epsilon > 0$. (The OMv conjecture in fact implies that this holds even if the second phase is repeated multiple times, but this is not needed in our discussion here.)

In this paper, we consider "hinted" variants of Mv and uMv, where matrices are given as "hints" of u, M and v, and later their submatrices

1) *The v-hinted Mv Problem* (formally defined in Definition 5.1): We are given a boolean matrix M in Phase 1, a boolean matrix V in Phase 2 (as a "hint" of v), and an index i in Phase 3. Then, we have to output the matrix-vector product Mv, where v is the i^{th} column of V.
2) *The Mv-hinted Mv Problem* (formally defined in Definition 5.6): We are given boolean matrices N and V in Phase 1 (as "hints" of M and v), a set of indices K in Phase 2, and an index i in Phase 3. Then, we have to output the matrix-vector product Mv, where v is as above, and M is the submatrix of N obtained by deleting the k^{th} rows of N for all $k \notin K$.
3) *The uMv-hinted uMv Problem* (formally defined in Definition 5.11): We are given boolean matrices U, N, and V in Phase 1 (as "hints" of u, M and v), a set of indices K in Phase 2, a set of indices L in Phase 3, and indices i and j in Phase 4. Then, we have to output the vector-matrix-vector product $u^\top Mv$, where u is the j^{th} column of U, v is as above, and M is the submatrix of N obtained by deleting the k^{th} rows and ℓ^{th} columns of N for all $k \notin K$ and $\ell \notin L$.

A naive algorithm for the first problem (v-hinted Mv) is to either compute Mv naively in $O(|M|)$ time in Phase 3 or precompute Mv for *all possible* v in Phase 2 by running state-of-the-art matrix multiplication algorithms [16], [17] to multiply MV. Our *v-hinted Mv conjecture* (formally stated in Conjecture 5.2) says that we cannot beat the running time of this naive algorithm in Phases 2 and 3 simultaneously even when we have polynomial time in Phase 1; i.e. there is *no* algorithm that spends polynomial time in Phase 1, time polynomially smaller than computing MV with state-of-the-art matrix multiplication algorithms in Phase 2, and $O(|M|^{1-\epsilon})$ time in Phase 3, for any constant $\epsilon > 0$. Similarly, the *Mv-hinted Mv and uMv-hinted uMv conjectures* state that we cannot beat naive algorithms for the Mv-hinted Mv and uMv-hinted uMv problems, which either precompute everything using fast matrix multiplication algorithms in one of the phases or compute Mv and uMv naively in the last phase; see Conjectures 5.7 and 5.12 for their formal statements.

Lower Bounds Based on Our Conjectures: The conjectures above allow us to argue tight conditional lower bounds for the dynamic matrix inverse problem as well as some of its applications. In particular, the uMv-hinted uMv conjecture leads to tight conditional lower bounds for element queries and updates, as well as, e.g., maintaining rank and determinant of a matrix undergoing element updates, and maintaining maximum matching size, st-reachability, cycle detection and DAG path counting in graphs undergoing edge insertions/deletions; see lower bounds in *blue* in Fig. 1 and 5 and the second applications table in the full version for the full list. Our v-hinted Mv conjecture leads to tight conditional lower bounds for column update and row query, as well as, e.g., maintaining adjoint and matrix product under the same type of updates and queries, maintaining bipartite maximum matching under node updates on one side of the graph; see lower bounds in *red* in Fig. 1 and 5 and the second applications table in the full version. Finally, our Mv-hinted Mv conjecture gives conditional lower bounds that match two algorithms of Sankowski [2] that we could not improve, as well as some of their applications; see lower bounds in *green* in Fig. 1 and 5 and the second applications table in the full version. All our tight conditional lower bounds remain tight even if there are improved matrix multiplication algorithms in the future; see, e.g., bounds inside brackets $[\cdot]$ in Fig. 1 and 5, which are valid assuming that a linear-time

matrix multiplication algorithm exists.

Remarks: Our conjectures only imply lower bounds for *worst-case* update and query time, which are the focus of this paper. To make the same bounds hold against amortized time, one can consider the *online* versions of these conjectures, where all phases except the first may be repeated; see full version of the paper. However, we feel that the online versions are too complicated to be the right conjectures, and that it is a very interesting open problem to either come up with clean conjectures that capture amortized update time, or break our upper bounds using amortization.

The reductions from our conjectures are pretty much the same as the existing ones. As discussed in Section I-C, we consider this an advantage of our conjectures. Finally, whether to believe our conjectures or not might depend on the readers' opinions. A more important point is that these easy-to-state conjectures capture the hardness of a number of important dynamic problems. On the way to make further progress on any of these problems is to break naive algorithms from our conjectures first.

C. Other Related Work

Look-Ahead Algorithms: The look-ahead setting refers to when we know the future changes ahead of time and was considered in, e.g., [10]–[13]. Look-ahead dynamic algorithms did not receive as much attention as the non-look-ahead setting due to limited applications, but it turns out that our algorithms require a rather weak look-ahead assumption, and become useful for the online bipartite matching problem [22]. Our results compared with the previous ones are summarized in Fig. 2.

Previously, Sankowski and Mucha [10] showed that algorithms that can look-ahead, i.e. they know which columns will be updated with what values and which rows will be queried, can maintain the inverse and determinant faster than Sankowski's none-look-ahead algorithms [2]. Kavitha [11] extended this result to maintaining rank under element updates, needing to only know which entries will be updated in the future but not their new values. For the case where the algorithm in [10] know n updates in the future, it is *tight* as a better bound would imply a faster matrix multiplication algorithm.

In this paper, we present faster look-ahead algorithms when n^k updates are known ahead of time, for any $k < 1$. More importantly, our algorithms only need to know ahead of time the columns that will be updated, but not the values of their entries. Our algorithms are compared with the previous ones by [10], [11] in Fig. 2. In Fig. 2 we do not state the time explicitly for all possible k, but only state which algorithms are faster and give explicit bounds only for $k = 0.25$. For detailed bounds, see the full version.

[9]A rough estimate for the index is enough: When looking t rounds into the future, we only need an index set of size $O(t)$ as prediction for all the future t update/query positions together.

One special case of our algorithms is maintaining a rank when we start from an empty $n \times n$ matrix and insert the i^{th} column at the i^{th} update. We can compute the rank after each insertion in $O(n^{\omega-1})$ time, or $O(n^{\omega})$ in total over n insertions. Since the maximum matching size in a bipartite graph G corresponds to the rank of a certain matrix M, and adding one node to, say, the right side of G corresponds to adding a column to M, our results imply that we can maintain the maximum matching size under the arrival of nodes on one side in $O(n^{\omega})$ total time. This problem is known as the *online matching* problem. Our bound improves the $O(m\sqrt{n})$ bound in [22][10] (where m is the number of edges), when the graph is dense, additionally our result matches the bound in the static setting by [24] (see also [25]). We note that previous algorithms did not lead to this result because [10] needs to know the new values ahead of time while [11] only handles element updates.

Existing Lower Bounds and Conjectures: Two known conjectures that capture the hardness for most dynamic problems are the Online Matrix-Vector Multiplication conjecture (OMv) [18] and the Strong Exponential Time Hypothesis (SETH) [19]. Since the OMv conjecture implies (roughly) an $\Omega(n)$ lower bound for dynamic matching and st-reachability, it automatically implies a lower bound for dynamic inverse, rank and determinant. However, it is not clear how to use these conjectures to capture the hardness of dynamic problems whose upper bounds can still possibly be improved with improved fast matrix multiplication algorithms.

More suitable conjectures should have dependencies on ω, the matrix multiplication exponent. Based on this type of conjectures, the best lower bound is $\Omega(n^{\omega-1}) = \Omega(n^{1.3729})$ assuming an $\Omega(n^{\omega})$ lower bound for checking if an n-node graph contains a triangle (the Strong Triangle conjecture) [19]. This lower bound does not match our upper bounds of $O(n^{1.407})$ (and note that [19] mentioned that closing the gap between their lower bound and Sankowski's upper bound is a very interesting open question). More importantly, this lower bound applies only for a special case where algorithms' preprocessing time is limited to $o(n^{\omega})$ (in contrast to, e.g., SETH- and OMv-based lower bounds that hold against algorithms with polynomial preprocessing time). Because of this, it unfortunately does not rule out (i) the possibilities to improve the update time of Sankowski's or our algorithms which have $O(n^{\omega})$ preprocessing time and more generally (ii) the existence of algorithms with lower update time but high preprocessing time, which are typically desired. In fact, with such limitation on the preprocessing time, it is easy to argue that maintaining some properties requires n^{ω} update time, which is *higher* than Sankowski's and our upper bounds. For example, assuming that any static algorithm for computing the matrix determinant requires $\Theta(n^{\omega})$ time, we can argue that an algorithm that uses $o(n^{\omega})$ time to

[10]Also see [23].

	Problem	Type of look-ahead	Type of updates	Update time for n^k look-ahead		
				$k = 1$	$k < 1$	$k = 0.25$
[10]	inverse (row query) and determinant but *not* rank	column index and values	column	$O(n^{\omega-1})$ (amortized)	3: slower than 2 for every $k < 1$	$O(n^{1.75})$
[11]	rank	column *and* row index	element	$O(n^{\omega-1})$ (amortized)	4: slowest for every $k < 1$	$O(n^{2.122})$
Theorem 4.1	inverse (row query), determinant and rank	column index[9]	column	$O(n^{\omega-1})$	2: slower than 1 for every $k < 1$	$O(n^{1.453})$
Theorem 4.2	inverse (element query), determinant and rank	column index[9]	element	$O(n^{\omega-1})$	1: fastest for every $k < 1$	$O(n^{1.392})$

Figure 2. Comparison of different look-ahead algorithms. ω is the exponent of matrix multiplication. All inverse algorithms need to know row indices of the queries ahead of time. (Note that the algorithm with element query does not need to know the exact element to be queried.) The results by [12] are not included, as they are subsumed by dynamic algorithms without look-ahead. The algorithms maintaining inverse and determinant can also maintain adjoint, linear system and other algebraic problems via the reductions from the full version of the paper. The rank reduction is adaptive and does not work with the type of look-ahead used in [10].

preprocess a matrix A requires n^ω time to maintain the determinant of A even when an update does nothing to A. (See the full version of this paper for more lower bounds of this type.) Because of this, we aim to argue lower bounds for algorithm with polynomial preprocessing time.

In light of the above discussions, the next appropriate choice is to make new conjectures. While there are many possible conjectures to make, we select the above because they are *simple* and *similar to the existing ones*. We believe that this provides some advantages: (i) It is easier to develop an intuition whether the new conjectures are true or not, based on the knowledge of the existing conjectures; for example, we discuss what a previous attempt to refute the OMv conjecture [33] means to our new conjectures in the full version of the paper. (ii) There is a higher chance that existing reductions (from known conjectures) can be applied to the new ones. Indeed, this is why our conjectures imply tight lower bounds for many problems beyond dynamic matrix inverse.

We note that while the term "hinted" was not used before in the literature, the concept itself is not that unfamiliar. For example, Patrascu's *multiphase problem* [20] is a hinted version of the vector-vector product problem: given a boolean matrix U in Phase 1, vector v in Phase 2, and an index i in Phase 3, compute the inner product $u^\top v$ where u is the i^{th} column of matrix U.

II. Overview of Our Algorithms

Let $A^{(0)}$ be the initial matrix A before any updates and denote with $A^{(t)}$ the matrix A after it received t updates. For now we will focus only on the case where $A^{(t)}$ is always invertible, as a reduction from [3] allows us to extend the algorithm to the setting where $A^{(t)}$ may become singular. We will also focus only on the case of element updates and queries. The same ideas can be extended to other cases.

Reduction to Transformation Inverse Maintenance (Details in Section IV-A): The core idea of the previously-best dynamic inverse algorithms of Sankowski [2] is to express the change of matrix $A^{(0)}$ to $A^{(t)}$ via some *transformation matrix* $T^{(0,t)}$, i.e. we write

$$A^{(t)} = A^{(0)}T^{(0,t)} \tag{1}$$

This approach is beneficial since $T^{(0,t)}$ has more structure than $A^{(t)}$: (i) obviously $T^{(0,0)} = \mathbb{I}$ (the identity matrix), and (ii) changing the (i,j)-entry of $A^{(t)}$ changes only the j^{th} column of $T^{(0,t)}$ (and such change can be computed in $O(n)$ time).[11] Thus for small t, the matrix $T^{(0,t)}$ differs in at most t columns from the identity matrix, which allows us to maintain its inverse more efficiently than that of some general matrix $A^{(t)}$, as will be explained in more detail later. Moreover, to get the (i,j)-entry of $(A^{(t)})^{-1}$ notice that

$$(A^{(t)})^{-1} = (T^{(0,t)})^{-1}(A^{(0)})^{-1}, \tag{2}$$

and thus we just have to multiply the i^{th} row of $(T^{(0,t)})^{-1}$ with the j^{th} column of $(A^{(0)})^{-1}$. This motivates the following problem.

Problem 2.1. (Maintaining the inverse of the transformation, $(T^{(0,t)})^{-1}$): *We start with $T^{(0,0)} = \mathbb{I}$. Each update is a change in one column. A query is made on a row of $(T^{(0,t)})^{-1}$. It can be assumed that $T^{(0,t)}$ is invertible for any t.*

As we will see below, there are many fast algorithms for Problem 2.1 when t is small. A standard "resetting technique" can then convert these algorithms into fast algorithms for maintaining matrix inverse: An element update to $A^{(t)}$ becomes a column update to $T^{(0,t)}$. When t gets large (thus algorithms for Problem 2.1 become slow), we use fast matrix multiplication to compute $(A^{(t)})^{-1}$ explicitly so that $T^{(0,t)}$ is "reset" to $\mathbb{I}$.

To summarize, it suffices to solve Problem 2.1. Our improvements follow directly from improved algorithms for this problem, which will be our focus in the rest of this section.

[11] To see this, write $T^{(0,t)} = (A^{(0)})^{-1}A^{(t)}$. The (i,j)-entry of $A^{(t)}$ will multiply only with the i^{th} column of $(A^{(0)})^{-1}$ and affects the j^{th} column of the product.

	[2]	[2]	Our	
Column update	$O(nt)$	$O(t^2)$	$O(n^x t + n^{\omega(1,x,\log_n t)-x})$	$[O(\sqrt{n}t)]$
Row query	$O(t)$	$O(t^2)$	$O(n^x t + n^{\omega(1,x,\log_n t)-x})$	$[O(\sqrt{n}t)]$

Figure 3. Comparison of different transformation maintenance algorithms. The task is to support column updates to $T^{(0,t)}$ and row queries to $(T^{(0,t)})^{-1}$, where t is the number of updates so far. Values in $[\cdot]$ correspond to the case of optimal matrix multiplication ($\omega = 2$) and are given for easier comparison of the complexities.

Previous maintenance of $(T^{(0,t)})^{-1}$: Sankowski [2] presented two algorithms for maintaining $(T^{(0,t)})^{-1}$; see Fig. 3. The first algorithm maintains $(T^{(0,t)})^{-1}$ *explicitly* by observing *if a matrix M differs from $\mathbb{I}$ in at most k columns, so does its inverse*.[12] This immediately implies that querying a row of $(T^{(0,t)})^{-1}$ needs $O(t)$ time, since $(T^{(0,t)})^{-1}$ differs from $\mathbb{I}$ in at most t columns. Moreover, expressing an update by a linear transformation, i.e.

$$T^{(0,t)} = T^{(0,t-1)}T^{(t-1,t)} \tag{3}$$

for some matrix $T^{(t-1,t)}$, and using the fact that $T^{(t-1,t)}$ and $(T^{(t-1,t)})^{-1}$ differ from $\mathbb{I}$ in only one column, computing $(T^{(0,t)})^{-1}$ boils down to multiplying a vector with $(T^{(0,t)})^{-1}$, thus taking $O(nt)$ update time, see for example Fig. 4. The $O(nt)$ update time of this algorithm is optimal in the sense that one column update to $T^{(0,t)}$ may cause $\Omega(nt)$ entries in $(T^{(0,t)})^{-1}$ to change; thus maintaining $(T^{(0,t)})^{-1}$ explicitly requires $\Omega(nt)$ update time in the worst case.[13] Sankowski's second algorithm breaks this bound with the cost of higher query time. Such a decrease in update time at the cost of increase query time is desired when balancing update and query time, e.g. for applications such as maintaining determinant, st-reachability, maximum matching size.

Sankowski's second algorithm expresses updates by a long chain of linear transformations:

$$\begin{aligned} T^{(0,t)} &= T^{(0,1)}T^{(1,2)}\dots T^{(t-2,t-1)}T^{(t-1,t)}, \text{thus} \\ (T^{(0,t)})^{-1} &= (T^{(t-1,t)})^{-1}(T^{(t-2,t-1)})^{-1}\dots(T^{(0,1)})^{-1}. \end{aligned} \tag{5}$$

Here each matrix $(T^{(i,i+1)})^{-1}$ is very sparse. The sparsity leads to the update time improvement over the first algorithm, since computing some entries of $(T^{(0,t)})^{-1}$ does not require all entries of each $(T^{(i,i+1)})^{-1}$ to be known (intuitively because most entries will be multiplied with zero).[14]The sparsity, however, also makes it hard to exploit fast matrix multiplication. Exploiting fast matrix multiplication one more time is the new aspect of our algorithm.

[12]We will give a more formal argument for this later in (8).

[13]For an example of one changed column inducing $\Omega(nt)$ changes in the inverse, we refer to the full version of the paper.

[14]Note that the update time of Sankowski's second algorithm in Fig. 3 is presented in a slightly simplified form. In particular, this bound only holds when $t = \Omega(\sqrt{n})$ (which is the only case we need in this paper), or otherwise it should be the bound of the number of arithmetic operations only.

Our new maintenance of $(T^{(0,t)})^{-1}$ via fast matrix multiplication: As discussed above and as can be checked in [2], both algorithms of Sankowski do *not* use fast matrix multiplication to maintain $(T^{(0,t)})^{-1}$; it is used only to compute $(A^{(t)})^{-1}$ as in Equation (2) (to "reset").[15] Our improvements are mostly because *we can use fast matrix multiplication to maintain* $(T^{(0,t)})^{-1}$. To start with, we write $T^{(0,t)}$ as

$$\begin{aligned} T^{(0,t)} &= T^{(0,t')}T^{(t',t)}, \\ \text{thus} \quad (T^{(0,t)})^{-1} &= (T^{(t',t)})^{-1}(T^{(0,t')})^{-1}. \end{aligned} \tag{6}$$

This looks very much like what Sankowski's first algorithm (see Equation (2)) *except that we may have $t' \ll t$*; this allows us to benefit from fast matrix multiplication when we compute $T^{(0,t')}T^{(t',t)}$, since both matrices are quite dense. Like the discussion above Problem 2.1, a column update of $T^{(0,t)}$ leads to a column update of $T^{(t',t)}$, and a row query to $(T^{(0,t)})^{-1}$ needs a row query to $(T^{(t',t)})^{-1}$. This seems to suggest that maintaining $(T^{(0,t)})^{-1}$ can be once again reduced to solving the same problem for $T^{(t',t)}$, and by repeating Sankowski's idea we should be able to exploit fast matrix multiplication and maintain $(T^{(0,t)})^{-1}$ faster.

There is, however, an obstacle to execute this idea: even just maintaining $T^{(t',t)}$ *explicitly* (without its inverse) already takes *too much time*. To see this, suppose that at time t we add a vector v to the j^{th} column of $T^{(0,t-1)}$; with e_j being a unit vector which has value 1 at the j^{th} coordinate and 0 otherwise, this can be expressed as

$$\begin{aligned} T^{(0,t)} &= T^{(0,t-1)} + e_j^\top v, \\ \text{thus (by (6))}\ T^{(t',t)} &= T^{(t',t-1)} + e_j^\top[(T^{(0,t')})^{-1}v]. \end{aligned} \tag{7}$$

This means that for every column update to $T^{(0,t)}$, we have to compute a matrix-vector product $(T^{(0,t')})^{-1}v$ just to obtain $T^{(t',t-1)}$. So for every update we have to read the entire inverse $(T^{(0,t')})^{-1}$, which has $\Omega(nt')$ non-zero entries. Given that we repeatedly reset the algorithm to exploit fast matrix multiplication by setting $t' \leftarrow t$, this yields a $\Omega(nt)$ lower bound on our approach, i.e. no improvement over Sankowski's first algorithm (column 1 of Fig. 3).

So to summarize, just maintaining $T^{(t',t)}$ is already too slow.

[15]In particular, both of Sankowski's algorithms maintain $(T^{(0,t)})^{-1}$ by performing matrix-vector products.

More details (may be skipped at first reading): We can write

$$T^{(0,t)} = \underbrace{T^{(0,t-1)} \left(\mathbb{I} + (T^{(0,t-1)})^{-1} \underbrace{[T^{(0,t)} - T^{(0,t-1)}]}_{C} \right)}_{T^{(t-1,t)}}, \text{ thus}$$

$$(T^{(0,t)})^{-1} = \underbrace{\left(\mathbb{I} + (T^{(0,t-1)})^{-1} \underbrace{[T^{(0,t)} - T^{(0,t-1)}]}_{C} \right)^{-1}}_{(T^{(t-1,t)})^{-1}} (T^{(0,t-1)})^{-1}. \tag{4}$$

Since $C = [T^{(0,t)} - T^{(0,t-1)}]$ contains only one non-zero column, $T^{(t-1,t)}$ differs from $\mathbb{I}$ only in one column. Consequently, $(T^{(t-1,t)})^{-1}$ can be computed in $O(n)$ time and differs from $\mathbb{I}$ only in one column. Thus $T^{(t-1,t)}(T^{(0,t-1)})^{-1}$ takes $O(nt)$ time to compute.

Figure 4. Updating $T^{(0,t)}$ explicitly takes $O(nt)$ time.

Implicit input, simplification and generalization of Sankowski's second algorithm (details in Section IV-C): To get around the above obstacle, we consider when updates to $T^{(t',t)}$ are given *implicitly*:

Problem 2.2. (Maintaining inverse of the transformation under implicit column updates) *We start with $T^{(t',t')} = \mathbb{I}$ at time t'. Each update is an index j, indicating that some change happens in the j^{th} column. Whenever the algorithm wants to know a particular entry in $T^{(t',t)}$ (at time $t \geq t'$), it can make a query to an* oracle*. The algorithm also has to answer a query made on a row of $(T^{(t',t)})^{-1}$ at any time t. The algorithm's performance is measured by its running time* and *the number of oracle queries. It can be assumed that $T^{(t',t)}$ is invertible for any t.*

In Section IV-C, we develop an algorithm for the above problem. It has the same update and query time as Sankowski's second algorithm, i.e. $O((t-t')^2)$ and additionally makes $O(t-t')$ oracle queries to perform each operation. Moreover, our algorithm does not need to maintain a chain of matrices as in Equation (5). Eliminating this chain allows a further use of fast matrix multiplication, which yields an additional runtime improvement for the setting of batch-updates and batch-queries, i.e. when more than one entry is changed/queried at a time. This leads to improvements in the look-ahead setting and for some graph problems such as online-matching.

The starting point of our algorithm for Problem 2.2 is the fact that $T^{(t',t)}$ and $(T^{(t',t)})^{-1}$ differs in at most $t-t'$ columns from the identity. Thus, by appropriately permuting rows and columns, we can write them as

$$T^{(t',t)} = \left(\begin{array}{c|c} C_1 & 0 \\ \hline C_2 & \mathbb{I} \end{array} \right)$$
$$(T^{(t',t)})^{-1} = \left(\begin{array}{c|c} C_1^{-1} & 0 \\ \hline -C_2 C_1^{-1} & \mathbb{I} \end{array} \right) \tag{8}$$

Here, C_1 and C_2 are $(t-t') \times (t-t')$- and $(n-t+t') \times (t-t')$-matrices, respectively. This observation immediately yields the following solution to our problem: (i) In order to maintain $(T^{(t',t)})^{-1}$ implicitly, we only need to know the C_1 block of $T^{(t',t)}$. Since a column update to $T^{(t',t)}$ may change C_1 in $O(t-t')$ entries (i.e. either a column of C_1 is modified or a new row and column is added to C_1), we only need $O(t-t')$ oracle queries to keep track of C_1^{-1} after each update.[16] (ii) To answer a query about some row of $(T^{(t',t)})^{-1}$ we may need a row of the C_2 block and compute the vector-matrix product of such row of C_2 with C_1^{-1}. Getting such row of C_2 requires $O(t-t')$ oracle queries.

In summary, we do not require to fully know the matrix $T^{(t',t)}$ in order to maintain its inverse. This algorithm for maintaining $(T^{(t',t)})^{-1}$ is formalized in Lemma 4.7 (Section IV-C).

Back to maintaining $(T^{(0,t)})^{-1}$: Using implicit $(T^{(t',t)})^{-1}$ maintenance (Details in Section IV-D): We now sketch how we use the algorithm that maintains $(T^{(t',t)})^{-1}$ with implicit updates (cf. Problem 2.2) to maintain $(T^{(0,t)})^{-1}$ (cf. Problem 2.1). The main idea is that we will implicitly maintain $T^{(t',t)}$ by explicitly maintaining $(T^{(0,t')})^{-1}$ and matrix

$$S^{(t',t)} := T^{(0,t)} - T^{(0,t')}. \tag{9}$$

[16]To maintain C_1^{-1} we use an extended version of [2, Theorem 1].

Like Equation (7), we can derive

$$T^{(0,t)} = T^{(0,t')} + S^{(t',t)}, \tag{10}$$
$$\text{thus (by (6))}\ T^{(t',t)} = \mathbb{I} + (T^{(0,t')})^{-1} S^{(t',t)}.$$

Thus, we can implement an oracle that provides an entry of $T^{(t',t)}$ by multiplying a row of $(T^{(0,t')})^{-1}$ with a column of $S^{(t',t)}$. This can be done pretty fast by exploiting the fact that these matrices are rather sparse, i.e. $(T^{(0,t')})^{-1}$ differs from the identity matrix in at most t' columns and $S^{(t',t)}$ has only $t'-t$ non-zero columns.

Summary: In a nutshell, our algorithm maintains

$$A^{(t)} = A^{(0)} T^{(0,t')} T^{(t',t)},$$
$$\text{thus}\quad (A^{(t)})^{-1} = (T^{(t',t)})^{-1} (T^{(0,t')})^{-1} (A^{(0)})^{-1}.$$

We keep the explicit values of $A^{(0)}$ and $T^{(0,t')}$ at any time. Additionally, we maintain explicitly a matrix $S^{(t',t)}$ satisfying Equation (10) (i.e. it collects all updates to $T^{(0,t)}$ since time t'). As a subroutine we run our algorithm for Problem 2.2 to maintain $(T^{(t',t)})^{-1}$ with implicit updates; call this algorithm $\mathcal{L}^{(t',t)}$, and see its detailed description in Section IV-C.

When, say, the entry (i,j) of $A^{(t)}$ is updated, we (i) update matrix $S^{(t',t)}$, and (ii) implicitly update $T^{(t',t)}$ by sending index j to $\mathcal{L}^{(t',t)}$ (the previously outlined algorithm for Problem 2.2). The first task is done by computing each update to $T^{(0,t)}$, which is not hard: since $T^{(0,t)} = (A^{(0)})^{-1} A^{(t)}$, we have to change the j^{th} column of $T^{(0,t)}$ to the product of the i^{th} column of $(A^{(0)})^{-1}$ and the changed entry (i,j) of $A^{(t)}$ (see footnote 11). For the second task, $\mathcal{L}^{(t',t)}$ might make some oracle queries. By Equation (10), each query can be answered by multiplying a row of $(T^{(0,t')})^{-1}$ with a column of $S^{(t',t)}$.

When, say, the i^{th} row of $(A^{(t)})^{-1}$ is queried, we need to multiply a row of $(T^{(t',t)})^{-1}$ with $(T^{(0,t')})^{-1}(A^{(0)})^{-1}$. Such row is obtained by making a query to algorithm $\mathcal{L}^{(t',t)}$; again, we use $(T^{(0,t')})^{-1}$ and $S^{(t',t)}$ to answer oracle queries made by $\mathcal{L}^{(t',t)}$. When multiplying the vector-matrix-matrix product from left to right, each vector-matrix product takes time linear to the product of the number of non-zero entries in the vector and the number of non-identity columns in the matrix.

Section IV-D describes in details how we implement the two operations above.

The running time $\mathcal{L}^{(t',t)}$ depend on $t-t'$. When $t-t'$ gets large, we "reset" $T^{(t',t)}$ to $\mathbb{I}$ by setting $t' \leftarrow t$ and compute $T^{(0,t)} = T^{(0,t')} T^{(t',t)}$ using fast matrix multiplication. The latter is done in a similar way to Sankowski's first algorithm. In particular, we write down equations similar to Equation (4), except that now we have $C = [T^{(0,t)} - T^{(0,t')}]$. Given that C is quite dense (since $t' \ll t$), we can exploit fast matrix multiplication here while the original algorithm that uses Equation (4) cannot. See details in Section IV-B.

Computing $T^{(0,t)} = T^{(0,t')} T^{(t',t)}$ also becomes slow when t is large. In this case, we "reset" both $T^{(0,t')}$ and $T^{(t',t)}$ to $\mathbb{I}$ by computing $A^{(t)} = A^{(0)} T^{(0,t')} T^{(t',t)}$ and pretend that $A^{(t)}$ is our new $A^{(0)}$. Once again we can exploit fast matrix multiplication here. See details in Section IV-A.

Discussions: Now that we can exploit fast matrix multiplication one more time compared to previous algorithms, it is natural to ask whether we can exploit it another time. A technical obstacle is that to use fast matrix multiplication twice we already have to solve a different problem (Problem 2.2 vs. Problem 2.1); thus it is unclear whether and how we should define another problem to be able to use fast matrix multiplication another time. A more fundamental obstacle is our conjectures: to get any further improvement we have to break these conjectures, as we will discuss in Section V.

III. Preliminaries

In this section we will define our notation and state some simple results about matrix multiplication and inversion.

Notation: Identity and Submatrices: The identity matrix is denoted by $\mathbb{I}$.

Let $I, J \subset [n] := \{1, ..., n\}$ and A be an $n \times n$ matrix, then the term $A_{I,J}$ denotes the submatrix of A consisting of the rows I and columns J. For some $i \in [n]$ the term $A_{[n],i}$ can thus be seen as the ith column of A.

Let $I = (i_1, ...i_p) \in [n]^p$, $J = (j_1, ...j_q) \in [n]^q$ and A be an $n \times n$ matrix, then the term $A_{I,J}$ denotes a matrix such that $(A_{I,J})_{s,t} = A_{i_s,j_t}$. Specifically for $i_1 < ... < i_p$ and $j_1 < ... < j_t$ the term $A_{I,J}$ is just the submatrix of A when interpreting I and J as sets instead of vectors. We may also mix the notation e.g. for $I = (i_1, ...i_p) \in [n]^p$ and $J \subset [n]$, we can consider J to be an ordered set such that $j_1 < ... < j_q$, then the term $A_{I,J}$ is just the matrix where $(A_{I,J})_{s,t} = A_{i_s,j_t}$.

Inner, Outer and Matrix Products: Given two vectors u and v we will write $u^\top v$ for the inner product and $uv^\top$ for the outer product. This way inner and outer product are just special cases of matrix multiplication, i.e. inner product is a $1 \times n$ matrix multiplied with an $n \times 1$ matrix, while an outer product is the product of an $n \times 1$ matrix by a $1 \times n$ matrix.

We will also often exploit the fact that each entry of a matrix product is given by an inner product: $(AB)_{i,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j} = A_{i,[n]} B_{[n],j} = e_i A B e_j$. In other words, to compute entry (i,j) of AB we just multiply the ith row of A with the jth column of B.

Fast Matrix Multiplication: We denote with $O(n^\omega)$ the complexity of multiplying two $n \times n$ matrices. Note that matrix multiplication, inversion, determinant and rank, all have the same complexity [35, Chapter 16]. Currently the best bound is $\omega < 2.3729$ [17], [36].

For rectangular matrices we denote the complexity of multiplying an $n^a \times n^b$ matrix with an $n^b \times n^c$ matrix

Problem	Known upper bound	New upper bound	Known lower bound	New lower bound
Bipartite maximum matching (online, total time)	$O(m\sqrt{n})$ [22]	$O(n^{\omega})$	- -	- -
Bipartite maximum matching (fully dynamic) edge update	$O(n^{1.447})$ $[O(n^{1+1/3})]$ [3]	$O(n^{1.407})$ $[O(n^{1+1/4})]$	$\Omega(n)$ [18] $\Omega(m^{0.814})^{\ddagger}$ [19]	$\Omega(n^{1.406})$ $[\Omega(n^{1+1/4})]$ Corollary 5.14
right side node update	$O(n^2)$ [3]	$O(n^{1.529})$ $[O(n^{1.5})]$	same as above	
Maximum matching (general graphs) edge update	$O(n^{1.447})$ $[O(n^{1+1/3})]$ [3]	$O(n^{1.407})$ $[O(n^{1+1/4})]$	$\Omega(n)$ [18] $\Omega(m^{0.814})^{\ddagger}$ [19]	$\Omega(n^{1.406})$ $[\Omega(n^{1+1/4})]$ Corollary 5.14
DAG path counting† and Transitive Closure edge update	$O(n^{1.447})$ $[O(n^{1+1/3})]$ [2]	$O(n^{1.407})$ $[O(n^{1+1/4})]$	$u+q=\Omega(n)$ [18]	$u+q=\Omega(n^{1.406})$ $[\Omega(n^{1+1/4})]$
pair query	$O(n^{1.447})$ $[O(n^{1+1/3})]$ [2]	$O(n^{1.407})$ $[O(n^{1+1/4})]$	$u+q=\Omega(m^{0.814})^{\ddagger}$ [19]	Corollary 5.14
same as above	$O(n^{1.529})$ $[O(n^{1.5})]$ [2] $O(n^{0.529})$ $[O(n^{0.5})]$ [2]			$u=\Omega(n^{1.528})$ $[\Omega(n^{1.5})]$ or $q=\Omega(n^{0.528})$ $[\Omega(n^{0.5})]$ Corollary 5.10
node update (incoming edges)	$O(n^2)$ [2]	$O(n^{1.529})$ $[O(n^{1.5})]$	$u+q=\Omega(n)$ [18] $u+q=\Omega(m^{0.814})^{\ddagger}$	$u+q=\Omega(n^{1.528})$ $[\Omega(n^{1.5})]$ Corollary 5.4
source query	$O(n)$ [2]	$O(n^{1.529})$ $[O(n^{1.5})]$	[19]	
edge update source query	$O(n^{1.529})$ $[O(n^{1.5})]$ [2] $O(n^{1.529})$ $[O(n^{1.5})]$ [2]	-	$n\cdot u+q=\Omega(n^2)$ [18]	$u+q=\Omega(n^{1.528})$ $[\Omega(n^{1.5})]$ Corollary 5.10
All-pair-distances (unweighted) edge update	$O(n^{1.897})$ $[\tilde{O}(n^{2-1/8})]$ [7]	$O(n^{1.724})$ $[\tilde{O}(n^{1+2/3})]$	same as edge update/pair query transitive closure	
pair query	$O(n^{1.265})$ $[\tilde{O}(n^{1+1/4})]$ [7]	$O(n^{1.724})$ $[\tilde{O}(n^{1+2/3})]$		
Strong connectivity edge update	$O(n^{1.529})$* $[O(n^{1+1/2})]$ [2]	$O(n^{1.529})$ $[O(n^{1.5})]$	$\Omega(n)$ [18] $\Omega(m^{0.814})^{\ddagger}$[19]	$\Omega(n^{1.406})$ $[\Omega(n^{1+1/4})]$ Corollary 5.14
node update (incoming edges)	$O(n^2)$* [2]	$O(n^{1.529})$ $[O(n^{1.5})]$	same as above	
Counting node disjoint ST-paths edge update	$O(n^{1.447})$ [3]	$O(n^{1.407})$ $[O(n^{1+1/4})]$	$\Omega(n)$ [18] $\Omega(m^{0.814})^{\ddagger}$[19]	$\Omega(n^{1.406})$ $[\Omega(n^{1+1/4})]$ (via transitive closure)
Counting spanning trees† edge update	$O(n^{1.447})$ $[O(n^{1+1/3})]$ [2]	$O(n^{1.407})$ $[O(n^{1+1/4})]$	-	-
Triangle detection node update	$O(n^2)$ [trivial]	-	$\Omega(n^2)$ [18]	-
node update (incoming edges)	$O(n^2)$ [trivial]	$O(n^{1.529})$ $[O(n^{1.5})]$	$\Omega(n)$ [18]	-
node update (turn node on/off)	$O(n^2)$ [trivial]	$O(n^{1.407})$ $[O(n^{1+1/4})]$	-	-
Directed cycle detection and directed k-cycle (constant k) edge update	$O(n^{1.447})$* $[O(n^{1+1/3})]$ [7]	$O(n^{1.407})$ $[O(n^{1+1/4})]$	$\Omega(n)$ ($k\geq 3$) [18]	$\Omega(n^{1.406})$ $[\Omega(n^{1+1/4})]$ ($k\geq 6$) Corollary 5.14
node update (incoming edges)	$O(n^2)$* [7]	$O(n^{1.529})$ $[O(n^{1.5})]$	same as above	
Directed k-path (constant k) edge update	$O(n^{1.447})$* $[O(n^{1+1/3})]$ [2]	$O(n^{1.407})$ $[O(n^{1+1/4})]$	same as transitive closure	same as transitive closure
pair query	$O(n^{1.447})$* $[O(n^{1+1/3})]$ [2]	$O(n^{1.407})$ $[O(n^{1+1/4})]$	for $k\geq 3$	for $k\geq 5$
node update (incoming edges)	$O(n^2)$* [2]	$O(n^{1.529})$ $[O(n^{1.5})]$	same as transitive closure for $k\geq 3$	same as transitive closure for $k\geq 3$
source query	$O(n)$* [2]	$O(n^{1.529})$ $[O(n^{1.5})]$		

Figure 5. The table displays the previous best upper/lower bounds and our results for dynamic gaph problems. Bounds inside brackets $[\cdot]$ are valid assuming that a linear-time matrix multiplication algorithm exists. Lower bounds marked with ‡ only hold for sparse graphs $m=O(n^{1.67})$ and when assuming $O(m^{1.407})$ pre-processing time. The complexities for problems marked with † are measured in the number of arithmetic operations. All other complexities measure the time. Bounds marked with * are new applications of dynamic matrix inverse that were not previously stated. Colors of upper bounds indicate which bound in Fig. 1 each bound in this table follows from. Colors of lower bounds indicate which conjecture in Fig. 1 each bound in this table follows from. For details, see the full version.

Figure 6. Invert (Fact 3.1)

Input: $n \times n$ matrix $A = \mathbb{I} + C$ where $J \subset [n]$ are the indices of the nonzero columns of C.
Output: $A^{-1} = \mathbb{I} + \tilde{C}$
1: $\tilde{C}_{J,J} \leftarrow (A_{J,J})^{-1}$
2: $\tilde{C}_{[n]\setminus J,J} \leftarrow -C_{[n]\setminus J,J}\tilde{C}_{J,J}$
3: **return** $\mathbb{I} + \tilde{C}$

Figure 7. PartialInvert (Fact 3.1)

Input: Rows $I \subset [n]$ of a matrix $A = \mathbb{I} + C$ where $J \subset [n]$ are the indices of the nonzero columns of C and $J \subset I$.
Output: Rows I of $A^{-1} = \mathbb{I} + \tilde{C}$
1: $\tilde{C}_{J,J} \leftarrow (A_{J,J})^{-1}$
2: $\tilde{C}_{I\setminus J,J} \leftarrow -C_{I\setminus J,J}\tilde{C}_{J,J}$
3: **return** $\mathbb{I} + \tilde{C}$

with $O(n^{\omega(a,b,c)})$ for any $0 \leq a, b, c$. Note that $\omega(\cdot,\cdot,\cdot)$ is a symmetric function so we are allowed to reorder the arguments. The currently best bounds for $\omega(1,1,c)$ can be found in [16].

The complexity of the algorithms presented in this paper depend on the complexity of multiplying and inverting matrices. For a more in-depth analysis of how we balance the terms that depend on ω (e.g. how we compute $\omega(a,b,c)$ for $a, b \neq 1$), we refer to the full version of the paper.

Transformation Matrices: Throughout this paper, we will often have matrices of the form $T = \mathbb{I} + C$, where C has few non-zero columns. We will often call these matrices transformation matrices.

Note that any matrix $T = \mathbb{I} + C$, where C has at most m non-zero columns, can be brought in the following form by permuting the rows and columns, which corresponds to permuting the columns and rows of its inverse T^{-1} [2, Section 5]:

$$T = \left(\begin{array}{c|c} C_1 & 0 \\ \hline C_2 & \mathbb{I} \end{array}\right)$$

Here C_1 is of size $m \times m$ and C_2 of size $(n-m) \times m$. The inverse is given by

$$T^{-1} = \left(\begin{array}{c|c} C_1^{-1} & 0 \\ \hline -C_2C_1^{-1} & \mathbb{I} \end{array}\right)$$

In general, without prior permutation of rows/columns, we can state for T and its inverse the following facts:

Fact 3.1. *Let T be an $n \times n$ matrix of the form $\mathbb{I} + C$ and let $J \subset [n]$ be the column indices of the non-zero columns of C, and thus for $K := [n]\setminus J$ we have $C_{[n],K} = 0$.*

Then:

- $(T^{-1})_{J,J} = (T_{J,J})^{-1}$, $(T^{-1})_{K,J} = -T_{K,J}(T_{J,J})^{-1}$, $(T^{-1})_{K,K} = \mathbb{I}$ *and* $(T^{-1})_{J,K} = 0$.
- *For $|J| = n^\delta$ the inverse T^{-1} can be computed in $O(n^{\omega(1,\delta,\delta)})$ field operations (Fig. 6).*
- *If given some set $I \subset [n]$ with $J \subset I$, $|I| = n^\varepsilon$, then rows I of T^{-1} can be computed in $O(n^{\omega(\varepsilon,\delta,\delta)})$ and for this we only need to know the rows I of T. (Fig. 7)*

We will often multiply matrices of the form $\mathbb{I} + C$ where C has few non-zero columns. The complexity of such multiplications is as follows:

Fact 3.2. *Let A, B be $n \times n$ matrices of the form $A = \mathbb{I} + C$, $B = \mathbb{I} + N$, where C has n^a non-zero columns and N has n^b non-zero columns.*

Then:

- *The product AB can be computed in $O(n^{\omega(1,a,b)})$ operations.*
- *If $J_C, J_N \subset [n]$ are the sets of column indices where C (or respectively N) is non-zero, then AB is of the form $AB = \mathbb{I} + M$ where M can only be non-zero on columns with index in $J_C \cup J_N$.*
- *If we want to compute only a subset of the rows, i.e. for $I \subset [n]$ we want to compute $(AB)_{I,[n]} = A_{I,[n]}B$, then for $|I| = n^c$ this requires $O(n^{\omega(c,a,b)})$ operations. For this we only require the rows with index $I \cup J_C$ of the matrix N, so we do not have to know the other entries of N to compute the product.*

This fact is a direct implication of $(I + C)(I + N) = I + C + N + CN$ and $CN_{[n],J_N} = C_{[n],J_C}N_{J_C,J_N}$.

IV. Dynamic Matrix Inverse

In this section we show the main algorithmic result, which are two algorithms for dynamic matrix inverse. The first one supports column updates and row queries, while the second one supports element updates and element queries. These two algorithms imply more than ten faster dynamic algorithms, see Fig. 5 and the full version for applications.

Theorem 4.1. *For every $0 \leq \varepsilon \leq 1$ there exists a dynamic algorithm for maintaining the inverse of an $n \times n$ matrix A, requiring $O(n^\omega)$ field operations during the pre-processing. The algorithm supports changing any column of A in $O(n^{1+\varepsilon} + n^{\omega(1,1,\varepsilon)-\varepsilon})$ field operations and querying any row of A^{-1} in $O(n^{1+\varepsilon})$ field operations.*

For current bounds on ω this implies a $O(n^{1.529})$ upper bound on the update and query cost ($\varepsilon \approx 0.529$). For $\omega = 2$ the update and query time become $O(n^{1.5})$ ($\varepsilon = 0.5$).

Theorem 4.2. *For every $0 \leq \varepsilon_1 \leq \varepsilon_2 \leq 1$ there exists a dynamic algorithm for maintaining the inverse of an $n \times n$ matrix A, requiring $O(n^\omega)$ field operations during the pre-processing, The algorithm supports changing any entry of A in $O(n^{\varepsilon_2+\varepsilon_1} + n^{\omega(1,\varepsilon_1,\varepsilon_2)-\varepsilon_1} + n^{\omega(1,1,\varepsilon_2)-\varepsilon_2})$ field operations and querying any entry of A^{-1} in $O(n^{\varepsilon_2+\varepsilon_1})$ field operations.*

When balancing the terms for current values of ω, the update and query cost are $O(n^{1.407})$ (for $\varepsilon_1 \approx 0.551$,

$\varepsilon_2 \approx 0.855$*). For* $\omega = 2$ *the update and query time become* $O(n^{1.25})$ *(for* $\varepsilon_1 = 0.5$*,* $\varepsilon_2 = 0.75$*).*

Throughout this section, we will write $A^{(t)}$ to denote the matrix A after t updates. The algorithms from both Theorem 4.1 and Theorem 4.2 are based on Sankowski's idea [2] of expressing the change of some matrix $A^{(t-1)}$ to $A^{(t)}$ via a linear transformation $T^{(0,t)}$, such that $A^{(t)} = A^{(0)}T^{(0,t)}$ and thus $(A^{(t)})^{-1} = (T^{(0,t)})^{-1}(A^{(0)})^{-1}$. The task of maintaining the inverse of $A^{(t)}$ thus becomes a task about maintaining the inverse of $T^{(0,t)}$. We will call this problem *transformation maintenance* and the properties for this task will be properly defined in Section IV-A. We note that proofs in Section IV-A essentially follow ideas from [2], but Sankowski did not state his result in exactly the form that we need.

In the following two subsections IV-B and IV-C, we describe two algorithms for this transformation maintenance problem. We are able to combine these two algorithms to get an even faster transformation maintenance algorithm in subsection IV-D, where we will also prove the main results Theorem 4.1 and Theorem 4.2.

Throughout this section we will assume that $A^{(t)}$ is invertible for every t. An extension to the case where $A^{(t)}$ is allowed to become singular is given in the full version.

A. Transformation Maintenance implies Dynamic Matrix Inverse

In the overview Section II we outlined that maintaining the inverse for some transformation matrix $T^{(0,t)}$ implies an algorithm for maintaining the inverse of matrix $A^{(t)}$. In this section we will formalize and prove this claim in the setting where $A^{(t)}$ receives entry updates. While the idea of using a transformation matrix $T^{(0,t)}$ goes back to [2], there was no general reduction given that reduces the dynamic matrix inverse to the transformation maintenance problem. Here we state and prove such a reduction:

Theorem 4.3. *Assume there exists a dynamic algorithm* $\mathcal{T}$ *that maintains the inverse of an* $n \times n$ *matrix* $M^{(t)}$ *where* $M^{(0)} = \mathbb{I}$*, supporting the following operations:*

- update($j_1, ..., j_k$,$c_1, ..., c_k$) *Set the* j_l*th column of* $M^{(t)}$ *to be the vector* c_l *for* $l = 1...k$ *in* $O(u(k, m))$ *field operations, where* m *is the number of so far changed columns.*
- query(I) *Output the rows of* $(M^{(t)})^{-1}$ *specified by the set* $I \subset [n]$ *in* $O(q(|I|, m))$ *field operations, where* m *is the number of so far changed columns.*

Also assume the pre-processing of this algorithm requires $O(p)$ *field operations.*

Let $k \leq n^{\varepsilon}$ *for* $0 \leq \varepsilon \leq 1$*, then there exists a dynamic algorithm* $\mathcal{A}$ *that maintains the inverse of any (non-singular) matrix* A *supporting the following operations:*

- update($(i_1, j_1)...(i_k, j_k), c_1...c_k$) *Set* $A^{(t)}_{i_l,j_l}$ *to be* c_l *for* $l = 1...k$ *in* $O(u(k, n^{\varepsilon}) + (kn^{-\varepsilon}) \cdot (p + n^{\omega(1,1,\varepsilon)}))$ *field operations.*
- query(I, J) *Output the sub-matrix* $(A^{(t)})^{-1}_{I,J}$ *specified by the sets* $I, J \subset [n]$ *with* $|I| = n^{\delta_1}, |J| = n^{\delta_2}$ *in* $O(q(n^{\delta_1}, n^{\varepsilon}) + n^{\omega(\delta_1,\varepsilon,\delta_2)})$ *field operations.*

The pre-processing requires $O(p + n^{\omega})$ *field operations.*

The high level idea of the algorithm $\mathcal{A}$ is to maintain $T^{(0,t)}$ such that $A^{(t)} = A^{(0)}T^{(0,t)}$, which allows us to express the inverse of $A^{(t)}$ via $(T^{(0,t)})^{-1}(A^{(0)})^{-1}$. Here the matrix $(A^{(0)})^{-1}$ is computed during the pre-processing and $(T^{(0,t)})^{-1}$ is maintained via the assumed algorithm $\mathcal{T}$. After changing n^{ε} entries of A, we reset the algorithm by computing $(A^{(t)})^{-1}$ explicitly and resetting $T^{(t,t)} = \mathbb{I}$. We will first prove that element updates to A correspond to column updates to T.

Lemma 4.4. *Let* $A^{(t_1)}$ *and* $A^{(t_2)}$ *be two non-singular matrices, then there exists a matrix* $T^{(t_1,t_2)} := \mathbb{I} + (A^{(t_1)})^{-1}(A^{(t_2)} - A^{(t_1)})$ *such that* $A^{(t_2)} = A^{(t_1)}T^{(t_1,t_2)}$.

Proof: We have $T^{(t_1,t_2)} = \mathbb{I} + (A^{(t_1)})^{-1}(A^{(t_2)} - A^{(t_1)})$, because:

$$\begin{aligned} &A^{(t_1)}\left[\mathbb{I} + (A^{(t_1)})^{-1}(A^{(t_2)} - A^{(t_1)})\right] \\ =&A^{(t_1)} + (A^{(t_2)} - A^{(t_1)}) = A^{(t_2)} \end{aligned}$$

■

Corollary 4.5. *Let* $0 \leq t' \leq t$ *and* $A^{(t)} = A^{(t')}T^{(t',t)}$*, where* $A^{(t')}$ *and* $A^{(t)}$ *differ in at most* k *columns. Then*

- *An entry update to* $A^{(t)}$ *corresponds to a column update to* $T^{(t',t)}$*, where the column update is given by a column of* $(A^{(t')})^{-1}$*, multiplied by some scalar.*
- *The matrix* $T^{(t',t)}$ *is of the form* $\mathbb{I} + C$*, where* C *has at most* k *non-zero columns.*

Proof: The first property comes from the fact that

$$\begin{aligned} T^{(t',t)} &= \mathbb{I} + (A^{(t')})^{-1}(A^{(t)} - A^{(t')}) \\ &= \mathbb{I} + (A^{(t')})^{-1}(A^{(t)} - A^{(t-1)} + A^{(t-1)} - A^{(t')}) \\ &= (A^{(t')})^{-1}(A^{(t)} - A^{(t-1)}) + \mathbb{I} \\ &\qquad + (A^{(t')})^{-1}(A^{(t-1)} - A^{(t')}) \\ &= (A^{(t')})^{-1}(A^{(t)} - A^{(t-1)}) + T^{(t',t-1)} \end{aligned}$$

and $(A^{(t)} - A^{(t-1)})$ is a zero matrix except for a single entry. Thus $(A^{(t')})^{-1}(A^{(t)} - A^{(t-1)})$ is just one column of $(A^{(t')})^{-1}$ multiplied by the non-zero entry of $(A^{(t)} - A^{(t-1)})$.

The second property is a direct implication of $T^{(t',t)} = \mathbb{I} + (A^{(t')})^{-1}(A^{(t)} - A^{(t')})$ as $(A^{(t)} - A^{(t')})$ is non-zero in at most k columns. ■

Proof of Theorem 4.3: We are given a dynamic algorithm $\mathcal{T}$ that maintains the inverse of an $n \times n$ matrix $M^{(t)}$ where $M^{(0)} = \mathbb{I}$, supporting column updates to $M^{(t)}$

and row queries to $(M^{(t)})^{-1}$. We now want to use this algorithm to maintain $(A^{(t)})^{-1}$.

Pre-processing: During the pre-processing we compute $(A^{(0)})^{-1}$ explicitly in $O(n^\omega)$ field operations and initialize the algorithm $\mathcal{T}$ in $O(p)$ operations.

Updates: We use algorithm $\mathcal{T}$ to maintain the inverse of $M^{(t)} := T^{(0,t)}$, where $T^{(0,t)}$ is the linear transformation transforming $A^{(0)}$ to $A^{(t)}$. Via Corollary 4.5 we know the updates to $A^{(t)}$ imply column updates to $T^{(0,t)}$, so we can use algorithm $\mathcal{T}$ for this task. Corollary 4.5 also tells us that the update performed to $M^{(t)} = T^{(0,t)}$ is simply given by a scaled column of $(A^{(0)})^{-1}$, so it is easy to obtain the change we have to perform to $M^{(t)}$.

Reset and average update cost: For the first n^ε columns that are changed in $T^{(0,t)}$, each update requires at most $O(u(k, n^\varepsilon))$ field operations. After changing n^ε columns we reset our algorithm, but instead of computing the inverse of $A^{(t)}$ explicitly in $O(n^\omega)$ as in the pre-processing, we compute it by first computing $(T^{(0,t)})^{-1}$ and then multiplying $(T^{(0,t)})^{-1}(A^{(0)})^{-1}$. Note that $T^{(0,t)}$ is of the form $I + C$ where C has at most n^ε nonzero columns, so its inverse $(T^{(0,t)})^{-1}$ can be computed explicitly in $O(n^{\omega(1,\varepsilon,\varepsilon)})$ operations (see Fact 3.1). This inverse $(T^{(0,t)})^{-1}$ is of the same form $I+C'$, hence the multiplication of $(T^{(0,t)})^{-1}(A^{(0)})^{-1}$ costs only $O(n^{\omega(1,\varepsilon,1)})$ field operations (via Fact 3.2) and the average update time becomes $O(u(k, n^\varepsilon) + kn^{-\varepsilon}(p + n^{\omega(1,1,\varepsilon)}))$, which for a fixed batch-size k (i.e. all updates are of the same size) can be made worst-case via standard techniques.

Queries: When querying a submatrix $(A^{(t)})^{-1}_{I,J}$ we simply have to compute the product of the rows I of $(T^{(0,t)})^{-1}$ and columns J of $(A^{(0)})^{-1}$. To get the required rows of $(T^{(0,t)})^{-1} = (M^{(t)})^{-1}$ we need $O(q(n^{\delta_1}, n^\varepsilon))$ time via algorithm $\mathcal{T}$. Because of the structure $(T^{(0,t)})^{-1} = \mathbb{I}+C$, where C has only upto n^ε nonzero columns, the product of the rows I of $(T^{(0,t)})^{-1}$ and the columns J of $(A^{(0)})^{-1}$ needs $O(n^{\omega(\delta_1,\varepsilon,\delta_2)})$ field operations (Fact 3.2). ∎

B. Explicit Transformation Maintenance

In the previous subsection we motivated that a *dynamic matrix inverse* algorithm can be constructed from a *transformation maintenance* algorithm.

The following algorithm allows us to quickly compute the inverse of a transformation matrix, if only a few are columns changed. The algorithm is identical to [2, Theorem 2] by Sankowski, for maintaining the inverse of any matrix. Here we analyze the complexity of his algorithm for the setting that the algorithm is applied to a transformation matrix instead.

Lemma 4.6. *Let $0 \le \varepsilon_0 \le \varepsilon_1 \le 1$ and let $T = \mathbb{I} + N$ be an $n \times n$ matrix where N has at most n^{ε_1} non-zero columns. Let C be a matrix with at most n^{ε_0} non-zero columns. If the inverse T^{-1} is already known, then we can compute the inverse of $T' = T + C$ in $O(n^{\omega(1,\varepsilon_1,\varepsilon_0)})$ field operations.*

Figure 8. UpdateColumnsInverse (Lemma 4.6)

Input: $n \times n$ matrices T^{-1} and C
Output: $(T + C)^{-1}$
1: $M \leftarrow \mathbb{I} + T^{-1}C$
2: $M^{-1} \leftarrow$ INVERT(M) (Fig. 6)
3: **return** $M^{-1}T^{-1}$

For the special case $\varepsilon_1 = 1$, $\varepsilon_0 = 0$ this result is identical to [2, Theorem 1], while for $\varepsilon_1 = 1$ this result is identical to [2, Theorem 2]. For $\varepsilon_0 = 0, \varepsilon_1 < 1$ this result is implicitly proven inside the proof of [2, Theorem 3]. Thus Lemma 4.6 unifies half the results of [2].

Note that for $\varepsilon_0 = 0$ the complexity simplifies to $O(n^{1+\varepsilon_1})$ field operations.

Proof of Lemma 4.6: The change from T to $T + C$ can be expressed as some linear transformation M:

$$T + C = T(\underbrace{\mathbb{I} + T^{-1}C}_{=:M})$$

Here C has at most n^{ε_0} non-zero columns and T^{-1} is of the form $\mathbb{I} + N$, where N has at most n^{ε_1} non-zero columns, so the matrix $M = \mathbb{I} + T^{-1}C$ can be computed in $O(n^{\omega(1,\varepsilon_1,\varepsilon_0)})$ field operations, see Fact 3.2.

The new inverse $(T + C)^{-1}$ is given by $(TM)^{-1} = M^{-1}T^{-1}$. Note that M is of form $\mathbb{I} + N$, where N has n^{ε_0} non-zero columns, so using Fact 3.1 (Fig. 6) we can compute M^{-1} in $O(n^{\omega(1,\varepsilon_1,\varepsilon_0)})$ field operations. Via Fact 3.1 we also know that M^{-1} is again of the form $I + N$, where N has at most n^{ε_0} non-zero columns, thus the product $(T + C)^{-1} = M^{-1}T^{-1}$ requires $O(n^{\omega(1,\varepsilon_1,\varepsilon_0)})$ field operations (Fact 3.2). In total we require $O(n^{\omega(1,\varepsilon_1,\varepsilon_0)})$ operations. ∎

C. Implicit Transformation Maintenance

In this section we will describe an algorithm for maintaining the inverse of a transformation matrix $T^{(t_1,t_2)}$ in an implicit form, that is, the entries of $(T^{(t_1,t_2)})^{-1}$ are not computed explicitly, but they can be queried.

We state this result in a more general way: Let $B^{(t)}$ be a matrix that receives column updates and where initially $B^{(0)} = \mathbb{I}$. Thus $B^{(t)}$ is a matrix that differs from $\mathbb{I}$ in only a few columns. As seen in equation (8) and Fact 3.1 such a matrix allows us to compute rows of its inverse $(B^{(t)})^{-1}$ without knowing the entire matrix $B^{(t)}$. Thus we do not require the matrix $B^{(t)}$ to be given in an explicit way, instead it is enough to give the matrix $B^{(t)}$ via some pointer to a data-structure D_B. Our algorithm will then query this data-structure D_B to obtain entries of $B^{(t)}$.

Lemma 4.7. *Let $B^{(t)}$ be a matrix receiving column updates where initially $B^{(0)} = \mathbb{I}$ and let $0 \le \varepsilon_0 \le \varepsilon_1 \le 1$. Here n^{ε_0} is an upper bound on the number of columns changed per update and n^{ε_1} is an upper bound on the number of columns where $B^{(t)}$ differs from the identity (e.g. via restricting*

Figure 9. UpdateInverse (Lemma 4.8)

Input: $n \times n$ matrices M, C, R.
Output: $(M + C + R)^{-1}$

1: $M^{-1} \leftarrow$ UPDATECOLUMNSINVERSE(M, M^{-1}, C) (Fig. 8)
2: $M \leftarrow M + C$
3: $M^{-1} \leftarrow$ (UPDATECOLUMNSINVERSE$(M^\top, M^{\top -1}, R^\top))^\top$ (Fig. 8)
4: **return** M^{-1}

$t \leq n^{\varepsilon_1 - \varepsilon_0}$). Assume matrix $B^{(t)}$ is given via some data-structure D that supports the method D.QUERY(I, J) to obtain any submatrix $B^{(t)}_{I,J}$.

Then there exists a transformation maintenance algorithm which maintains $(B^{(t)})^{-1}$ supporting the following operations:

- update($J^{(t)}$): *The set $J^{(t)} \subset [n]$ specifies the column indices where $B^{(t)}$ and $B^{(t-1)}$ differ.*
  The algorithm updates its internal data-structure using at most $O(n^{\omega(\varepsilon_1,\varepsilon_1,\varepsilon_0)})$ field operations. To perform this update, the algorithm has to query D to obtain two submatrices of $B^{(t)}$ of size $n^{\varepsilon_0} \times n^{\varepsilon_1}$ and $n^{\varepsilon_1} \times n^{\varepsilon_0}$.
- query(I) *The algorithm outputs the rows of $(B^{(t)})^{-1}$, specified by $I \subset [n]$, $|I| = n^\delta$ in $O(n^{\omega(\delta,\varepsilon_1,\varepsilon_1)})$ field operations. To perform the query, the algorithm has to query D to obtain a submatrix of $B^{(t)}$ of size $n^\delta \times n^{\varepsilon_1}$.*

The algorithm requires no pre-processing.

While our algorithm of Lemma 4.7 is new, in a restricted setting it has the same complexity as the transformation maintenance algorithm used in [2, Theorem 4]. When restricting to the setting where the matrix $B^{(t)}$ is given explicitly and no batch updates/queries are performed (i.e. $\varepsilon_0 = \delta = 0$), then the complexity of Lemma 4.7 is the same as the transformation maintenance algorithm used in [2, Theorem 4]. [17]

Before we prove Lemma 4.7, we will prove the following lemma, which is implied by Fact 3.1. This lemma allows us to quickly invert matrices when the matrix is obtained from changing few rows *and* columns.

Lemma 4.8. *Let $0 \leq \varepsilon_0 \leq \varepsilon_1 \leq 1$ and let M, C, R be square matrices of size at most $n^{\varepsilon_1} \times n^{\varepsilon_1}$.*

If C has at most n^{ε_0} non-zero columns, R has at most n^{ε_0} non-zero rows and we already know the inverse M^{-1}, then we can compute the inverse $(M + C + R)^{-1}$ in $O(n^{\omega(\varepsilon_1,\varepsilon_1,\varepsilon_0)})$ field operations.

[17] Our algorithm is slightly faster for the setting of batch updates and batch queries (i.e. more than one column is changed per update or more than one row is queried at once). When considering batch updates and batch queries, Sankowski's variant of Lemma 4.7 can be extended to have the complexity $\Omega(n^{\omega(\varepsilon_1,\varepsilon_0,\varepsilon_0)+\varepsilon_1-\varepsilon_0})$ and $\Omega(n^{\omega(\varepsilon_1,\delta,\varepsilon_0)+\varepsilon_1-\varepsilon_0})$ operations, because all internal computations are successive and can not be properly combined/batched using fast-matrix-multiplication.

Proof: We want to compute $(M + C + R)^{-1}$, where R has at most n^{ε_0} non-zero rows and C has at most n^{ε_0} non-zero columns.

Let $m = n^{\varepsilon_1}$, $\delta_1 = 1$ and $\delta_0 = \varepsilon_0/\varepsilon_1$, then M, C, R are $m \times m$ matrices.

We can compute $(M + C)^{-1}$ via Lemma 4.6 (Fig. 8) in $O(m^{\omega(1,\delta_1,\delta_0)}) = O(n^{\omega(\varepsilon_1,\varepsilon_1,\varepsilon_0)})$ operations.

Let $B = M+C$, then $(M+C+R)^\top = B^\top + R^\top$ so $M^\top$ is obtained from $B^\top$ by changing at most n^{ε_0} columns and we can use Lemma 4.6 (Fig. 8) again to obtain $(M + C + R)^{-1} = ((B^\top + R^\top)^{-1})^\top$ using $O(n^{\omega(\varepsilon_1,\varepsilon_1,\varepsilon_0)})$ operations. ∎

With the help of Lemma 4.8 (Fig. 9), we can now prove Lemma 4.7. The high level idea is to see the matrix $B^{(t)}$ to be of the form $I + C$ similar to equation (8) in the overview (Section II), then we only maintain the C_1^{-1} block during the updates. When performing queries, we then may have to compute some rows of the product $-C_2 C_1^{-1}$.

Proof of Lemma 4.7: Let $J^{(i)}$ be the set we received at the ith update. At time t let $I^{(t)} = \bigcup_{i=1}^{t} J^{(i)}$ be the set of column indices of all so far changed columns, and let $M^{(t)}$ be the matrix s.t. $M^{(t)}_{I^{(t)},I^{(t)}} = (B^{(t)})_{I^{(t)},I^{(t)}}$ and $M^{(t)}_{i,j} = \mathbb{I}_{i,j}$ otherwise. We will maintain $I^{(t)}$, $M^{(t)}$ and $(M^{(t)})^{-1}$ explicitly throughout all updates.

For $t = 0$ we have $B^{(0)} = \mathbb{I}$ and $I^{(0)} = \emptyset$, $M^{(0)} = \mathbb{I} = (M^{(0)})^{-1}$, so no pre-processing is required.

Updating $I^{(t)}$ and $M^{(t)}$: When $B^{(t)}$ is "updated" (i.e. we receive a new set J), we set $I^{(t)} = I^{(t-1)} \cup J$. As J specifies the columns in which $B^{(t)}$ differs to $B^{(t-1)}$, we query D to obtain the entries $B^{(t)}_{I^{(t)},J}$ and $B^{(t)}_{J,I^{(t)}}$ and update these entries in $M^{(t)}$ accordingly. Thus we now have $M^{(t)}_{I^{(t)},I^{(t)}} = B^{(t)}_{I^{(t)},I^{(t)}}$.

The size of the queried submatrices $B^{(t)}$ is at most $n^{\varepsilon_0} \times n^{\varepsilon_1}$ and $n^{\varepsilon_1} \times n^{\varepsilon_0}$, because by assumption at most n^{ε_1} columns are changed in total (so $|I^{(t)}| \leq n^{\varepsilon_1}$) and at most n^{ε_0} columns are changed per update (so $|J| \leq n^{\varepsilon_0}$).

Updating $(M^{(t)})^{-1}$: Next, we have to compute $(M^{(t)})^{-1}$ from $(M^{(t-1)})^{-1}$. Note that the matrix $M^{(t)}$ is equal to the identity except for the submatrix $M^{(t)}_{I^{(t)},I^{(t)}}$, i.e. without loss of generality (after reordering rows/columns) $M^{(t)}$ and its inverse look like this:

$$M^{(t)} = \begin{pmatrix} \mathbb{I} & 0 \\ 0 & M^{(t)}_{I^{(t)},I^{(t)}} \end{pmatrix}$$

$$(M^{(t)})^{-1} = \begin{pmatrix} \mathbb{I} & 0 \\ 0 & (M^{(t)}_{I^{(t)},I^{(t)}})^{-1} \end{pmatrix}$$

So we have $(M^{(t)}_{I^{(t)},I^{(t)}})^{-1} = ((M^{(t)})^{-1})_{I^{(t)},I^{(t)}}$, and most importantly $M^{(t)}_{I^{(t)},I^{(t)}}$ is obtained from $M^{(t-1)}_{I^{(t)},I^{(t)}}$ by changing upto n^{ε_0} rows and columns. We already know the inverse $(M^{(t-1)}_{I^{(t)},I^{(t)}})^{-1} = ((M^{(t-1)})^{-1})_{I^{(t)},I^{(t)}}$, hence we can compute $(M^{(t)})^{-1}$ via Lemma 4.8 (Fig. 9) using $O(n^{\omega(\varepsilon_1,\varepsilon_1,\varepsilon_0)})$

Figure 10. MaintainTransform (Lemma 4.7)

Input: Data-structure D representing the matrix $B^{(t)}$ throughout all updates. We can call some function $D.\text{QUERY}(I, J)$ to receive $B^{(t)}_{I,J}$. In each update we also receive a set $J \subset [n]$ specifying the column indices where $B^{(t)}$ and $B^{(t-1)}$ differ.

Maintain: $(B^{(t)})^{-1}$ in an implicit form (rows can be queried). Internally we maintain:

- $I^{(t)} = \bigcup_{i=1}^{t} J^{(i)} \subset [n]$, where $J^{(i)}$ is the set we received at the ith update.
- An $n \times n$ matrix $M^{(t)}$ s.t. $M^{(t)}_{I^{(t)},I^{(t)}} = (B^{(t)})_{I^{(t)},I^{(t)}}$ and $M^{(t)}_{i,j} = \mathbb{I}_{i,j}$ for all other entries (i, j)
- The inverse $(M^{(t)})^{-1}$.

initialization: (We receive the data-structure D)

INITIALIZE(D):

1: $t \leftarrow 0, M^{(0)} \leftarrow \mathbb{I}$, $J^{(0)} \leftarrow \emptyset$.
2: Remember D

update operation: (We receive $J \subset [n]$)

UPDATE(J):

1: $t \leftarrow t + 1$
2: $I^{(t)} \leftarrow I^{(t-1)} \cup J$
3: Update $M^{(t)}$ {This requires to query $B^{(t)}_{J,I^{(t)}}$ and $B^{(t)}_{I^{(t)},J}$ by calling $D.\text{QUERY}(J, I^{(t)})$ and $D.\text{QUERY}(I^{(t)}, J)$}
4: {We will now compute two matrices R, C s.t. $M^{(t)} = M^{(t-1)} + R + C$}
5: $R, C \leftarrow$ 0-matrices
6: $R_{J,I^{(t)}} \leftarrow M^{(t)}_{J,I^{(t)}} - M^{(t-1)}_{J,I^{(t)}}$
7: $C_{I^{(t)}\setminus J,J} \leftarrow M^{(t)}_{I^{(t)}\setminus J,J} - M^{(t-1)}_{I^{(t)}\setminus J,J}$
8: $(M^{(t)})^{-1} \leftarrow \mathbb{I}$
9: $((M^{(t)})^{-1})_{I^{(t)},I^{(t)}} \leftarrow \text{UPDATEINVERSE}(M^{(t-1)}_{I^{(t)},I^{(t)}}, ((M^{(t-1)})^{-1})_{I^{(t)},I^{(t)}}, R_{I^{(t)},I^{(t)}}, C_{I^{(t)},I^{(t)}})$ (Fig. 9)

{Note that $((M^{(t)})^{-1})_{I^{(t)},I^{(t)}} = (M^{(t)}_{I^{(t)},I^{(t)}})^{-1} = (B^{(t)}_{I^{(t)},I^{(t)}})^{-1} = ((B^{(t)})^{-1})_{I^{(t)},I^{(t)}}$, because the matrices differ only in columns $I^{(t)}$ from the identity matrix, see Fact 3.1.}

query operation: (Querying some rows with index $J \subset [n]$ of $(B^{(t)})^{-1}$)

QUERY(J):

1: $N \leftarrow \mathbb{I}$
2: Obtain $B^{(t)}_{J\setminus I^{(t)},I^{(t)}}$ by calling $D.\text{QUERY}(J\setminus I^{(t)}, I^{(t)})$.
3: $N_{J\setminus I^{(t)},I^{(t)}} \leftarrow -B^{(t)}_{J\setminus I^{(t)},I^{(t)}}(M^{(t)})^{-1}_{I^{(t)},I^{(t)}}$
4: $N_{I^{(t)},I^{(t)}} \leftarrow (M^{(t)})^{-1}_{I^{(t)},I^{(t)}}$
5: **return** rows J of N.

operations.

This concludes all performed computations during an update. The total cost is $O(n^{\omega(\varepsilon_1,\varepsilon_1,\varepsilon_0)})$ field operations.

Queries: Next we will explain the query routine, when trying to query rows with index $J \subset [n]$ of the inverse. Remember that $B^{(t)}$ is of the form of Fact 3.1, i.e. $B^{(t)} = \mathbb{I} + C$, where the non-zero columns of C have their indices in $I^{(t)}$. Thus we have $((B^{(t)})^{-1})_{I^{(t)},I^{(t)}} = ((B^{(t)}_{I^{(t)},I^{(t)}})^{-1})$ and $((B^{(t)})^{-1})_{[n]\setminus I^{(t)},I^{(t)}} = -B^{(t)}_{[n]\setminus I^{(t)},I^{(t)}}(B^{(t)}_{I^{(t)},I^{(t)}})^{-1}$.

This means by setting some matrix $N = \mathbb{I}$ except for the submatrix $N_{J,I^{(t)}}$, where $N_{J\setminus I^{(t)},I^{(t)}} := -B^{(t)}_{J\setminus I^{(t)},I^{(t)}}(B^{(t)}_{I^{(t)},I^{(t)}})^{-1} = -B^{(t)}_{J\setminus I^{(t)},I^{(t)}}(M^{(t)})^{-1}_{I^{(t)},I^{(t)}}$ and $N_{J\cup I^{(t)},I^{(t)}} := (M^{(t)})^{-1}_{J\cup I^{(t)},I^{(t)}}$, then rows J of N and rows J of $(B^{(t)})^{-1}$ are identical, so we can simply return these rows of N.

The query complexity is as follows:

The required submatrix $B^{(t)}_{I\setminus I^{(t)},I^{(t)}}$ is queried via $D^{(t)}_{B^{(t)}}$ and is of size at most $n^{\delta} \times n^{\varepsilon_1}$. The product $-B^{(t)}_{J,I^{(t)}}(M^{(t)})^{-1}_{I^{(t)},I^{(t)}}$ requires $O(n^{\omega(\delta,\varepsilon_1,\varepsilon_2)})$ field operations via Fact 3.2. ∎

D. Combining the Transformation Maintenance Algorithms

The task of maintaining the transformation matrix can itself be interpreted as a dynamic matrix inverse algorithm, where updates change columns of some matrix $T^{(0,t)}$ and queries return rows of $(T^{(0,t)})^{-1}$. This means the trick of maintaining $(A^{(t)})^{-1} = (T^{(t',t)})^{-1}(A^{(t')})^{-1}$ for $t \geq t'$ can also be used to maintain $(T^{(0,t)})^{-1}$ in the form $(T^{(0,t)})^{-1} = (T^{(t',t)})^{-1}(T^{(0,t')})^{-1}$ instead.

This is the high-level idea of how we obtain the following Lemma 4.9 via Lemma 4.6 and Lemma 4.7. We use Lemma 4.6 to maintain $(T^{(0,t')})^{-1}$ and Lemma 4.7 to maintain $(T^{(t',t)})^{-1}$.

We will state the new algorithm as maintaining the inverse of some matrix $B^{(t)}$ where $B^{(t)}$ receives column updates. Note that the following result is slightly more general than maintaining the inverse of some $T^{(0,t)}$ as we do *not* require $B^{(t)} = \mathbb{I}$.

Lemma 4.9. *Let $0 \leq \varepsilon_0 \leq \varepsilon_1 \leq \varepsilon_2 \leq 1$ and $k = n^{\varepsilon_0}$.*

There exists a transformation maintenance algorithm that maintains the inverse of $B^{(t)}$, supporting column updates to $B^{(t)}$ and submatrix queries to the inverse $(B^{(t)})^{-1}$. Assume that throughout the future updates the form of $B^{(t)}$ is $\mathbb{I} + C^{(t)}$, where $C^{(t)}$ has always at most n^{ε_2} nonzero columns (e.g. by restricting the number of updates $t \leq n^{\varepsilon_2-\varepsilon_0}$). The complexities are:

- update($j_1, ..., j_k, c_1, ..., c_k$)*: Set the columns j_l of $B^{(t)}$ to be c_l for $l = 1, ..., k$ in $O(n^{\omega(\varepsilon_2,\varepsilon_1,\varepsilon_0)} + n^{\omega(1,\varepsilon_2,\varepsilon_1)-\varepsilon_1+\varepsilon_0})$ field operations.*
- query(I,J)*: Output the submatrix $(B^{(t)})^{-1}_{I,J}$ where $I, J \subset [n]$, $|I| = n^{\delta_1}$, $|J| = n^{\delta_2}$ in $O(n^{\omega(\delta_1,\varepsilon_2,\varepsilon_1)} + n^{\omega(\delta_1,\min\{\varepsilon_2,\delta_2\},\varepsilon_1)})$ field operations.*

The pre-processing requires at most $O(n^\omega)$ operations, though if $B^{(0)} = \mathbb{I}$ the algorithm requires no pre-processing.

Before proving Lemma 4.9 we want to point out that both Theorems 4.1 and 4.2 are direct implications of Lemma 4.9:

Proof of Theorem 4.2 and Theorem 4.1:

The column update algorithm from Theorem 4.1 is obtained by letting $\varepsilon_0 = 0$, $\varepsilon_1 = \varepsilon$, $\varepsilon_2 = \delta_2 = 1$ and $\delta_1 = 0$ in Lemma 4.9.

Theorem 4.2 is obtained by combining Theorem 4.3 and Lemma 4.9: Theorem 4.3 explains how a transformation maintenance algorithm can be used to obtain an element update dynamic matrix inverse algorithm and we use the algorithm from Lemma 4.9 as the transformation maintenance algorithm.

To summarize Theorem 4.3, it says that: Assume there exists an algorithm for maintaining T^{-1} where $T = \mathbb{I}$ initially then T receives n^{ε_0} column changes per update such that T stays of the form $\mathbb{I} + C$ where C has at most n^{ε_2} columns. If the update time is $u(n^{\varepsilon_0}, n^{\varepsilon_2})$ and the query time (for querying n^{δ_1} rows) is $q(n^{\delta_1}, n^{\varepsilon_2})$, then there exists an element update dynamic matrix inverse algorithm that supports changing n^{ε_0} elements per update and update time $O(u(n^{\varepsilon_0}, n^{\varepsilon_2}) + (n^{-\varepsilon_2+\varepsilon_0}) \cdot (p + n^{\omega(1,1,\varepsilon_2)}))$.

For $u(n^{\varepsilon_0}, n^{\varepsilon_2}) = O(n^{\omega(\varepsilon_2,\varepsilon_1,\varepsilon_0)} + n^{\omega(1,\varepsilon_2,\varepsilon_1)-\varepsilon_1+\varepsilon_0})$ and no pre-processing time p as in Lemma 4.9, we obtain with $\varepsilon_0 = 0$ the update complexity of Theorem 4.2 $O(n^{\varepsilon_2+\varepsilon_1} + n^{\omega(1,\varepsilon_1,\varepsilon_2)-\varepsilon_1} + n^{\omega(1,1,\varepsilon_2)-\varepsilon_2})$.

The query time of Theorem 4.3 for querying an element of T^{-1} is with $q(n^{\delta_1}, n^{\varepsilon_2}) = O(n^{\omega(\delta_1,\varepsilon_2,\varepsilon_1)})$ given via $O(q(1, n^{\varepsilon_2}) + n^{\omega(0,\varepsilon_2,0)}) = O(n^{\varepsilon_2+\varepsilon_1})$.

■

Next, we will prove Lemma 4.9.

Proof of Lemma 4.9:

Let $B^{(t)}$ be the matrix at round t, i.e. $B^{(0)}$ is what the matrix looks like at the time of the initialization/pre-processing. As pre-processing we compute $(B^{(0)})^{-1}$, which can be done in $O(n^\omega)$ operations, though for $B^{(0)} = \mathbb{I}$ this can be skipped since $(B^{(0)})^{-1} = \mathbb{I}$.

We implicitly maintain $B^{(t)}$ by maintaining another matrix $T^{(t',t)}$ such that $B^{(t)} = B^{(t')}T^{(t',t)}$ for some $t' \leq t$, so $(B^{(t)})^{-1} = (T^{(t',t)})^{-1}(B^{(t')})^{-1}$. The matrix $(B^{(t')})^{-1}$ is maintained via Lemma 4.6 while $(T^{(t',t)})^{-1}$ is maintained via Lemma 4.7. After a total of n^{ε_1} columns were changed (e.g. when t is a multiple of $n^{\varepsilon_1-\varepsilon_0}$), we set $t' = t$, which means $B^{(t')}$ receives an update that changes upto n^{ε_1} columns. Additionally the matrix $T^{(t',t)}$ is reset to be the identity matrix and the algorithm from Lemma 4.7 is reset as well.

Maintaining $(B^{(t')})^{-1}$: The matrix $(B^{(t')})^{-1}$ is maintained in an explicit form via Lemma 4.6, which requires $O(n^{\omega(1,\varepsilon_2,\varepsilon_1)})$ operations. As this happens every $n^{-\varepsilon_1+\varepsilon_0}$ rounds, the cost for this is $O(n^{\omega(1,\varepsilon_2,\varepsilon_1)-\varepsilon_1+\varepsilon_0})$ operations on average per update. (This can be made worst case via standard techniques, see for example the full version of this paper.)

Maintaining $(T^{(t',t)})^{-1}$: We now explain how $(T^{(t',t)})^{-1}$ is maintained via Lemma 4.7. We have $B^{(t)} = B^{(t')}T^{(t',t)}$ which means the matrix $T^{(t',t)}$ is of the following form (this can be seen by multiplying both sides with $B^{(t')}$):

$$T^{(t',t)} = \mathbb{I} + (B^{(t')})^{-1}(B^{(t)} - B^{(t')})$$

We do not want to compute this product explicitly, instead we construct a simple data-structure D (Fig. 12) to represent $T^{(t',t)} = \mathbb{I} + (B^{(t')})^{-1}(B^{(t)} - B^{(t')})$. (Note that in the algorithmic description Fig. 11 the matrix $S^{(t)} = B^{(t)} - B^{(t')}$.) This data-structure allows queries to submatrices of $T^{(t',t)}$, by computing a small matrix product. More accurately, for any set $I, J \subset [n]$ calling $D.\text{QUERY}(I, J)$ to obtain $T^{(t',t)}_{I,J}$ requires to compute the product $((B^{(t')})^{-1})_{I,[n]}(B^{(t)} - B^{(t')})_{[n],J}$. Since $B^{(t')}$ (and thus also $(B^{(t')})^{-1}$), see Fact 3.1) is promised to be of the form $I + C$, where C has at most n^{ε_2} non-zero columns, querying this new data-structure for $|I| = n^a, |J| = n^b$ requires $O(n^{\omega(a,\varepsilon_2,b)})$ field operations for any $0 \leq a, b \leq 1$.

When applying Lemma 4.7 to maintain $(T^{(t',t)})^{-1}$, the update complexity is bounded by $O(n^{\omega(\varepsilon_1,\varepsilon_1,\varepsilon_0)} + n^{\omega(\varepsilon_0,\varepsilon_2,\varepsilon_1)} + n^{\omega(\varepsilon_1,\varepsilon_2,\varepsilon_0)}) = O(n^{\omega(\varepsilon_2,\varepsilon_1,\varepsilon_0)})$.

The average update complexity for updating both $(T^{(t',t)})^{-1}$ and $(B^{(t')})^{-1}$ thus becomes $O(n^{\omega(\varepsilon_2,\varepsilon_1,\varepsilon_0)} + n^{\omega(1,\varepsilon_2,\varepsilon_1)-\varepsilon_1+\varepsilon_0})$.

Queries: Next, we will analyze the complexity of querying a submatrix $(B^{(t)})^{-1}_{I,J}$. To query such a submatrix, we need to multiply the rows I of $(T^{(t',t)})^{-1}$ with the columns J of $(B^{(t')})^{-1}$. Querying the rows of $(T^{(t',t)})^{-1}$ requires at most $O(n^{\omega(\delta,\varepsilon_2,\varepsilon_1)})$ operations according to Lemma 4.7 (querying entries of $T^{(t',t)}$ via data-structure D is the bottleneck). Note that $(B^{(t')})^{-1}$ is of the form $\mathbb{I} + C$ where C has at most n^{ε_2} nonzero columns, so for $|J| = n^{\delta_2}$ multiplying the rows and columns requires at most $O(n^{\omega(\delta_1,\varepsilon_1,\min\{\delta_2,\varepsilon_2\})})$ operations (see Fact 3.2).

■

E. Applications

There is a wide range of applications, which we summarized in Fig. 5 and the second applications table in the full version. The reductions can be found in the appendix of the full version, because some have already been stated before (e.g. [2], [3], [7], [37]) while many others are just known reductions for static problems applied to the dynamic setting.

In this section we want to highlight the most interesting applications, for an extensive list of all applications we refer to the full version.

Algebraic black box reductions: The dynamic matrix inverse algorithms from [2] can also be used to maintain the determinant, adjoint or solution of a linear system. However,

Figure 11. ColumnUpdateRowQuery (Lemma 4.9)

Input: An $n \times n$ matrix $B^{(0)} = \mathbb{I} + C^{(0)}$ and inputs of the form $B^{(t)} = T^{(t-1)} + C^{(t)}$ where $C^{(t)}$ has nonzero columns $J^{(t)}$.

Output: Maintain $(B^{(t)})^{-1}$ in an implicit form s.t. submatrices can be queried.

initialization:

INITIALIZE($B^{(0)}$)

1: Compute $(B^{(0)})^{-1}$ (or just set $(B^{(0)})^{-1} \leftarrow \mathbb{I}$ in case of $B^{(0)} = \mathbb{I}$)
2: $S^{(0)} \leftarrow$ zero-matrix
3: $t' \leftarrow 0$, $t \leftarrow 0$
4: D.UPDATE($(B^{(0)})^{-1}, S^{(0)}$) (Initialize data-structure D Fig. 12)
5: MAINTAINTRANSFORM.INITIALIZE(D) (Initialize Fig. 10 for $T^{(t',t)} = \mathbb{I}$)

update operation:

UPDATE(C)

1: $t \leftarrow t + 1$,
2: $S^{(t)} \leftarrow S^{(t-1)} + C$, $J^{(t)} \leftarrow$ indices of non-zero columns of C.
3: **if** $|\bigcup_{i=1}^{t} J^{(i)}| \geq n^{\varepsilon}$ **then**
4: $(B^{(t)})^{-1} \leftarrow$ UPDATECOLUMNSINVERSE($B^{(t')}, (B^{(t')})^{-1}, S^{(t)}$) (Fig. 8)
5: $t' \leftarrow t$
6: $S^{(t)} \leftarrow$ zero-matrix
7: D.UPDATE($(B^{(t)})^{-1}, S^{(t)}$) (Fig. 12)
8: MAINTAINTRANSFORM.INITIALIZE(D) (Reinitialize MAINTAINTRANSFORM Fig. 10 for $T^{(t',t)} = \mathbb{I}$)
9: **else**
10: D.UPDATE($(B^{(t')})^{-1}, S^{(t)}$) (Fig. 12)
11: MAINTAINTRANSFORM.UPDATE($J^{(t)}$) (Fig. 10).
12: **end if**

query operation: (Querying some submatrix $(B^{(t)})^{-1})_{I,J}$)

QUERY(I, J)

1: Obtain rows I of $(T^{(t',t)})^{-1}$ by calling MAINTAINTRANSFORM.QUERY($I, [n]$) (Fig. 10).
2: **return** $((T^{(t',t)})^{-1})_{I,[n]}((B^{(t')})^{-1})_{[n],J}$

these reductions are white box. In the static setting we already know, that determinant, adjoint and matrix inverse are equivalent and that we can solve a linear system via matrix inversion.However, not all static reductions can be translated to work in the dynamic setting. For example the Baur-Strassen theorem [38], [39] used to show the hardness of the determinant in the static setting can not be used in the dynamic setting. Likewise the typical reduction of linear system to matrix inversion does not work in the dynamic setting either. Usually one would solve $Ax = b$ by inverting A and computing the product $A^{-1}b$. However, in the dynamic setting the matrix A^{-1} is not explicitly given,

Figure 12. ProductDataStructure (Lemma 4.9)

(Used inside Fig. 10 and Fig. 11)

Input: Two $n \times n$ matrices A and B given via pointers.

Output: Maintain $P := \mathbb{I} + AB$ in an implicit form s.t. submatrices can be queried.

Update operation: Called if matrix A or B change.

UPDATE(A, B):

1: Remember the pointers to matrices A and B.

Query operation: Returns the submatrix $P_{I,J}$ for $I, J \subset [n]$.

QUERY(I, J):

1: $N \leftarrow \mathbb{I}$
2: $N_{I,J} \leftarrow N_{I,J} + A_{I,[n]}B_{[n],J}$
3: **return** $N_{I,J}$

Figure 13. ElementUpdate (Theorems 4.2 and 4.3)

Input: An $n \times n$ matrix $A^{(0)}$ and inputs of the form $A^{(t)} = A^{(t-1)} + C^{(t)}$ where $C^{(t)}$ has nonzero entries at $(i_1^{(t)}, j_1^{(t)})...(i_k^{(t)}, j_k^{(t)})$.

Output: Maintain $(A^{(t)})^{-1}$ in an implicit form s.t. submatrices can be queried.

initialization:

INITIALIZE($A^{(0)}$)

1: Compute $(A^{(0)})^{-1}$
2: $S^{(0)} \leftarrow$ zero-matrix
3: COMBINEDTRANSFORMATION.INITIALIZE($\mathbb{I}$) (Initialize Fig. 11 for $B^{(0)} = \mathbb{I}$)
4: $t \leftarrow 0$

update operation:

UPDATE(C)

1: $t \leftarrow t + 1$
2: $S^{(t)} \leftarrow S^{(t-1)} + C$, $J^{(t)} \leftarrow$ indices of the non-zero columns of C.
3: **if** $|\bigcup_{i=1}^{t} J^{(i)}| \geq n^{\varepsilon}$ **then**
4: $(A^{(0)})^{-1} \leftarrow$ UPDATECOLUMNSINVERSE($A^{(0)}, (A^{(0)})^{-1}, S^{(t)}$)
5: $S^{(0)} \leftarrow$ zero-matrix
6: $t \leftarrow 0$
7: COLUMNUPDATEROWQUERY.INITIALIZE($\mathbb{I}$) (Reinitialize Fig. 11 where $B^{(0)} := T^{(0,0)} = \mathbb{I}$)
8: **else**
9: $\tilde{C} \leftarrow (A^{(0)})^{-1}C$ (i.e. we select some columns of $(A^{(0)})^{-1}$)
10: COLUMNUPDATEROWQUERY.UPDATE($\tilde{C}$) (Update Fig. 11 where $B^{(t)} := T^{(0,t)}$)
11: **end if**

query operation: (Querying some submatrix $(A^{(t)})^{-1})_{I,J}$)

QUERY(I, J)

1: Query rows I of $(T^{(0,t)})^{-1}$ by calling COLUMNUPDATEROWQUERY.QUERY($I, [n]$) (Fig. 11)
2: **return** $((T^{(0,t)})^{-1})_{I,[n]}((A^{(0)})^{-1})_{[n],J}$

one would first have to query all entries of the inverse. Thus it is an interesting question, what the relationship of the dynamic versions of matrix inverse, determinant, adjoint, linear system is.

Can any dynamic matrix inverse algorithm be used to maintain determinant, adjoint, solution to a linear system, or was this a special property of the algorithms in [2]? Is the dynamic determinant easier in the dynamic setting, or is it as hard as the dynamic matrix inverse problem?

In the full version we are able to confirm the equivalence: dynamic matrix inverse, adjoint, determinant and linear system are all equivalent in the dynamic setting, i.e. there exist black box reductions that result in the same update time. This is also an interesting difference to the static setting, where there is no reduction from matrix inverse, determinant etc. to solving a linear system.

Results based on column updates: For many dynamic graph problems (e.g. bipartite matching, triangle detection, st-reachability) there exist $\Omega(n^2)$ lower bounds for dense graphs, when we allow node updates [18]. Thanks to the new column update dynamic matrix inverse algorithm we are able to achieve sub-$O(n^2)$ update times, even though we allow (restricted) node updates. For example the size of a maximum bipartite matching can be maintained in $O(n^{1.529})$, if we restrict the node updates to be only on the left or only on the right side. Likewise triangle detection and st-reachability can be maintained in $O(n^{1.529})$, if we restrict the node updates to change only outgoing edges. Especially for the dynamic bipartite matching problem this is a very interesting result, because often one side is fixed: Consider for example the setting where users have to be matched with servers, then the server infra-structure is rarely updated, but there are constantly users that will login/logout. Previously only for the incremental setting (i.e. no user will logout) there existed (amortized) sub-$O(n^2)$ algorithms [22]. The total time of [22] for n node insertions is $O(\sqrt{n}m)$, so $O(m/\sqrt{n}) = O(n^{1.5})$ amortized update time for dense graphs. In the full version we improve this to $O(n^{\omega-1})$.

V. Conditional Lower Bounds

In this section we will formalize the current barrier for dynamic matrix algorithms. We obtain conditional lower bounds for the trade-off between update and query time for column update/row query dynamic matrix inverse, which is the main tool of all currently known element update/element query algorithms by using them as transformation maintenance algorithms, see Theorem 4.3. The lower bounds we obtain (Corollary 5.5) are tight with our upper bounds when the query time is not larger than the update time. We also obtain worst-case lower bounds for element update and element query (Corollary 5.15 and Corollary 5.9), which are tight with our result Theorem 4.2 and Sankowski's result [2, Theorem 3]. The lower bounds are formalized in terms of dynamic matrix products over the boolean semi-ring and thus they also give lower bounds for dynamic transitive closure and related graph problems.

The conditional problems and conjectures defined in this section should be understood as questions. The presented problems are a formalization of the current barriers and the trade-off between using fast-matrix multiplication to precompute lots of information vs using slower matrix-vector multiplication to compute only required information in an online fashion. Our conjectures ask: Is there a better third option?

We will start this lower bound section with a short discussion of past lower bound results. Then we follow with three subsections, each giving tight bounds for a different type of dynamic matrix inverse algorithm. In last subsection V-D we will discuss, why other popular conjectures for dynamic algorithms are not able to capture the current barrier for dynamic matrix inverse algorithms.

Previous lower bounds: In [40] an unconditional linear lower bound is proven in the restricted computational model of algebraic circuits (*history dependent algebraic computation trees*) for the task of dynamically maintaining the product of two matrices supporting element updates and element queries. Via some reduction the lower bound then also holds for the dynamic matrix inverse. Using a reduction from dynamic matrix product to dynamic matrix inverse (presented in the full version) a similar conditional lower bound $\Omega(n^{1-\varepsilon})$ in the RAM-model for all constants $\varepsilon > 0$ can be obtained from the OMv conjecture [18].

One can also obtain a $\Omega(n^{2-\varepsilon})$ lower bound via OMv for dynamic matrix inverse with column updates and column queries, and (when reducing from OuMv) for an algorithm supporting both column and row updates and only element queries.

A. Column Update, Row Query

In this subsection we will present a new conditional lower bound for the dynamic matrix inverse with column updates and row queries, based on the dynamic product of two matrices. The new problem for the column update setting can be seen as an extension of the OMv conjecture. Instead of having online vectors, a set of possible vectors is given first and then one vector is selected from this list. We call this problem *v-hinted Mv* as it is similar to the OMv problem when provided a hint for the vectors.

Definition 5.1 (v-hinted Mv)**.** *Let the computations be performed over the boolean semi-ring and let $t = n^\tau$, $0 < \tau < 1$. The v-hinted Mv problem consists of the following phases:*

1) *Input an $n \times t$ matrix M*
2) *Input a $t \times n$ matrix V*
3) *For an input index $i \in [n]$ output $MV_{[n],i}$ (i.e. multiply M with the ith column of V).*

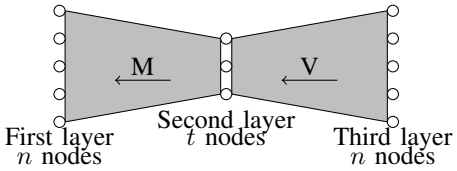

Figure 14. Graphical representation of the matrices M and V.

The definition of the v-hinted Mv problem is based on boolean matrix operations, so it can also be interpreted as a graph problem, i.e. the transitive closure problem displayed in Fig. 14. For this interpretation, the matrices M and V can be seen as a tripartite graph, where M lists the directed edges between the first layer of n nodes and the second layer of n^τ nodes. The matrix V specifies the edges between the second layer and the third layer of n nodes. All edges are oriented in the direction: first layer $\leftarrow$ second layer $\leftarrow$ third layer. The last phase of the v-hinted Mv problem consists of queries, where we have to answer which nodes of the first layer can be reached by some node i in the third layer, i.e. we perform a *source query*.

To motivate a lower bound, let us show two simple algorithms for solving the v-hinted Mv problem:

- Precompute the product MV in phase 2 using $O(n^{\omega(1,1,\tau)})$ operations, and output the ith column of the product in phase 3.
- Do not compute anything in phase 2 and compute $MV_{i,[n]}$ in phase 3 using a matrix-vector product in $O(n^{1+\tau})$ operations.

Currently no polynomially better way than these two options are known.[18] We ask if there is another third option with a substantially different complexity and formalize this via the following conjecture: We conjecture that the trivial algorithm is essentially optimal, i.e. we cannot do better than to decide between precomputing everything in phase 2 or to compute a matrix-vector product in phase 3. The conjecture can be seen as formalizing the trade-off between *pre-computing everything via fast matrix multiplication* vs *computing only required information online via vector-matrix product*.

Conjecture 5.2 (v-hinted Mv conjecture)**.** *Any algorithm solving v-hinted Mv with high probability, while requiring polynomial pre-processing time in phase 1, requires $\Omega(n^{\omega(1,1,\tau)-\varepsilon})$ operations for phases 2 or $\Omega(n^{1+\tau-\varepsilon})$ operations for phase 3 for all constant $\varepsilon > 0$.*

Theorem 5.3. *Assuming the v-hinted Mv Conjecture 5.2, the dynamic matrix-product with row updates and column queries requires $\Omega(n^{\omega(1,1,\tau)-\tau-\varepsilon})$ update time (worst-case), if the query time (worst-case) is $O(n^{1+\tau-\varepsilon})$ for some constant $\varepsilon > 0$.*

The same lower bound holds for any column update, row query algorithm, as we can just maintain the transposed product.

For current ω when balancing update and query time, this implies lower bound of $\Omega(n^{1.528})$.

The proof of Theorem 5.3 can be found in the full version.

Note that the lower bound from Theorem 5.3 allows for a trade-off between query and update time. The bound is tight with our upper bound from Theorem 4.1, if the query time is not larger than the update time. We can also give a more direct lower bound for the dynamic matrix inverse, that captures the algebraic nature of the v-hinted Mv problem.

By expressing the boolean matrix products as a graph as in Fig. 14, we obtain the following lower bound for transitive closure.

Corollary 5.4. *Assuming the v-hinted Mv Conjecture 5.2, the dynamic transitive closure problem (and DAG-path counting and k-path for $k \geq 3$) with polynomial pre-processing time and node updates (restricted to updating only incoming edges) and query operations for obtaining the reachability of any source node, requires $\Omega(n^{\omega(1,1,\tau)-\tau-\varepsilon})$ update time (worst-case), if the query time (worst-case) is bounded by $O(n^{1+\tau-\varepsilon})$ for some constant $\varepsilon > 0$.*

For current ω when balancing update and query time, this implies a lower bound of $\Omega(n^{1.528})$.

The proof of Corollary 5.4 can be found in the full version.

Theorem 5.3 implies the same lower bound for column update/row query dynamic matrix inverse and adjoint via reductions from the full version.

Corollary 5.5. *Assuming the v-hinted Mv Conjecture 5.2, the dynamic matrix inverse (and dynamic adjoint) with column updates and row queries requires $\Omega(n^{\omega(1,1,\tau)-\tau-\varepsilon})$ update time (worst-case), if the query time (worst-case) is $O(n^{1+\tau-\varepsilon})$ for some constant $\varepsilon > 0$.*

For current ω when balancing update and query time, this implies lower bound of $\Omega(n^{1.528})$.

B. Element Update, Row Query

Next, we want to define a problem which is very similar to the v-hinted Mv problem, but allows for a lower bound for the weaker setting of element updates and row query dynamic matrix inverse.

First, remember the high-level idea of the v-hinted Mv problem: We are given a matrix M and a set of possible vectors (i.e. a matrix) V and have to output only one matrix vector product Mv for some v in V, but since we don't know which vector is going to be chosen our only choices are pre-computing everything or waiting for the choice of v. When

[18] One can, however, improve the time requirement of phase 3 by a factor of $\log n$ using the technique from [41], but no $O(n^{1+\tau-\varepsilon})$ algorithm is known for some constant $\varepsilon > 0$. For further discussion what previous results for the Mv- and OMv-problem imply for our conjectures/problems, we refer to the full version.

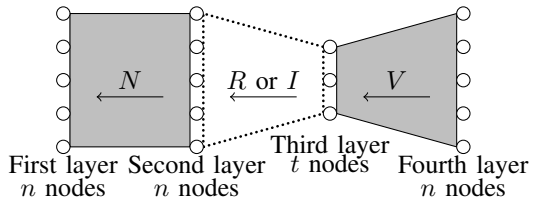

Figure 15. Graphical representation of the Mv-hinted Mv problem (Definition 5.6).

trying to extend this problem to element updates, then we obviously can not insert the matrix V via element updates one by one, as that would cause a too high overhead in the reduction and thus a very low lower bound. So instead we will give V already during the pre-processing, but the matrix M is not fully known. Instead, the matrix M is created from building blocks, which are selected by the element updates. Formally the problem is defined as follows:

Definition 5.6 (Mv-hinted Mv)**.** *Let all operations be performed over the boolean semi-ring and let $t = n^\tau$ for $0 < \tau < 1$. The* Mv-hinted Mv *problem consists of the following phases:*

1) *Input matrices $N \in R^{n\times n}, V \in R^{t\times n}$*
2) *Input $I \in [n]^t$.*
3) *Input index $j \in [n]$ and output $N_{[n],I}V_{[t],j}$.*

This problem has three different interpretations. One is to consider this a variant of the v-hinted Mv problem, but with two hints: one for M and one for the vector v (hence the name Mv-hinted Mv). First in phase 1, we are given a matrix N and a matrix V (i.e. a set of vectors) as a hint for M and v. In phase 2 the hint for M concretized by constructing M from columns of N.

Another interpretation for this problem is as some *dynamic 3-matrix product* NRV, where R is an initially all-zero rectangular $n\times t$ matrix. Here the phase 2 can be seen as updates to the R matrix, where for $I = (i_1, ..., i_t)$ the entries $R_{i_j,j}$ are set to 1 for $j = 1, ..., t$, and all other entries are left unchanged.

The third interpretation for the problem is graph theoretic and considers the problem to be a dynamic transitive closure problem with edge updates and source query. This graphical representation is displayed in Fig. 15. We are given 4 groups of nodes, the first, second and fourth group are of size n while the third group is only of size t. There exist directed edges from the second to the first group given by the non-zero entries of N and the edges from the fourth to the third group are given by V. In phase 2, t edges are inserted from the third to the second layer. In phase 3 we have to answer which nodes in the first layer can be reached from source-node j in the fourth group.

The *Mv-hinted Mv* problem can be solved by the following trivial algorithm: Assuming polynomial pre-processing time, we do not know how to exploit the given information N and V. We have no idea which entries of N will be multiplied with which entries of V and we can not try all exponentially many possible combinations for the vector I, so we do not know how to compute anything useful. For the next phases we have the following options:

- In phase 2, compute the product $N_{[n],I}V$ using $O(n^{\omega(1,\tau,1)})$ operations. In phase 3 we simply output the jth column of that product.
- We do not compute anything in phase 2, but remember the set I. In phase 3 we compute $N_{[n],I}V_{[t],j}$ as a vector-matrix-vector product in $O(n^{1+\tau})$ time.

Again we ask, if there is a better third option than trade-off between *pre-computing everything* vs *waiting and computing only required information*. We formalize this question as the following conjecture: The two options are essentially optimal, meaning there is no better way than to pre-compute everything in phase 2 or to wait and perform a matrix vector product in phase 3.

Conjecture 5.7. *Any algorithm solving* Mv-hinted Mv *with high probability, while requiring polynomial pre-processing time in Phase 1, satisfies one of the following:*

- *Phase 2 requires $\Omega(n^{\omega(1,\tau,1)-\varepsilon})$.*
- *Phase 3 requires $\Omega(n^{1+\tau-\varepsilon})$.*

For every $\varepsilon > 0$.

Since the Mv-hinted Mv problem can be represented as a product of three matrices NRV, we obtain the following lower bound for dynamic 3-matrix product algorithms with element updates and row queries.

Theorem 5.8. *Assuming the Mv-hinted Mv Conjecture 5.7, any dynamic matrix product algorithm with polynomial pre-processing time, element updates and column queries requires $\Omega(n^{\omega(1,1,\tau)-\tau-\varepsilon})$ worst-case update time, if the worst-case query time is $O(n^{1+\tau-\varepsilon})$ for some constant $\varepsilon > 0$.*

The same lower bound holds for any element update, row query algorithm, as we can just maintain the transposed product.

For current ω when balancing update and query time, this implies lower bound of $\Omega(n^{1.529})$.

The proof of Theorem 5.8 can be found in the full version. Note that Theorem 5.8 also implies a lower bound on element update and element query, as we could query n elements to get an entire column of the product. The same lower bounds hold for dynamic matrix inverse and adjoint via the reduction in the full version. Hence we get a lower bound of $\Omega(n^{\omega(1,\tau,1)-\tau-\varepsilon})$ per element update or $\Omega(n^{\tau-\varepsilon})$ per element query for every $\varepsilon > 0$, which is tight via Sankowski's result presented in [2, Theorem 3], when n queries are not slower than one update.

Corollary 5.9. *Assuming the Mv-hinted Mv Conjecture 5.7, any dynamic matrix inverse (or dynamic ajoint) algorithm with element updates and row queries requires $\Omega(n^{\omega(1,1,\tau)-\tau-\varepsilon})$ update time (worst-case), if the query time (worst-case) is $O(n^{1+\tau-\varepsilon})$ for some constant $\varepsilon > 0$.*

For current ω when balancing update and query time, this implies lower bound of $\Omega(n^{1.528})$.

Additionally, any dynamic matrix inverse (or dynamic adjoint) algorithm with element updates and element queries requires $\Omega(n^{\omega(1,1,\tau)-\tau-\varepsilon})$ update time (worst-case), if the query time (worst-case) is $O(n^{\tau-\varepsilon})$ for some constant $\varepsilon > 0$.

The graph theoretic representation of the problem, yield the same lower bound for the dynamic transitive closure and DAG path counting problem with edge updates and source queries (and thus also edge updates and n pair queries).

Corollary 5.10. *Assuming the Mv-hinted Mv Conjecture 5.12, dynamic transitive-closure and DAG path counting with polynomial pre-processing time, edge updates and source queries, requires $\Omega(n^{\omega(1,1,\tau)-\tau-\varepsilon})$ worst-case update time, if the worst-case query time is $O(n^{1+\tau-\varepsilon})$ for some $\varepsilon > 0$.*

For current ω, this implies a lower bound of $\Omega(n^{1.528})$.

Additionally, any dynamic transitive-closure or DAG path counting algorithm with edge updates and pair queries requires $\Omega(n^{\omega(1,1,\tau)-\tau-\varepsilon})$ update time (worst-case), if the query time (worst-case) is $O(n^{\tau-\varepsilon})$ for some constant $\varepsilon > 0$.

Note that these lower bounds specify a trade-off between update and query time, i.e. we can query faster, if we are willing to pay a higher update time. This trade-off is tight unless the query time for querying a row exceeds the update time.

C. Element Update, Element Query

The Mv-hinted Mv problem allowed us to specify a lower bound for dynamic matrix inverse algorithms with slow update and fast query time (i.e. [2, Theorem 3]). We also want to obtain a lower bound for the case that update and query time are balanced. The Mv-hinted Mv problem does not properly capture the hardness of single element queries, because the problem asks for an entire column to be queried. When querying a column via $O(n)$ element queries, we would have some kind of look-ahead information, because after the first element query, the next $O(n)$ positions for the queries are known, since they are in the same column. The next problem we define is a variation of the Mv-hinted Mv problem, where we try to fix this issue by querying only a single value. The new problem can be considered a hinted variant of the OuMv problem [18], where we repeat the idea of restricting some matrix N to obtain a matrix M and giving hints for the vectors u and v.

Definition 5.11 (uMv-hinted uMv)**.** *Let all operations be performed over the boolean semi-ring and let $t_1 = n^{\tau_1}, t_2 = n^{\tau_2}$, $0 < \tau_1, \tau_2 < 1$. The uMv-hinted uMv problem consists of the following phases:*

1) *Input matrices $U \in R^{n \times t_1}, N \in R^{n \times n}, V \in R^{t_2 \times n}$*
2) *Input $I \in [n]^{t_1}$.*
3) *Input $J \in [n]^{t_2}$.*
4) *Input indices $i, j \in [n]$ and output $(UN_{I,J}V)_{i,j}$.*

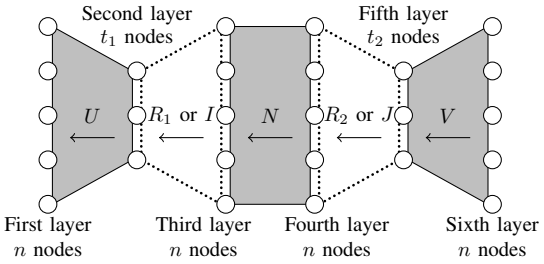

Figure 16. Graphical representation of the uMv-hinted uMv problem (Definition 5.11).

This problem can be considered a dynamic 5-matrix product UR_1NR_2V, where R_1 and R_2 are initially all-zero rectangular matrices. Phase 2 and 3 can be seen as updates to the R_1 and R_2 matrices, where some entries are set to 1. A more intuitive illustration for this problem is to see the problem as a vector-matrix-vector product $u^\top Mv$, where during the pre-processing we are given a hint what the vectors u, v and the matrix M could be. This interpretation of the problem is also the source for its name uMv-hinted uMv. The two phases 2 and 3 concretize the hint for M by constructing M from rows and columns of N via $M = N_{I,J}$. (Note that I and J are vectors, not sets, so $N_{I,J}$ is not a typical submatrix, see submatrix notation in the preliminaries Section III. Instead, rows and columns can be repeated and re-ordered.) During the last phase one row u of U and one column v of V are selected and the product $u^\top Mv$ has to be computed.

Similar to the Mv-hinted Mv problem (Definition 5.1), we can specify the uMv-hinted uMv problem as a transitive closure problem. This graphical representation is displayed in Fig. 16. We are given 6 groups of nodes, the first group is of size n and the second group is of size t_1. There exist directed edges from the second to the first group, specified by U. The third and fourth group are of size n and have directed edges from the fourth to third group, specified by N. The fifth group is of size t_2, while the 6th group is of size n. There also exists directed edges from the sixth to the fifth group, specified by V. In phase 2 each node in the second group gets a directed edge from a node in the third group. In phase 3 each node in the fifth group gets a directed edge to a node in the fourth group. In phase 4 we have to answer whether the jth node in the last group can reach the ith node in the first group.

The uMv-hinted uMv problem can be solved by the following trivial algorithm. Depending on the values for t_1

and t_2 we have the following three options:

- Compute the product $UN_{I,[n]}$ in phase 2, using $O(n^{\omega(1,\tau_1,1)})$ operations. In phase 4 we compute the product of the ith row of $UN_{I,[n]}$ and the jth column of V.
- Compute the product $N_{I,J}V$ in phase 3, using $O(n^{\omega(\tau_1,\tau_2,1)})$ operations. In phase 4 we compute the product of the ith row of U and the jth column of $M_{I,J}V$.
- Compute the product $(UN_{I,J}V)_{i,j}$ as a vector-matrix-vector product in $O(n^{\tau_1\tau_2})$

(Note that computing $UN_{I,J}$ needs as much time as computing $N_{I,J}V$, so this would be the same as the second variant.) Again we ask, if there is a better option than pre-computing everything via fast-matrix multiplication or to wait which information is going to be required, or maybe there exists some clever pre-processing even though we do not know which entries of N will be multiplied with which entries of V or U. The question is formalized via the conjecture that the three options of the trivial algorithm are essentially optimal. So similar to the v-hinted Mv and Mv-hinted Mv conjecture, the uMv-hinted uMv conjecture can be seen as a trade-off between pre-computing everything vs waiting for the next phase and computing only required information, which forms the fundamental barrier for all currently known techniques for the dynamic matrix inverse algorithms.

Conjecture 5.12. *Any algorithm solving uMv-hinted uMv with high probability, while requiring polynomial pre-processing time in Phase 1, satisfies one of the following:*

- *Phase 2 requires* $\Omega(n^{\omega(1,\tau_1,1)-\varepsilon})$.
- *Phase 3 requires* $\Omega(n^{\omega(\tau_2,\tau_1,1)-\varepsilon})$.
- *Phase 4 requires* $\Omega(n^{\tau_1+\tau_2-\varepsilon})$.

For every $\varepsilon > 0$.

Since the uMv-hinted uMv problem can be represented as a 5-matrix product, we obtain the following lower bound for dynamic 5-matrix product algorithms with element updates and element queries.

Theorem 5.13. *Assuming the uMv-hinted uMv Conjecture 5.12, the dynamic 5-matrix-product with polynomial time pre-processing, element updates and element queries requires* $\Omega(\min_{\tau_1,\tau_2}(n^{\tau_1+\tau_2} + n^{\omega(1,\tau_1,\tau_2)-\tau_2} + n^{\omega(1,1,\tau_1)-\tau_1})n^{-\varepsilon})$ *worst-case time for all* $\varepsilon > 0$ *for updates or queries.*

For current ω*, this implies a lower bound of* $\Omega(n^{1.407})$.

The proof of Theorem 5.13 can be found in the full version. From the graph theoretic representation of the uMv-hinted uMv problem, we obtain the same lower bound for the dynamic transitive closure problem (and dynamic DAG path counting as well as dynamic k-path for $k \geq 5$) with edge updates and pair queries. The same lower bound can also be obtained for cycle detection (and k-cycle detection for $k \geq 6$) by adding an edge from the first to the last layer during the query phase. The reduction from transitive closure to strong connectivity is done as in [19, Lemma 6.4].

Corollary 5.14. *Assuming the uMv-hinted uMv Conjecture 5.12, dynamic transitive-closure (and DAG path counting, strong connectivity, k-path, cycle detection and k-cycle detection) with polynomial pre-processing time, element updates and element queries requires* $\Omega(\min_{\tau_1,\tau_2}(n^{\tau_1+\tau_2} + n^{\omega(1,\tau_1,\tau_2)-\tau_2} + n^{\omega(1,1,\tau_1)-\tau_1})n^{-\varepsilon})$ *worst-case time for all* $\varepsilon > 0$ *for updates or queries.*

For current ω*, this implies a lower bound of* $\Omega(n^{1.407})$.

Since dynamic matrix inverse (and adjoint) can be used to maintain a 5-matrix product (see the full version), we obtain the same lower bound for dynamic matrix inverse with element updates and queries. The lower bound extends even to determinant and rank. For determinant this is because element update/query determinant is equivalent to element update/query inverse (see the full version). For rank the reduction is a bit longer using graph problems:

For element queries, transitive closure can be solved via st-reachability. For the reduction we only have to prove that even though s and t are fixed, the reachibility between any pair (u, v) can be queried. For this we simply add edges (s, u) and (v, t) and check if s can reach t. Afterward we remove these two edges again. Thanks to [19], we know bipartite perfect matching can solve st-reachability, which in turn can be solved by rank via a reduction from the full version. Thus we obtain the following corollaries:

Corollary 5.15. *Assuming the uMv-hinted uMv Conjecture 5.12, any dynamic matrix inverse (and dynamic adjoint, determinant, rank) algorithm with polynomial time pre-processing, element updates and element queries requires* $\Omega((n^{\tau_1+\tau_2} + n^{\omega(1,\tau_1,\tau_2)-\tau_2} + n^{\omega(1,1,\tau_1)-\tau_1})n^{-\varepsilon})$ *worst-case time for all* $\varepsilon > 0$ *for updates or queries.*

For current ω*, this implies a lower bound of* $\Omega(n^{1.407})$.

Corollary 5.16. *Assuming the uMv-hinted uMv Conjecture 5.12, dynamic bipartite perfect matching with polynomial pre-processing time, element updates requires* $\Omega((n^{\tau_1+\tau_2} + n^{\omega(1,\tau_1,\tau_2)-\tau_2} + n^{\omega(1,1,\tau_1)-\tau_1})n^{-\varepsilon})$ *worst-case time for all* $\varepsilon > 0$.

For current ω*, this implies a lower bound of* $\Omega(n^{1.407})$.

D. Discussion on Super-Linear Bounds for Dynamic Matrix Inverse

The high-level idea of our lower bounds can be summarized as *precomputing everything* vs *waiting what information is going to be required*, i.e. if we do not know which information is going to be required in the next phase, then we can either do nothing and compute a vector-matrix product or we can precompute all possibilities using fast-matrix multiplication. Both of these option are a bit slow, one one hand using many vector-matrix products is slower than using

fast matrix-multiplication, on the other hand pre-computing everything will compute never needed information. The trade-off between these two options forms the barrier for all currently known techniques for the dynamic matrix inverse. Our conjectures ask, if there is some better third option available.

This nature of *precomputing everything* vs *waiting* can also be seen in Pătraşcu's multiphase problem [20]. Unfortunately the multiphase problem, like other popular problems for lower bounds such as OMv, triangle detection, orthogonal vectors, SETH or 3-orthogonal vectors, are all unable to give super-linear lower bounds for the dynamic matrix inverse.

Online matrix-vector [18]: The OMv conjecture states, that given a boolean matrix M and polynomial time pre-processing of that matrix, computing n products Mv_i, $i = 1, ..., n$, requires $\Omega(n^{3-\varepsilon})$ time, if the algorithm has to output Mv_i before receiving the next vector v_{i+1}.

We now explain why this conjecture can not give super-linear bounds for dynamic matrix inverse.

- *Element updates:* In worst-case the n vectors $(v_i)_{1 \leq i \leq n}$ contains $\Theta(n^2)$ bits of information. An element update contains only $O(\text{polylog}(n))$ bits of information, unless we use some really large field, which would result in slow field operations. Thus for all n vectors, we have to perform $\Omega(n^{2-\varepsilon})$ updates in total for every constant $\varepsilon > 0$, which means no super-linear lower bound for element updates is possible.
- *Column updates:* For the setting column update and column query, the OMv conjecture is able to give a $\Omega(n^{2-\varepsilon})$ lower bound for the dynamic matrix inverse and the related OuMv conjecture can give the same lower bound for column and row update, element query dynamic inverse (both lower bounds are a result via reductions from the full version). However, for column update/row query we again have the problem that we would have to perform n updates (or one update and n queries), yielding no super-linear lower bound, when using this reduction from the full version.

Multiphase problem [20]: In the multiphase problem we have three phases: First, we are given a $k \times n$ matrix M, then a vector v and lastly we have to answer whether some $(Mv)_i$ is 0 or 1. From all other presented problems, this problem captures best the issue of *online computation* vs *precomputation*, however, the conjectured time ($\Omega(kn)$ when given v or $\Omega(k)$ when given i) is not large enough for a reduction, because we have to perform $O(n^{1-\varepsilon})$ element updates for every constant $\varepsilon > 0$, just to insert the information of v, so we can not get super-linear lower bounds.

Static problems conjectured to require $\Omega(n^\omega)$ time (e.g. triangle detection): An intuitive approach to obtain lower bounds is to reduce some static problem[19] to a dynamic one. These type of lower bounds have the issue, that they require a pre-processing time that is lower than the time required to solve the problem in the static way. For example one could get a super-linear $n^{\omega-1}$ (amortized) lower bound for dynamic matrix inverse via st-reachability by reducing from triangle detection as in [19]. However, this lower bound only holds, if one assumes $o(n^\omega)$ pre-processing time.

Assuming $o(n^\omega)$ pre-processing has two problems:

- It does not rule out algorithms with fast update time that have $\Omega(n^\omega)$ pre-processing. Our dynamic algorithms and the ones from [2], [3] are of this type. We are interested in understanding why these algorithm can not achieve linear update time, even though their pre-processing is larger than $\Omega(n^\omega)$.
- For some problems the $o(n^\omega)$ pre-processing requirement will refute any non-trivial algorithms (see the full version for a proof). For example any dynamic matrix determinant algorithm with $o(n^\omega)$ pre-processing must have $\Omega(n^\omega)$ update time. This is why algorithm with larger pre-processing are interesting.

Static problem conjectured to require $\Omega(n^{\omega+\varepsilon})$: In the previous paragraph we highlighted the problems of using static problems that are conjectured to take $\Omega(n^\omega)$ time. Here we want to discuss static problems that are conjectured to have a higher complexity.

- *APSP:* Computing all-pairs-shortest-paths with polynomialy bounded edge weights (i.e. n^c) is conjectured to require $\Omega(n^{3-\varepsilon})$ for every constant $\varepsilon, c > 0$. So far APSP seems unsuited for algebraic algorithms since these algorithms always incur a pseudo-polynomial dependency on the edge weights.
- *BMM:* The Boolean-Matrix-Multiplication conjecture forbids the use of fast matrix multiplication, so it can not be used to bound the complexity of algebraic algorithms.
- *k-clique:* It is conjectured that detecting a k-clique in a graph requires $\Omega(n^{k\omega/3})$. For $k = 3$ the k-clique problem is triangle detection, which was covered in the previous paragraph. For $k > 3$ there is no known reduction to matrix inverse without increasing the dimension to $n^{k/3}$ in which case we have the same problem as in the previous paragraph.
- *k-orthogonal:* In the k-orthogonal vectors problem we are given k sets $S^1, ..., S^k$, each containing n vectors of dimension $d = n^{o(1)}$. The task is to find a k-tuple $(i_1, ..., i_k)$ such that $\sum_{j=1}^{n} S^1_{i_1,j} \cdot ... \cdot S^k_{i_k,j} = 0$. This is conjectured to require $\Omega(n^{k-\varepsilon})$ time for every constant $\varepsilon > 0$. For $k > 2$ no reduction to dynamic matrix inverse is known, while for $k = 2$ we again have the

[19]Static problem refer to problems that do not have several phases. For example triangle detection or APSP are typical static problems used for dynamic lower bounds. [19]

same issue as with multiphase and OMv: We require to perform too many updates, just to insert the sets S^1, S^2.

VI. Open Problems

Amortization: All the results in this paper focus on worst-case update time. A major open problem is whether one can get faster update time via amortization or describe reasonable conjectures that hold for amortized update time as well. Curently there only exist amortized lower bounds for sparse graphs and for algorithms with small pre-processing time [19]. (We could extend our conjectures to imply lower bounds for amortized update time by repeating some phases, but we do not feel that they are reasonable enough. If interested, the full version describes how to amortize the lower bounds via repetition of some phases.) It will be already groundbreaking if amortization can improve the update time for some applications, such as st-reachability.

Refuting or supporting our conjectures: In this paper we need to propose new conjectures to capture the power of dynamic matrix multiplication. Since these conjectures are new, they need to be scrutinized. Breaking one of these conjectures would give a hope for improved algorithms for many problems considered in this paper. It might also be possible to support these conjectures with, e.g. via algebraic circuit lower bounds.

Distances: Many of the upper and lower bounds in Fig. 5 are tight. However, for the distance problems (st-distance or all-pair-distances) there are no matching upper and lower bounds. The best lower bound so far is obtained via transitive closure/reachability, but the upper bound is far above this lower bound. A major open problem is to close or at least narrow this gap.

Another open problem related to distances would be to extend our results to weighted graphs. The results can easily be extended to support integer weights in $[1, W]$ at the cost of an extra W factor. Thus these algebraic techniques are only suited for small integer weights. We wonder if it is possible to obtain an algorithm with $\log W$ dependency, e.g. via approximation as in [42].

Maintaining the object: Algebraic techniques tend to only return a quantitative answer: The size of the maximum matching, the distance or reachability between two nodes etc. Consider for instance our online bipartite matching algorithm. We trivially know which nodes on the right are part of the matching, as each newly added right node that increases the matching size must be part of the matching. Yet, we do not know which nodes on the left are matched or which edges are used. Is it possible to obtain the maintained object such as the matching or the path?

Sparse graphs: So far dynamic matrix inverse is the only technique that returns a non-trivial upper bound for st-reachability. However, for sparse graph even this upper bound is slower than just trivially running breath/depth first search in $O(m)$ time. We wonder if it is possible to obtain a $O(m^{1-\varepsilon})$ upper bound or a $\Omega(m)$ conditional lower bound. Currently the best lower bound for sparse graphs is $\Omega(m^{0.814})$, if one assumes $O(m^{1.407})$ pre-processing time [19].

Derandomization: While the dynamic matrix inverse for non-singular matrices is deterministic, we require randomization to extend the result to the setting where the matrix is allowed to temporarily become singular. Likewise most graph application such as reachability are randomized. Is it possible to derandomize some of these applications or can we make them at least las-vegas instead of monte-carlo?

Acknowledgment

The authors would like to thank Amir Abboud for comments and pointing out an open problem in [19]. We thank Adam Karczmarz for pointing out the strong-connectivity application.

This project has received funding from the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme under grant agreement No 715672. Danupon Nanongkai and Thatchaphol Saranurak were also partially supported by the Swedish Research Council (Reg. No. 2015-04659).

References

[1] J. van den Brand, D. Nanongkai, and T. Saranurak, "Dynamic matrix inverse: Improved algorithms and matching conditional lower bounds," *CoRR*, vol. abs/1905.05067, 2019.

[2] P. Sankowski, "Dynamic transitive closure via dynamic matrix inverse (extended abstract)," in *FOCS*. IEEE Computer Society, 2004, pp. 509–517.

[3] ——, "Faster dynamic matchings and vertex connectivity," in *SODA*. SIAM, 2007, pp. 118–126.

[4] Y. T. Lee and A. Sidford, "Efficient inverse maintenance and faster algorithms for linear programming," in *FOCS*. IEEE Computer Society, 2015, pp. 230–249.

[5] M. B. Cohen, Y. T. Lee, and Z. Song, "Solving linear programs in the current matrix multiplication time," *CoRR*, vol. abs/1810.07896, 2018.

[6] J. van den Brand, "A deterministic linear program solver in current matrix multiplication time," *unpublished*, 2019, to be announced at SODA'20.

[7] P. Sankowski, "Subquadratic algorithm for dynamic shortest distances," in *COCOON*, ser. Lecture Notes in Computer Science, vol. 3595. Springer, 2005, pp. 461–470.

[8] J. van den Brand and D. Nanongkai, "Dynamic approximate shortest paths and beyond: Subquadratic and worst-case update time," *CoRR*, vol. abs/1909.10850, 2019.

[9] J. van den Brand and T. Saranurak, "Sensitive distance and reachability oracles for large batch updates," *CoRR*, vol. abs/1907.07982, 2019.

[10] P. Sankowski and M. Mucha, "Fast dynamic transitive closure with lookahead," *Algorithmica*, vol. 56, no. 2, pp. 180–197, 2010.

[11] T. Kavitha, "Dynamic matrix rank with partial lookahead," *Theory Comput. Syst.*, vol. 55, no. 1, pp. 229–249, 2014, announced at FSTTCS'08.

[12] S. Khanna, R. Motwani, and R. H. Wilson, "On certificates and lookahead in dynamic graph problems," *Algorithmica*, vol. 21, no. 4, pp. 377–394, 1998, announced at SODA'96.

[13] M. Yannakakis, "Graph-theoretic methods in database theory," in *PODS*. ACM Press, 1990, pp. 230–242.

[14] J. Sherman and W. J. Morrison, "Adjustment of an inverse matrix corresponding to a change in one element of a given matrix," *The Annals of Mathematical Statistics*, vol. 21, no. 1, pp. 124–127, 1950.

[15] C. Demetrescu and G. F. Italiano, "Trade-offs for fully dynamic transitive closure on dags: breaking through the $o(n^2$ barrier," *J. ACM*, vol. 52, no. 2, pp. 147–156, 2005, announced at FOCS'00.

[16] F. L. Gall and F. Urrutia, "Improved rectangular matrix multiplication using powers of the coppersmith-winograd tensor," in *SODA*. SIAM, 2018, pp. 1029–1046.

[17] F. L. Gall, "Powers of tensors and fast matrix multiplication," in *ISSAC*. ACM, 2014, pp. 296–303.

[18] M. Henzinger, S. Krinninger, D. Nanongkai, and T. Saranurak, "Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture," in *STOC*. ACM, 2015, pp. 21–30.

[19] A. Abboud and V. V. Williams, "Popular conjectures imply strong lower bounds for dynamic problems," in *FOCS*. IEEE Computer Society, 2014, pp. 434–443.

[20] M. Patrascu, "Towards polynomial lower bounds for dynamic problems," in *STOC*. ACM, 2010, pp. 603–610.

[21] V. Strassen, "Gaussian elimination is not optimal," *Numerische Mathematik*, vol. 13, pp. 354–356, 1969.

[22] B. Bosek, D. Leniowski, P. Sankowski, and A. Zych, "Online bipartite matching in offline time," in *FOCS*. IEEE Computer Society, 2014, pp. 384–393.

[23] A. Bernstein, J. Holm, and E. Rotenberg, "Online bipartite matching with amortized replacements," in *SODA*. SIAM, 2018, pp. 947–959.

[24] L. Lovász, "On determinants, matchings, and random algorithms," in *FCT*, 1979, pp. 565–574.

[25] M. Mucha and P. Sankowski, "Maximum matchings via gaussian elimination," in *FOCS*. IEEE Computer Society, 2004, pp. 248–255.

[26] D. Nanongkai, T. Saranurak, and C. Wulff-Nilsen, "Dynamic minimum spanning forest with subpolynomial worst-case update time," in *FOCS*. IEEE Computer Society, 2017, pp. 950–961.

[27] D. Nanongkai and T. Saranurak, "Dynamic spanning forest with worst-case update time: adaptive, las vegas, and $o(n^{1/2 - \epsilon})$-time," in *STOC*. ACM, 2017, pp. 1122–1129.

[28] C. Wulff-Nilsen, "Fully-dynamic minimum spanning forest with improved worst-case update time," in *STOC*. ACM, 2017, pp. 1130–1143.

[29] M. Henzinger, S. Krinninger, and D. Nanongkai, "Decremental single-source shortest paths on undirected graphs in near-linear total update time," *J. ACM*, vol. 65, no. 6, pp. 36:1–36:40, 2018, announced at FOCS'14. [Online]. Available: https://dl.acm.org/citation.cfm?id=3218657

[30] ——, "Sublinear-time decremental algorithms for single-source reachability and shortest paths on directed graphs," in *STOC*. ACM, 2014, pp. 674–683.

[31] ——, "Sublinear-time maintenance of breadth-first spanning tree in partially dynamic networks," in *ICALP (2)*, ser. Lecture Notes in Computer Science, vol. 7966. Springer, 2013, pp. 607–619.

[32] D. Chakraborty, L. Kamma, and K. G. Larsen, "Tight cell probe bounds for succinct boolean matrix-vector multiplication," in *STOC*. ACM, 2018.

[33] K. G. Larsen and R. R. Williams, "Faster online matrix-vector multiplication," in *SODA*. SIAM, 2017, pp. 2182–2189.

[34] R. Clifford, A. Grønlund, and K. G. Larsen, "New unconditional hardness results for dynamic and online problems," in *FOCS*. IEEE Computer Society, 2015, pp. 1089–1107.

[35] P. Bürgisser, M. Clausen, and M. A. Shokrollahi, *Algebraic complexity theory*, ser. Grundlehren der mathematischen Wissenschaften. Springer, 1997, vol. 315.

[36] V. V. Williams, "Multiplying matrices faster than coppersmith-winograd," in *STOC*. ACM, 2012, pp. 887–898.

[37] K. Mulmuley, U. V. Vazirani, and V. V. Vazirani, "Matching is as easy as matrix inversion," *Combinatorica*, vol. 7, no. 1, pp. 105–113, 1987, announced at STOC'87.

[38] W. Baur and V. Strassen, "The complexity of partial derivatives," *Theor. Comput. Sci.*, vol. 22, pp. 317–330, 1983.

[39] J. Morgenstern, "How to compute fast a function and all its derivatives: A variation on the theorem of baur-strassen," *ACM SIGACT News*, vol. 16, no. 4, pp. 60–62, 1985.

[40] G. S. Frandsen, J. P. Hansen, and P. B. Miltersen, "Lower bounds for dynamic algebraic problems," *Inf. Comput.*, vol. 171, no. 2, pp. 333–349, 2001, announced at STACS'99.

[41] R. Williams, "Matrix-vector multiplication in sub-quadratic time: (some preprocessing required)," in *SODA*. SIAM, 2007, pp. 995–1001.

[42] U. Zwick, "All pairs shortest paths using bridging sets and rectangular matrix multiplication," *J. ACM*, vol. 49, no. 3, pp. 289–317, 2002.