On Dynamic Graph Algorithms with Predictions[*]

Jan van den Brand[†]   Sebastian Forster[‡]   Yasamin Nazari[§]   Adam Polak[¶]

Abstract

Dynamic algorithms operate on inputs undergoing updates, e.g., insertions or deletions of edges or vertices. After processing each update, the algorithm has to answer queries regarding the current state of the input data. We study dynamic algorithms in the model of algorithms with predictions (also known as learning-augmented algorithms). We assume the algorithm is given imperfect predictions regarding future updates, and we ask how such predictions can be used to improve the running time. In other words, we study the complexity of dynamic problems parameterized by the prediction accuracy. This can be seen as a model interpolating between classic online dynamic algorithms – which know nothing about future updates – and offline dynamic algorithms with the whole update sequence known upfront, which is similar to having perfect predictions. Our results give smooth tradeoffs between these two extreme settings.

Our first group of results is about partially dynamic problems with edge updates. We give algorithms for incremental and decremental transitive closure and approximate APSP that take as an additional input a predicted sequence of updates (edge insertions, or edge deletions, respectively). They preprocess it in $\tilde{O}(n^{(3+\omega)/2})$ time, and then handle updates in $\tilde{O}(1)$ worst-case time and queries in $\tilde{O}(\eta^2)$ worst-case time. Here η is an error measure that can be bounded by the maximum difference between the predicted and actual insertion (deletion) time of an edge, i.e., by the ℓ_∞-error of the predictions.

The second group of results concerns fully dynamic problems with vertex updates, where the algorithm has access to a predicted sequence of the next n updates. We show how to solve fully dynamic triangle detection, maximum matching, single-source reachability, and more, in $O(n^{\omega-1} + n\eta_i)$ worst-case update time. Here η_i denotes how much earlier the i-th update occurs than predicted.

Our last result is a reduction that transforms a worst-case incremental algorithm without predictions into a fully dynamic algorithm which is given a predicted deletion time for each element at the time of its insertion. As a consequence we can, e.g., maintain fully dynamic exact APSP with such predictions in $\tilde{O}(n^2)$ worst-case vertex insertion time and $\tilde{O}(n^2(1+\eta_i))$ worst-case vertex deletion time (for the prediction error η_i defined as above).

Our algorithms from the first two groups, given sufficiently accurate predictions, achieve running times that go below known lower bounds for classic (without predictions) dynamic algorithms under the OMv Hypothesis. Moreover, our dependence on the prediction errors (so-called smoothness) is conditionally optimal, under plausible fine-grained complexity assumptions, at least in certain parameter regimes.

1 Introduction.

Dynamic algorithms maintain a solution to a computational problem – e.g., single source distances in a graph – for an input that undergoes a sequence of updates – e.g., edge insertions or deletions. The goal is to process such updates as efficiently as possible, at least faster than recomputing the solution from scratch. This is however not always plausible, as evidenced by numerous fine-grained conditional lower bounds, see, e.g., [53, 2, 31].

The recent line of research on learning-augmented algorithms provides many examples of how performance of classic algorithms can be provably improved using imperfect *predictions*, generated, e.g., by machine-learning models (see surveys by Mitzenmacher and Vassilvitskii [49, 50]). Among others, predictions allow us to improve competitive ratios of online algorithms (e.g., [47, 57]), running times of static algorithms (e.g., [22, 17]), approximation ratios of polynomial-time approximation algorithms (e.g., [26, 28, 52]). Often these improvements go beyond what is

---

provably possible for classic algorithms (e.g., [47, 26] and many others). In this work we ask the following natural question:

How could we use predictions to improve dynamic algorithms?

We make two choices to narrow down this question. First, we focus on *predictions about future input data.* This is akin to many previous results on learning-augmented online algorithms (e.g., [47, 8]), and in contrast with settings where the predictions are about the output (e.g., [4, 22, 26]). Second, we focus on *improving the running time,* which is the most studied performance measure for dynamic algorithms.

Online and Offline Dynamic Problems, and Conditional Lower Bounds. Dynamic problems are most often studied in their online variants, i.e., future updates are not known to the algorithm, and it has to perform them one by one. On the other hand, offline dynamic algorithms (see, e.g, [25, 62, 37, 14, 55, 18]) are given a sequence of updates upfront. Note that the offline model is equivalent to our proposed model with predictions in the case that predictions are perfectly accurate. That is, we study an interpolation between offline and online dynamic algorithms, and ask how an algorithm's performance degrades with increasing inaccuracies of predictions. This interpolation question makes sense only for dynamic problems whose offline variants admit faster algorithms than their corresponding online variants do. Ideally, we would like to be able to say that predictions – even imperfect ones – allow certain dynamic problems to be solved faster than what is provably plausible without such predictions in the classic online model. In other words, we aim for running times going below known conditional lower bounds.

There are many reductions showing hardness of dynamic problems under popular fine-grained complexity assumptions about static problems, such as 3SUM, APSP, or CNF-SAT (see, e.g., [2]). However, all these reductions share what is from our perspective a limitation: they are not adaptive, they are capable of producing the whole input sequence at once. Hence, they already imply hardness for offline variants of dynamic problems, and thus they cannot provide tight conditional lower bounds for those dynamic problems whose offline variants happen to be strictly easier than corresponding online variants. It is an interesting open problem to find a reduction – from a natural static problem to a natural online dynamic problem – that does not have this limitation [1].

To our best knowledge, the only known tool in fine-grained complexity that is capable of distinguishing between online and offline variants of dynamic problems is the Online Matrix-Vector Multiplication (OMv) Hypothesis [31], a conditional assumption about an online problem itself. Therefore, in this work we focus (mostly) on problems with known tight OMv lower bounds. These lower bounds often show that recomputing from scratch is basically the best we can hope for (without predictions).

Warm-up: OMv with Predictions. Before we delve into graph problems, let us start with a simple sanity check and verify that the cubic time barrier for OMv can be broken using predictions. If it was not the case, our project would be hopeless because then problems with known OMv lower bounds would remain hard even with predictions.

Recall that in the OMv problem we are first given Boolean matrix $M \in \{0, 1\}^{n \times n}$, and then we need to answer n queries of the form: Given vector $v_i \in \{0, 1\}^n$, what is the Boolean product Mv? We have to answer queries one by one, in an online fashion: we have to output Mv_i before we get to learn what v_{i+1} is. OMv is conjectured to require cubic time, up to subpolynomial factors [31]. We propose the following variant of OMv with predictions:

Online matrix-vector multiplication (OMv) with predictions

Input offline:	matrix $M \in \{0, 1\}^{n \times n}$;
	predicted vectors $\hat{v}_1, \hat{v}_2, \ldots, \hat{v}_n \in \{0, 1\}^n$.
Input online:	vectors $v_1, v_2, \ldots, v_n \in \{0, 1\}^n$.
Output:	matrix-vector products $Mv_1, Mv_2, \ldots, Mv_n \in \{0, 1\}^n$ over Boolean semiring.

We show a simple algorithm whose running time depends on the ℓ_1-error of the predictions, which is one of the standard error measures for learning-augmented algorithms (see, e.g., [47, 4]).

OBSERVATION 1.1. *OMv with predictions can be solved in total time $O(n^\omega + n \sum_i ||v_i - \hat{v}_i||_1)$. More precisely, preprocessing runs in $O(n^\omega)$ time, and i-th query requires $O(n||v_i - \hat{v}_i||_1)$ time.*

Proof. In the preprocessing phase, compute $M \cdot [\hat{v}_1, \hat{v}_2, \ldots, \hat{v}_n]$ over integers, in time $O(n^\omega)$. Then, after receiving each vector v_i, compute (over integers) $Mv_i = M\hat{v}_i + M(v_i - \hat{v}_i)$. The first term, $M\hat{v}_i$ can be retrieved in $O(n)$

time because it is the i-th column of the matrix computed during the preprocessing. The second term, $M(v_i - \hat{v}_i)$ can be computed in time $O(n||v_i - \hat{v}_i||_1)$, because $||v_i - \hat{v}_i||_1$ equals the number of non-zeros of $v_i - \hat{v}_i$. Indeed, $||v_i - \hat{v}_i||_1 = ||\hat{v}_i - v_i||_0$, because v_i and $\hat{v}_i$ contain only zeros and ones. □

Note that $||v_i - \hat{v}_i||_1 \leqslant n$, and hence $\sum_i ||v_i - \hat{v}_i||_1 \leqslant n^2$, so even with arbitrarily bad predictions our algorithm never needs more than cubic time, i.e., it is *robust* in the learning-augmented terminology. On the other hand, for perfect predictions (i.e., $\sum_i ||v_i - \hat{v}_i||_1 = 0$) we achieve the running time of the best offline algorithm, i.e., we are *consistent*. In Section 1.2 we discuss these concepts further.

Moreover, the running time of this algorithm matches two conditional lower bounds. First, the dependence on the total prediction error $\eta = \sum_i ||v_i - \hat{v}_i||_1$ cannot be improved from $n\eta$ to $n\eta^{0.99}$ under the OMv hypothesis; this is because OMv reduces to OMv with predictions, with $\eta = O(n^2)$, by providing arbitrary (e.g., all-zero) vectors as predictions. Second, the n^ω term cannot be improved under the assumption that Boolean matrix multiplication is not easier than the general matrix multiplication, because even with perfect predictions (i.e., $\eta = 0$), solving OMv with predictions entails computing a Boolean matrix product. These arguments, however, do not rule out the possibility of, e.g., an $O(n^\omega + \eta^{3/2})$ time algorithm, which would be a meaningful improvement over the above algorithm. We find it an interesting open problem to provide a fine-grained conditional lower bound, matching the running time of Observation 1.1, that holds already restricted to instances with $\eta = \Theta(n^\alpha)$ for arbitrary fixed $\alpha \in (0, 2)$.

Finally, note that going below the cubic time barrier for OMv, even with accurate predictions, requires fast matrix multiplication. Hence, it is not a surprise that our graph algorithms discussed below – that go beyond known OMv-based lower bounds – use algebraic algorithms for matrix multiplication.

1.1 Our Results. Our main results can be categorized into three groups that differ in the considered settings, types of predictions and measures of prediction errors. We discuss the groups one by one. Table 1 provides a summary.

Partially Dynamic Graph Problems. Our first group of results is about partially dynamic problems with edge updates. These are problems on graphs undergoing edge insertions (incremental variant), and graphs undergoing edge deletions (decremental variant), but not both types of updates at the same time (this would be a fully dynamic variant, which we address later). While in general it is not the case that incremental and decremental variants must be equivalent, it turns out that all our results give the same bounds for both variants.

Our partially dynamic (incremental and decremental) algorithms take as an additional input a predicted sequence of updates (edge insertions or edge deletions, respectively). To illustrate our setting, we provide as an example a detailed definition of the incremental transitive closure problem, also known as all-pairs reachability, in directed graphs.

Incremental transitive closure with predictions

Input offline: predicted sequence of edge insertions $\hat{e}_1, \hat{e}_2, \ldots, \hat{e}_m \in E$.

Input online: sequence of interleaved updates and queries, i.e.,
       – edge insertions $e_1, e_2, \ldots, e_m \in E$, and
       – reachability queries $(s_1, t_1), (s_2, t_2), \ldots, (s_q, t_q) \in V \times V$.

Output: for each query (s_i, t_i),
       – YES if there is a path from s_i to t_i in the graph at the current moment,
       – NO otherwise.

The decremental variant is very similar; the difference is that the predicted sequence tells the algorithm in which order the edges are supposed to be deleted, not inserted, and that the online update operations are edge deletions, not edge insertions. In the incremental setting, it is natural to assume we start with an empty graph. However, in the decremental setting (also for classic algorithms without predictions), the initial graph has to be given upfront. In our setting with predicted sequence of updates, the initial graph can be considered to be given implicitly by the set of all edges appearing in the sequence. On the other hand, we can also think of an equivalent input format in which we are first given the initial graph $G = (V, E)$ where each edge $e \in E$ comes with an additional label $\hat{t}(e) \in [m]$ describing the predicted position of this edge in the sequence of deletions.

The second problem that we consider in this setting is the $(1 + \epsilon)$-approximate all-pairs shortest paths (APSP) problem in unweighted directed graphs. The problem definition is similar to the above transitive closure definition

– the only difference is that each query (s_i, t_i) has to output a number in range $[d(s_i, t_i), (1 + \epsilon) \cdot d(s_i, t_i)]$, where $d(s_i, t_i)$ denotes the length of a shortest path from s_i to t_i at the current moment.

The query times of our algorithms for these problems depend on prediction error η that has a somewhat technical definition. We discuss this in detail in Sections 2.1 and 3. For now let us only say that η can be upper bounded by the largest absolute difference between the predicted and actual insertion (deletion) time of an edge – i.e., the ℓ_∞ error of the predictions – which we denote by $\eta_\infty \overset{\text{def}}{=} \max_{e \in E} |t(e) - \hat{t}(e)|$, where $t(e)$ denotes the actual update time of an edge (i.e., $t(e_i) \overset{\text{def}}{=} i$), and $\hat{t}(e)$ denotes the predicted update time of an edge (i.e., $\hat{t}(\hat{e}_j) \overset{\text{def}}{=} j$). We discuss the choice of this error measure in Section 1.2.

Our partially dynamic algorithms with predictions are specified in Theorem 1.1.

THEOREM 1.1. (FULL DETAILS IN THEOREMS 3.1 AND 3.2) *Each of the following dynamic graph problems:*

- *incremental transitive closure,*
- *decremental transitive closure,*
- *incremental $(1 + \epsilon)$-approximate unweighted all-pairs shortest paths,*
- *decremental $(1 + \epsilon)$-approximate unweighted all-pairs shortest paths*

with a predicted sequence of edge updates can be solved by a deterministic algorithm with $\tilde{O}(n^{(3+\omega)/2})$ preprocessing time, $\tilde{O}(1)$ worst-case edge update time, and $\tilde{O}(\eta^2)$ query time, for prediction error $\eta \leqslant \eta_\infty$.

Without predictions, both these problems (transitive closure and approximate APSP) in both partially dynamic variants (incremental and decremental) can be solved by algorithms with $O(n^3)$ total update time and $O(1)$ query time [34, 35, 56, 20, 21, 9, 58, 41, 10], and under the OMv hypothesis this running time is tight (up to subpolynomial factors) even if one allows up to $O(n^{1-\varepsilon})$ query time [31]. In other words, classic algorithms without predictions face the cubic time barrier for these problems, and we show it can be bypassed with sufficiently accurate predictions.

We also show that the dependence of the running time in Theorem 1.1 on the prediction error η_∞ cannot be improved to strictly subquadratic under the OMv hypothesis.

THEOREM 1.2. *Unless the OMv Hypothesis fails, there is no algorithm for incremental (decremental) transitive closure with predictions with arbitrary polynomial prepocessing time, worst-case $O(n^{1-\varepsilon})$ edge insertion (deletion) time, and $O(\eta_\infty^{2-\varepsilon})$ query time, for any $\varepsilon > 0$. This already holds with respect to amortized time bounds per operation.*

Finally, we note that the algorithms of Theorem 1.1 are not *robust* – for large enough prediction errors they can be slower than the best known classic algorithm – but they can be made robust, at least in the amortized-time sense, using the black-box approach described in Section 1.2.

Fully Dynamic Graph Problems with Predicted Vertex Updates. Here we consider fully dynamic variants of various graph problems on directed graphs with vertex updates: triangle detection, single source reachability, strong connectivity, cycle detection, maximum matching, number of vertex disjoint st-paths. For *online* vertex updates, there is no dynamic algorithm with $O(n^{2-\epsilon})$ update time for any constant $\epsilon > 0$, conditional on the OMv hypothesis [31]. All these problems (except triangle detection) can also be naively recomputed from scratch in $\tilde{O}(n^2)$ time [12, 19]. Thus no polynomial improvement over the trivial approach is possible for these online dynamic problems. For *offline* vertex updates, there is no dynamic algorithm with $O(n^{\omega-1-\epsilon})$ update time, conditional on the triangle detection hypothesis [2]. On the upper bound side, this is matched by [62, 13] with $O(n^{\omega-1})$ update time. Our Theorem 1.3 provides a smooth trade-off between the online and offline model.

We use η_i to denote the prediction error of the i-th performed update. The prediction is a sequence of vertex updates. It can happen that an update occurs earlier than initially predicted, i.e. an update is moved several positions ahead in the sequence. When the data structure performs the i-th update, η_i is how many positions this update occurs too early. If this i-th update was not predicted at all (i.e. it doesn't even occur in the predicted sequence), we define $\eta_i = \infty$.

THEOREM 1.3. (FULL DETAILS IN THEOREM 5.1) *Fully dynamic triangle detection, single source reachability, strong connectivity, directed cycle detection, maximum matching size, number of vertex disjoint st-paths, with $\Theta(n)$ vertex updates and predictions can be solved in $O(n^\omega + n \sum_i \min\{\eta_i, n\})$ total time. More precisely, we have $O(n^\omega)$ preprocessing time and the i-th update takes $\tilde{O}(n^{\omega-1} + n \cdot \min\{\eta_i, n\})$ time.*

We remark that one can interpret our total time complexity as scaling with the ℓ_1-norm of the error. For $\eta_i' := \min\{\eta_i, n\}$, $\eta' \in \mathbb{N}^n$ we have total time $O(n^\omega + n\|\eta'\|_1)$ for n updates.

This result is obtained via reductions by [60, 61, 13], which reduce these dynamic graph problems to dynamic matrix inverse. In dynamic matrix inverse, we are given a dynamic $n \times n$ matrix $\mathbf{M}$ and must maintain information about $\mathbf{M}^{-1}$. The case of vertex updates on graph reduces to row and column updates to $\mathbf{M}$ (i.e. updates can replace one row and column of $\mathbf{M}$ at a time). The offline model with only column updates was studied previously in [62, 13] and for entry updates in [38]. The online model with row and column updates was studied in [60]. We construct a dynamic matrix inverse algorithm with predictions in Section 4 which then implies Theorem 1.3 via reductions from [60, 61].

Fully Dynamic Graph Problems with Predicted Deletion Times. The last setting that we study is also a fully dynamic one but with a weaker prediction requirement than above. First, only deletions are predicted; the algorithm needs no prior knowledge of insertions. Second, deletions are predicted only at the time of corresponding insertions. In other words, compared to a classic fully dynamic setting, the only difference is that each insertion comes with an additional number predicting when the currently inserted item (e.g., vertex or edge) is going to be deleted.

This model is inspired by a recent result by [54] in which they assume an offline sequence of predicted deletions – i.e. deletion times have no error – but in our case we can handle deletion errors. We first extend their techniques to give the following result:

THEOREM 1.4. *Consider a sequence of T updates and suppose we are given an incremental dynamic algorithm with worst-case update time Γ. Assume also that at any point in time we have a prediction on the order of deletions of current items, such that for the i inserted item the error η_i indicates the number of elements predicted to be earlier than i-th item that actually arrive later ($\eta_i = 0$ if the prediction is correct or the element arrives later). Then we have a fully-dynamic algorithm with the following guarantees:*

- *An insertion update is performed in $O(\Gamma \log^2 T)$ worst-cast time.*

- *The deletion of the i-th element can be performed in $O((1 + \eta_i) \cdot \Gamma \log^2 T)$ worst-case time.*

We can use this reduction, combined with an incremental APSP algorithm observed by [63] to get the following:

THEOREM 1.5. *Given a weighted and directed graph undergoing online vertex insertions and predicted vertex deletions, we can maintain exact weighted all-pairs shortest paths with the following guarantees:*

- *An insertion update can be performed $O(n^2 \log^2 n)$ worst-cast time.*

- *A deletion of the i-th inserted vertex v_i can be performed in $O(n^2(\eta_i \log^2 n + 1))$ worst-case time, where error $\eta_i \in [0, n]$ indicates how many vertices were predicted to be deleted before v_i that are actually deleted after v_i.*

This can be compared to a recent fully dynamic worst-case exact APSP bound of $\tilde{O}(n^{2.5})$ by [48] improving upon a long line of work on sub-cubic update times for APSP [63, 3, 29, 16]. We note that $n^{2.5}$ seems to be a natural barrier inherent to current algorithmic approaches for this problem, but there is no known conditional lower bound formalizing this intuition.

1.2 Further Discussion.

Consistency, Robustness, and Smoothness. Typically, algorithms with predictions are designed with three goals in mind: (1) *consistency*, that is a near-optimal (or at least better than worst-case) performance when predictions are accurate; (2) *robustness*, that is retaining worst-case guarantees of classic algorithms even when predictions are adversarial; and (3) *smoothness*, that is a graceful degradation of algorithm's performance with increasing prediction error, providing an interpolation between the former two extremes. Let us discuss how these goals translate to our model of dynamic algorithms with predictions.

In this context, consistency alone is just equivalent to having an offline algorithm that is faster than the fastest known (or, even better, fastest conditionally possible) online algorithm.

Robustness can often be dealt with by black-box best-of-both-worlds types of arguments. It sometimes becomes an issues in contexts where the performance measure of choice is the competitive ratio, but it is rarely an issue when we optimize the running time. For static algorithms, one can just simulate two algorithms – one with predictions,

Table 1: Summary of our results. Each column describes one group of results, built around one technique or data structure.

	Theorem 1.1	Theorem 1.3	Theorem 1.4
Setting:	partially dynamic (incremental or decremental)	fully dynamic (insertions and deletions)	fully dynamic (insertions and deletions)
Type of updates:	edge updates	vertex updates	vertex updates
Predictions:	sequence of all updates (insertions or deletions)	sequence of next n updates	deletion times (given during insertions)
Running time:	preprocessing: $\tilde{O}(n^{(3+\omega)/2})$ update: $\tilde{O}(1)$ query: $\tilde{O}(\eta_\infty^2)$	preprocessing: $O(n^\omega)$ update: $O(n^{\omega-1} + n \cdot \min\{\eta_i, n\})$	no preprocessing insertion: $\tilde{O}(n^2)$ deletion: $\tilde{O}(\eta_i n^2)$
Error measure:	ℓ_∞	ℓ_1	ℓ_1
Applications to graph problems:	transitive closure $(1+\epsilon)$-approximate APSP	triangle detection single-source reachability strong connectivity directed cycle detection maximum matching size #vertex-disjoint st-paths	exact APSP
Main technical tool:	All-Pairs Bottleneck Paths	dynamic matrix inverse	reduction to incremental

and another one with best known worst-case guarantees – step by step, in parallel, and stop whenever one of these algorithms stops. This approach incurs only a factor-of-two multiplicative slowdown (which is negligible for asymptotic complexity) compared to the faster algorithm, on a per-instance basis. For dynamic algorithms, we need a more careful approach: Whenever the currently faster algorithm finishes processing a request, stop and return its answer; when a new request comes, resume the simulation from where it stopped, letting the slower algorithm possibly catch up. This way we can retain amortized running time guarantees of the better of the two algorithms, on a per-instance basis. We remark that it seems challenging to have a similar black-box tool for worst-case per request guarantees.

Smoothness is perhaps the least well defined of the three terms. Intuitively, we want the algorithms to tolerate as big prediction errors as possible without compromising on performance too much. For dynamic algorithms with predictions, some level of smoothness can always be achieved trivially. Assuming the best offline algorithm is polynomially faster than the best online algorithm – which is anyway required to claim consistency – one can always rerun the offline algorithm from scratch after encountering each difference between the predicted and the actual input sequence, and therefore tolerate some polynomial number of errors. We achieve better smoothness than this baseline benchmark by (1) incorporating error measures that distinguish between small errors and large errors, (2) getting better dependence on them, and (3) sometimes even showing that this dependence is conditionally optimal.

Predictions, Prediction Errors, and Learnability. We note that the predictions that we use, and error measures that quantify predictions accuracy, are standard in the learning-augmented literature. Predictions of the entire input sequence (theorem 1.1) are used, e.g., for scheduling problems [8], predicting only a certain window of input sequence (theorem 1.3) is required for learning-augmented weighted caching [36], and predictions of the time of the next operation concerning the current item (theorem 1.4) is the by-now-standard setup for unweighted caching [47].

The most ubiquitous way of measuring how for the prediction is from the truth is the ℓ_1-distance (e.g., [47, 22, 4], and many more), but for certain problems (e.g., flow time minimization with uncertain processing times [5, 6]) the ℓ_∞-error of predictions is a more natural (and sometimes even necessary) choice.

Our prediction errors can be illustrated with an example of road networks: Every day the same roads get congested during rush hour, so one can try to predict the updates in a dynamic road network. However, such predictions will not be perfect, because the exact order in which roads become congested may differ from day to day.

Since all our predictions are essentially permutations, the question of when such predictions can be efficiently learned is addressed by standard tools in the literature [30, 39].

1.3 Related Work. Over the past couple of years the field of learning-augmented algorithms blossomed enormously, and it is implausible to list here all relevant contributions. We refer the interested reader to survey articles by Mitzenmacher and Vassilvitskii [49, 50] and a website with a list of papers maintained by Lindermayr and Megow [44]. There are numerous works on using predictions for improving competitive ratios of online problems (e.g., [47, 57, 8, 4, 7], and many, many more) and running times of static problems (e.g., [22, 17]). Dynamic algorithms can be seen as a certain kind of data structures, and there are already several examples of learning-augmented data structures (see, e.g., [40, 27, 43]), but they focus primarily on index data structures, such as binary search trees, and hence they are not directly related to our work.

A concept related to offline dynamic graph algorithms is that of a graph timeline, as defined by [42], in which a sequence of graphs $G_1, \ldots, G_T$ is given upfront and any two subsequent graphs differ by only one edge being added or removed. [37] studies in this model several types of undirected connectivity queries over a time range, asking, e.g., if a path exists in at least one graph in a given interval.

Finally, there is a separate line of work on temporal graphs (see for instance the survey [33]), also related to the offline model, in which each edge is labelled with (a collection of) time intervals indicating when it is available. Often the goal in this line of work is understanding certain dynamics on networks (like information diffusion or convergence to certain properties), which is different from our computation efficiency objectives in the dynamic settings.

Concurrent Work. In the concurrent and independent work, Liu and Srinivas [45] consider the same predicted-deletions model as in our Theorem 1.4. Their result is also based on a reduction from the fully-dynamic setting to the incremental setting. However, unlike our work, their algorithm does not directly rely on a similar reduction by [54], whereas we use analysis of [54] as a black-box. We note that [45] present many other applications (e.g. all-pairs max-flow/min-cut approximation, or uniform sparsest cut) in the predicted-deletions model that can also be derived from Theorem 1.4. On the technical side, they also show how to handle the case where number of updates T is not known upfront, which we do not consider.

In another independent work, Henzinger, Saha, Seybold, and Ye [32] initiate a systematic study of the time complexity of dynamic graph algorithms with predictions. While their focus is on conditional fine-grained lower bounds, they also provide some algorithms. In particular, their combinatorial (i.e., not using fast matrix multiplication) algorithms for transitive closure, approximate APSP, and triangle detection have bounds similar to our Theorems 1.1 and 1.3 but with worse preprocessing times. They also consider prediction models with error measures very different from ours.

1.4 Notation. We write $O(n^\omega)$ for the time complexity of multiplying two $n \times n$ matrices, where the current best bound is $\omega < 2.372$ [24]. For the matrix product of rectangular matrices, we write $\mathrm{MM}(a,b,c)$ for the time complexity of multiplying $a \times b$ and $b \times c$ matrices. We write $\mathbf{I}$ for the identity matrix.

All the graphs considered in this paper are directed.

2 Technical Overview.

In this section we briefly explain the main ideas behind our results.

2.1 Partially Dynamic Algorithms (Section 3). Our algorithms for partially dynamic problems – transitive closure, approximate APSP, exact SSSP – use a connection between these problems and the all-pairs bottleneck paths (APBP) problem. The latter is a variant of the all-pairs shortest paths (APSP) problem in which, instead of minimizing the sum of edge weights, we minimize the maximum edge weight along a path. As opposed to APSP,

which is conjectured to require cubic time [59, 65], APBP can be solved in strongly subcubic time [64], and the best known APBP algorithm runs in $O(n^{(3+\omega)/2}) \leqslant O(n^{2.687})$ time [23].

Transitive Closure. First, let us explain the connection of partially dynamic transitive closure with APBP. If we use edge insertion times as edge weights, and solve APBP, we obtain a matrix B such that

$$B[u,v] = \min\{\max_{e \in \mathcal{P}}\{\text{insertion time of } e\} \mid \mathcal{P} \in uv\text{-paths}\}.$$

Hence, $B[u,v] \leqslant k$ if and only if there is a path from u to v in the graph after the first k insertions. This observation itself is sufficient to solve incremental[1] transitive closure in the offline setting (in other words, with perfect predictions) faster than the OMv-based cubic time lower bound for the online setting.

Let us now explain how we handle prediction errors. After each edge insertion, we keep track of the longest prefix of the predicted sequence of updates that contains only the already inserted edges. Let us denote the length of this prefix by p. We also maintain the set of "out-of-order" edges E_{err} that have been already inserted but are not contained in that prefix. Upon receiving a reachability query (u,v) we construct an auxiliary graph H on at most $2|E_{err}| + 2$ nodes: the endpoints of edges in E_{err} and nodes u and v. For every pair of nodes $x, y \in H$, we add an edge (x,y) to H if $(x,y) \in E_{err}$ or if $B[x,y] \leqslant p$. It is easy to see that there is a path from u to v in H if and only if there is a path from u to v in the original graph. Constructing H and finding a uv-path takes time $O(|E_{err}|^2)$, and we show that the number of out-of-order edges $|E_{err}|$ can be bounded by the ℓ_∞-error of the predictions. The same approach can be adapted to the decremental setting.

Approximate APSP. With an $O(\epsilon^{-1} \log n)$ overhead, in addition to answering queries on whether there is a path from u to v, we are also able to report the length of a shortest path within up to $1 + \epsilon$ multiplicative approximation error. Instead of the single matrix B, for every $d \in \{(1+\epsilon)^0, (1+\epsilon)^1, \ldots, (1+\epsilon)^{\log_{1+\epsilon}(n)}\}$, we compute matrix $B^{(d)}$ of bottleneck paths with up to d hops. Each such matrix can be computed in $O(n^{(3+\omega)/2} \log d)$ time by repeatedly squaring the input weight matrix using (min, max)-product [23]. Note that $B^{(d)}[u,v] \leqslant k$ if and only if there is a path from u to v of length at most d in the graph after the first k insertions. Now, we can equip the auxiliary graph H with edge weights corresponding to $(1 + \epsilon)$-approximate distances in the original graph, and answer the queries by running the Dijkstra algorithm in H.

2.2 Fully Dynamic Matrix Inverse with Predictions (Section 4).
By using standard reductions from dynamic graph problems to dynamic matrix inverse (see, e.g., [60, 61, 13]), Theorem 5.1 reduces to maintaining the matrix inverse of some matrix $\mathbf{M}$ undergoing rank-1 updates, i.e. updates where we are given two vectors u, v and then set $\mathbf{M} \leftarrow \mathbf{M} + uv^\top$. For these reductions, it suffices to return $v^\top \mathbf{M}^{-1}$ after each update.

From Rank-1 to Entry Updates. In general, without predictions, rank-1 updates are strictly harder than entry updates (i.e. updates that change only a single entry of the matrix $\mathbf{M}$ at a time). Rank-1 updates require $\Omega(n^2)$ update time [31], whereas entry updates can be handled in $O(n^{1.406})$ update time [13]. However, in the prediction setting, one can actually reduce dynamic matrix inverse with rank-1 updates to dynamic matrix inverse with entry updates, i.e. we can reduce updates for general dense u and v to the special case where u and v are sparse. (Entry updates are just the special case where u, v have one non-zero entry each.)

Let $(u^{(1)}, v^{(1)}), \ldots, (u^{(n)}, v^{(n)})$ be the next n predicted rank-1 updates. Then we can describe the rank-1 updates as follows. Let $\mathbf{U}$ and $\mathbf{V}$ be the $n \times n$ matrices obtained by stacking the vectors $(u^{(t)})_{t=1,\ldots,n}$ and $(v^{(t)})_{t=1,\ldots,n}$ next to each other. Let $\mathbf{D}$ be a diagonal matrix that is initially all zero, and consider the matrix formula:

$$(2.1) \qquad f(\mathbf{M}, \mathbf{U}, \mathbf{V}^\top, \mathbf{D}) = \mathbf{V}^\top (\mathbf{M} + \mathbf{U}\mathbf{D}\mathbf{V}^\top)^{-1}.$$

Here, switching the diagonal entries of $\mathbf{D}$ one-by-one from 0 to 1 corresponds to adding $u^{(t)}(v^{(t)})^\top$ to $\mathbf{M}$ and then inverting the result. Thus, the task of maintaining the inverse of $\mathbf{M}$ subject to rank-1 updates, while returning $(v^{(t)})^\top \mathbf{M}^{-1}$ after each update, can be reduced to the task of returning the t-th row of $f(\mathbf{A}, \mathbf{U}, \mathbf{V}^\top, \mathbf{D})$ subject to entry updates to $\mathbf{D}$. In [11], v.d.Brand has shown that any dynamic matrix formula that can be written using the basic matrix operations (addition, subtraction, multiplication and inversion), such as formula f in (2.1), reduces to dynamic matrix inverse again (Lemma 4.1), while supporting the same kind of updates and queries. Thus, we must maintain the inverse of a certain matrix that is subject to *entry* updates while supporting queries to its rows,

---

[1]Or decremental! In the offline variant they are equivalent.

because the input to our formula f only receives entry updates and we only require rows of f. Since we only need to consider entry updates, that means we can now focus only on the special case where we receive rank-1 updates where both u and v are sparse with only one non-zero entry each. Only if we receive an update that was not predicted at all, do we need to perform a rank-1 update with dense vectors.

Matrix Inverse with Predictions. Dynamic matrix inverse in the offline model where all updates are given ahead of time was solved in $O(n^\omega)$ total time by Sankowski and Mucha [62], and later generalized by v.d.Brand, Nanongkai and Saranurak [13] to only require the sequence of column indices of all future updates but not the actual entries of the new columns. These previous data structures are offline, i.e. require correct predictions about the entire update sequence ahead of time and cannot support updates that differ from the prediction received during initialization. Building on their techniques, we construct a dynamic algorithm with predictions that is robust against inaccurate predictions.

Let $\mathbf{M}$ be the dynamic input matrix. We write $\mathbf{M}^{(i)}$ for a variant of $\mathbf{M}$ that is updated only every 2^i iterations. So $\mathbf{M}^{(0)}$ is always identical to $\mathbf{M}$ and $\mathbf{M}^{(i)}$ is identical to $\mathbf{M}$ every 2^i iterations.

We maintain these matrices for $i = 0, 1, \ldots, \log n$ in the following implicit form. That is, only the matrices $\mathbf{L}^{(i)}, (\mathbf{R}^{(i)})^\top \in \mathbb{F}^{n \times 2^i}$ are stored in memory where

$$(\mathbf{M}^{(i)})^{-1} = (\mathbf{M}^{(i+1)})^{-1}(\mathbf{I} + \mathbf{L}^{(i)}\mathbf{R}^{(i)}).$$

The matrix $(\mathbf{M}^{(\log n)})^{-1}$ is also stored explicitly in memory. Note that maintaining $(\mathbf{M}^{(\log n)})^{-1}$ takes $O(n^{\omega-1})$ amortized time, as we recompute this inverse in $O(n^\omega)$ time every $2^{\log n} = n$ updates.

Via the Woodbury matrix identity, one can show (Lemma 4.4) that the matrices $\mathbf{L}^{(i)}$ and $\mathbf{R}^{(i)}$ are of the form:

$$\mathbf{R}^{(i)} = (\mathbf{V}^{(i)})^\top(\mathbf{M}^{(i+1)})^{-1}, \quad \mathbf{L}^{(i)} = \mathbf{U}^{(i)}(\mathbf{I} + \mathbf{U}^{(i)}\mathbf{R}^{(i)})^{-1},$$

where $\mathbf{U}^{(i)}, \mathbf{V}^{(i)}$ are given by the at most 2^i vectors u, v of the past at most 2^i updates of the form $uv^\top$ by which $\mathbf{M}^{(i)}$ and $\mathbf{M}^{(i+1)}$ differ. (These vectors will have one non-zero entry each, since reduced to entry updates.) Here $\mathbf{R}^{(i)}$ is composed of some (at most 2^i many) rows of $(\mathbf{M}^{(i)})^{-1}$, because we have entry updates, and thus we can assume each v to be a standard unit vector. Therefore, we can compute $\mathbf{L}^{(i)}$ and $\mathbf{R}^{(i)}$ in $O((T_i + \mathrm{MM}(n, 2^i, 2^i))/2^i)$ amortized time, where T_i is the time required to obtain the at most 2^i rows of $(\mathbf{M}^{(i+1)})^{-1}$. If our predictions are correct, then when we previously computed $\mathbf{L}^{(i+1)}, \mathbf{R}^{(i+1)}$, we could have also precomputed the required rows of $(\mathbf{M}^{(i+1)})^{-1}$ in $O(\mathrm{MM}(n, 2^{i+1}, 2^{i+1}))$ time, which is subsumed by the time required to compute $\mathbf{L}^{(i+1)}, \mathbf{R}^{(i+1)}$. Thus for correct predictions, we can assume $T_i = n2^i$. This leads to $O(\sum_i \mathrm{MM}(n, 2^i, 2^i)/2^i) = O(n^{\omega-1})$ amortized time per update[2].

A similar idea of maintaining $O(\log n)$ copies of matrix $\mathbf{M}$ that are updated every 2^i iterations was also used in [62, 13] but they did not handle incorrect predictions efficiently.

Now, observe what happens in our dynamic algorithm if a prediction is incorrect, i.e. we perform an update in a column that was originally predicted to occur some η iterations into the future. In that case the required rows of $(\mathbf{M}^{(i+1)})^{-1}$ might not be precomputed. However, the rows are precomputed in $(\mathbf{M}^{(j)})^{-1}$ for every $j \leq \min\{\log \eta, \log n\}$. We can wlog assume that the rows are precomputed in $(\mathbf{M}^{(\log n)})^{-1}$ because the entire inverse is computed from scratch every n iterations. Thus the missing row must only be computed in $(\mathbf{M}^{(\ell)})^{-1}$ for $\ell = 0, 1, \ldots, \min\{\log \eta, \log n\}$. So we obtain an additional $O(\sum_{\ell=0}^{\min\{\log \eta, \log n\}} n2^\ell) = O(n\min\{\eta, n\})$ cost for each update that occurs η iterations earlier than initially predicted.

At last, consider what happens if we perform an update that was not predicted at all, i.e., we receive two vectors u, v for a rank-1 update. Since the update was not predicted, the previous reduction does not hold and the vectors remain dense. If v is dense, computing the respective row of $\mathbf{R}^{(i)}$ is not just copying a row of $(\mathbf{M}^{(i+1)})^{-1}$ but rather it requires computing a vector-matrix product. Computing this product is done recursively, i.e.,

$$v^\top(\mathbf{M}^{(i)})^{-1} = v^\top(\mathbf{M}^{(i+1)})^{-1}(\mathbf{I} + \mathbf{L}^{(i+1)}\mathbf{R}^{(i+1)}) = v^\top(\mathbf{M}^{(\log n)})^{-1}\prod_{j=i+1}^{\log(n)-1}(\mathbf{I} + \mathbf{L}^{(j)}\mathbf{R}^{(j)}),$$

which takes $O(n^2)$ operations. Note that for $i = 0$, this actually computes $v^\top(\mathbf{M}^{(j)})^{-1}$ for all $j = 0, 1, \ldots, \log n$ at once within $O(n^2)$ time.

---

[2]We focus here in the outline on amortized complexity, but this can be made worst-case (see Section 4).

2.3 Fully Dynamic Algorithms with Predicted Deletion Times (Section 6). We also consider the fully dynamic model in which predictions give no information about insertions whatsoever but the relative ordering of deletions is predicted – by specifying for each item, at the time of its insertions, the position of its future deletion. Our algorithm is based on a result by Peng and Rubinstein [54][3] that gives a reduction from a *fully dynamic semi-online* data structure, in which the order of deletions is given exactly, to an *insert-only online* data structure. In particular, assuming that the insert-only data structure has *worst-case* update time Γ, their semi-online data structure has update time $O(\Gamma \log T)$ for a sequence of T updates. We extend this reduction to the case where this order of deletions is not known exactly but it is predicted with some errors. The error for the i-th element is denoted by η_i, indicating that there are η_i deletions that were predicted to happen before the deletion of element i but will arrive after its deletion. This error incurs an additional worst-case update time overhead of roughly $O(\eta_i \Gamma)$.

At a high-level, the idea of Peng and Rubinstein [54] is that if the current list of the already performed insertions happens to be in the reverse order of deletion times, then a deletion can be performed in time $O(\Gamma)$ by simply rewinding the computation of the most recently performed insertion. Moreover, at any point in time, one can rewind some recent insertions and then re-insert these elements in a different order, to better prepare for future deletions. Since re-ordering the elements at each update would be expensive, they get an amortized bound by maintaining a sequence of buckets that keep partial reverse orderings. The amortized bound follows by ensuring that a set of $O(2^j)$ elements are re-ordered in every 2^j updates for each $j = 0, \ldots, \lceil \log T \rceil$. In our case, when the deletion of the i-th element arrives η_i positions *earlier* than predicted, we rewind the computation of the last $\eta_i + 1$ insertions, until we get to delete the correct element, in time $O(\eta_i \Gamma)$, and then re-insert the η_i unnecessarily deleted elements.

3 Partially Dynamic Algorithms.

In this section we prove our upper bounds (Theorem 1.1) and lower bounds (Theorem 1.2) for partially dynamic graph problems with predictions.

First, let us introduce two closely related concepts – all-pairs bottleneck paths and $(\min, \max)$-product – that we heavily use throughout the section. In the all-pairs bottleneck paths (APBP) problem, we are given a directed graph $G = (V, E)$ with edge weights $w : E \to \mathbb{N}$, and we have to compute a matrix $B \in \mathbb{N}^{V \times V}$ with

$$B[u, v] \overset{\text{def}}{=} \min\{\max_{e \in \mathcal{P}} w(e) \mid \mathcal{P} \in uv\text{-paths}\}.$$

APBP can be solved in $O(n^{(3+\omega)/2}) \leqslant O(n^{2.687})$ time [23]. The $(\min, \max)$-product of two $n \times n$ matrices A, B is defined as $(A \otimes B)[i, j] \overset{\text{def}}{=} \min_k \max\{A[i, k], B[k, j]\}$. It can also be computed in $O(n^{(3+\omega)/2})$ time [23]. Now, let us explain the relation between the two. Consider a directed graph $G = (V, E)$ with edge weights $w : E \to \mathbb{Z}$, and let W denote the corresponding weight matrix, i.e., $W[u, v] = w(u, v)$ if $(u, v) \in E$, $W[u, v] = +\infty$ if $(u, v) \notin E$, and $W[u, u] = -\infty$. Observe that, for $d \in \mathbb{Z}_+$, the $(\min, \max)$-product of d copies of W gives all-pairs bottleneck paths with up to d hops:

$$(\underbrace{W \otimes W \otimes \cdots \otimes W}_{d \text{ times}})[u, v] = \min\{\max_{e \in \mathcal{P}} w(e) \mid \mathcal{P} \in uv\text{-paths}, |\mathcal{P}| \leqslant d\}.$$

Such a product can be computed by the binary exponentiation in $O(n^{(3+\omega)/2} \log d)$ time. For $d = n$, we get exactly APBP, and the extra $\log n$ factor can be avoided [64].

3.1 Upper Bounds. Now we proceed to describe our partially dynamic algorithm for $(1 + \epsilon)$-approximate APSP with predictions. Since transitive closure is a strictly easier problem, Theorem 1.1 will follow. We begin with the incremental variant. Recall that $\hat{e}_1, \hat{e}_2, \ldots, \hat{e}_m$ denotes the predicted sequence of updates, and $e_1, e_2, \ldots, e_m$ the actual one.

THEOREM 3.1. *Incremental $(1+\epsilon)$-approximate all-pairs shortest paths in unweighted directed graphs with predicted sequence of edge updates can be solved by a deterministic algorithm with $O(n^{(3+\omega)/2} \log^2 n)$ preprocessing time and $O(\log n)$ worst-case edge insertion time. Each query that is asked between the i-th and $(i + 1)$-th insertion requires*

---

[3]A reduction similar to [54], but with only an amortized update time guarantee, was also given by [15].

$O(\bar{\eta}_i^2 \log\log n) = O(\eta_\infty^2 \log\log n)$ *time, where $\bar{\eta}_i$ is the current number of edges in the graph that are not contained in the longest prefix of the predicted sequence that has already been inserted, i.e.,*

$$\bar{\eta}_i \stackrel{\text{def}}{=} i - \max \{j \mid \{\hat{e}_1, \hat{e}_2, \ldots, \hat{e}_j\} \subseteq \{e_1, e_2, \ldots, e_i\}\}.$$

Before proving the theorem, let us explain how the above measure of prediction error $\bar{\eta}_i$ can be upper bounded by the more standard ℓ_∞-error, denoted by η_∞. Let $\Pi \in S_m$ denote the permutation of predicted insertions corresponding to the actual sequence of insertions, i.e., $e_1, e_2, \ldots, e_m = \hat{e}_{\Pi(1)}, \hat{e}_{\Pi(2)}, \ldots, \hat{e}_{\Pi(m)}$. With this notation, we have $\bar{\eta}_i = i - \max \{j \mid \{1, 2, \ldots, j\} \subseteq \{\Pi(1), \Pi(2), \ldots, \Pi(i)\}\}$. Our goal is to show that $\bar{\eta}_i \leqslant \eta_\infty \stackrel{\text{def}}{=} \max_j |j - \Pi(j)|$, for every $i \in [m]$. Fix $i \in [m]$, and let $k = \Pi^{-1}(i - \bar{\eta}_i + 1)$. Note that, by definition of $\bar{\eta}_i$, it holds that $i - \bar{\eta}_i + 1 \notin \{\Pi(1), \Pi(2), \ldots, \Pi(i)\}$. In other words, $k \geqslant i + 1$. Then, $\eta_\infty \geqslant k - \Pi(k) \geqslant i + 1 - (i - \bar{\eta}_i + 1) = \bar{\eta}_i$, as desired.

Proof. [Proof of Theorem 3.1] Upon receiving the predicted sequence of edge insertions, the algorithm creates a weighted directed graph with edge set $E = \{\hat{e}_1, \hat{e}_2, \ldots, \hat{e}_m\}$ and edge weights $w : E \to \mathbb{Z}$ equal to predicted insertion times, i.e., $w(\hat{e}_i) = i$ for every $i \in [m]$. Then, for every $d \in \{(1+\epsilon)^0, (1+\epsilon)^1, \ldots, (1+\epsilon)^{\log_{1+\epsilon}(n)}\}$, the algorithm computes matrix $B^{(d)}$ of bottleneck paths with up to $\lceil d \rceil$ hops. Observe that $B^{(d)}[u, v] \leqslant k$ if and only if there is a path from u to v of length at most d using only edges from $\{\hat{e}_1, \hat{e}_2, \ldots, \hat{e}_k\}$. Computing all $B^{(d)}$'s takes $O(n^{(3+\omega)/2} \log^2 n)$ time in total.

On top of that, the algorithm creates a dictionary data structure (e.g., a balanced BST) that will allow translating edges given as pairs of nodes to their indices in the predicted sequence of insertions, and another BST that will maintain the set S of indices of already inserted edges (initially, $S = \emptyset$. This ends the preprocessing phase.

When an edge (u, v) is inserted, the algorithm first finds its index in the predicted sequence of insertions, i.e., j such that $\hat{e}_j = (u, v)$, and then simply adds j to S. This takes $O(\log n)$ time.

To handle a query (u, v) the algorithm proceeds as follows. Let $i = |S|$ be the number of insertions so far. The algorithm first finds the largest prefix of the predicted sequence of insertions that has been already inserted, i.e., the largest j such that $\{1, 2, \ldots, j\} \subseteq S$. This is simply the smallest positive integer not in S minus one, and it can be found in $O(\log n)$ time assuming the BST maintains sizes and value ranges of its subtrees. Then, the algorithm uses the BST to list "out-of-order" edges E_{err} that have been already inserted but are not contained in that prefix – these correspond to elements of S larger than j. In other words, the current edge set of the graph is exactly $\{\hat{e}_1, \hat{e}_2, \ldots, \hat{e}_j\} \cup E_{err}$. Note that $|E_{err}| = i - j = \bar{\eta}_i$, and E_{err} can be constructed in $O(\bar{\eta}_i \log n)$ time.

Next, the algorithm creates a directed weighted auxiliary graph H. The nodes of H are endpoints of edges in the list E_{err} and nodes u and v, that is at most $2\bar{\eta}_i + 2$ nodes in total. The edges of H are of two kinds. First, there are all the edges from E_{err}, each with weight 1. Second, for every pair of nodes $x, y \in H$, the algorithm selects the smallest d such that $B^{(d)}[x, y] \leqslant j$, and, if such d exists, it adds to H edge (x, y) with weight d. This second type of auxiliary edges represents paths using only edges from $\{\hat{e}_1, \hat{e}_2, \ldots, \hat{e}_j\}$, and their weights are upper bounds of the path lengths within up to $(1 + \epsilon)$ multiplicative approximation error. It takes $O(\bar{\eta}_i^2 \log\log n)$ time to construct H.

Finally, the algorithm finds a (weighted) shortest path from u to v in H, which takes $O(\bar{\eta}_i^2)$ time, using the Dijkstra algorithm. Let us justify that the (weighted) length of this path $d_H(u, v)$ is a correct $(1+\epsilon)$-approximation of the (unweighted) shortest path length in the original graph $d_G(u, v)$, i.e., that $d_H(u, v) \in [d_G(u, v), (1+\epsilon) \cdot d_G(u, v)]$. Clearly, $d_G(u, v) \leqslant d_H(u, v)$. For the remaining direction, fix a path of length $d_G(u, v)$ in the original graph. This path can be split into segments, each being either a single out-of-order edge or a subpath composed only of edges from $\{\hat{e}_1, \hat{e}_2, \ldots, \hat{e}_j\}$. Every such segment is represented by an edge in H and the weight of this edge is at most $(1 + \epsilon)$ times larger than the length of the segment. Hence, $d_H(u, v) \leqslant (1+\epsilon) \cdot d_G(u, v)$. $\square$

Now we discuss the decremental variant, which is very similar. The main difference is that we will be looking at suffixes of the predicted sequence of deletions that were not yet deleted.

THEOREM 3.2. *There is a deterministic decremental algorithm for $(1 + \epsilon)$-approximate all-pairs shortest paths in unweighted directed graphs with predicted sequence of edge deletions with $O(n^{(3+\omega)/2} \log^2 n)$ preprocessing time and $O(\log n)$ worst-case edge deletion time. Each query that is asked between the i-th and $(i + 1)$-st deletion requires $O(\bar{\eta}_i^2 \log\log n) = O(\eta_\infty^2 \log\log n)$ time, where $\bar{\eta}_i$ is the current number of edges in the graph that are not contained*

in the longest suffix of the predicted sequence that has not yet been deleted, i.e.,

$$\bar{\bar{\eta}}_i \stackrel{\text{def}}{=} \min \left\{ j \mid \{\hat{e}_j, \hat{e}_{j+1}, \ldots, \hat{e}_m\} \cap \{e_1, e_2, \ldots, e_i\} = \emptyset \right\} - i - 1.$$

Proof. The algorithm closely mimics the incremental algorithm given in the proof of Theorem 3.1. We highlight the differences.

In the preprocessing phase the algorithm also computes hop-bounded bottleneck paths $B^{(d)}$'s for $O(\log n)$ exponentially growing hop bounds d, but the difference is that now the edge weight of edge $\hat{e}_i$ is $w(\hat{e})_i = -i$. It follows that $B^{(d)}[u, v] \leqslant -k$ if and only if there is a path from u to v of length at most d using only edges from $\{\hat{e}_k, \hat{e}_{k+1}, \ldots, \hat{e}_m\}$.

Set S still contains indices of edges present in the graph. That is, initially $S = [m]$, and deleting an edge boils down to removing its index from S.

To handle a query, the algorithm represents the current edge set of the graph $\{\hat{e}_j, \hat{e}_{j+1}, \ldots, \hat{e}_m\} \cup E_{err}$, for j as small as possible. Note that $|E_{err}| = \bar{\bar{\eta}}_i$. The auxiliary graph H again contains u, v, and endpoints of E_{err}. For $(x, y) \notin E_{err}$, the weight of (x, y) in H is the smallest d such that $B^{(d)}[x, y] \leqslant -j$, if such d exists and otherwise edge (x, y) is not included in H. As before, $d_H(u, v) \in [d_G(u, v), (1 + \epsilon) \cdot d_G(u, v)]$.

Observe that, as before, $\bar{\bar{\eta}}_i \leq \eta_\infty$. □

3.2 Lower Bounds. In this section we show that the dependence of the running time of our partially dynamic algorithms on the prediction error η_∞ is (conditionally) optimal, at least in certain parameter regimes.

THEOREM 3.3. *Unless the OMv Hypothesis fails, there is no algorithm for incremental (decremental) transitive closure with predictions with arbitrary polynomial prepocessing time, worst-case $O(n^{1-\varepsilon})$ edge insertion (deletion) time, and $O(\eta_\infty^{2-\varepsilon})$ query time, for any $\varepsilon > 0$. This already holds with respect to amortized time bounds per operation.*

Proof. Henzinger et al. [31] proved that the OMv hypothesis implies that the following OuMv problem also cannot be solved in $O(n^{3-\varepsilon})$ time, for any $\varepsilon > 0$, even after arbitrary polynomial preprocessing time. In the OuMv problem we are first given Boolean matrix $M \in \{0, 1\}^{n \times n}$, and then we need to answer online n queries of the form: Given two Boolean vectors $u_i, v_i \in \{0, 1\}^n$, what is the Boolean product uMv?

We show how to reduce an instance of OuMv to an instance of incremental (decremental) transitive closure with predictions, on a graph with $O(n)$ nodes, with a request sequence containing $O(n^2)$ updates and $O(n)$ queries, and with the maximum prediction error $\eta_\infty = O(n)$. The reduction itself runs in $O(n^2)$ time. Therefore, under the OMv hypothesis, it cannot hold simultaneously that the preprocessing time is polynomial in n, the update time is truly sublinear in n, and the query time is truly subquadratic in η_∞.

We first focus on the incremental variant of the problem. We will think of the reduction as an algorithm solving the OuMv problem and having black-box access to an algorithm for incremental transitive closure with predictions. (We note that our reduction is modelled after a similar one in [31, Lemma 4.7 in the arXiv version], however we need to insert edges on two sides of the graph in order to reduce the number of queries and get a meaningful bound.)

Upon receiving matrix $M \in \{0, 1\}^{n \times n}$, the reduction creates a graph composed of four layers of n nodes each, and generates a predicted sequence of edge insertions. Let the vertex set be $V = \{a_1, \ldots, a_n\} \cup \{b_1, \ldots, b_n\} \cup \{c_1, \ldots, c_n\} \cup \{d_1, \ldots, d_n\}$, and let E_M denote the following set of edges between b-nodes and c-nodes, corresponding to matrix M,

$$E_M = \{(b_i, c_j) \mid (i, j) \in [n] \times [n], M[i, j] = 1\}.$$

The predicted sequence of edge insertions starts with all the edges from E_M, in an arbitrary fixed order, followed by

$$\begin{aligned}
&(a_1, b_1), (c_1, d_1), (a_1, b_2), (c_2, d_1), \ldots, (a_1, b_n), (c_n, d_1), \\
&(a_2, b_1), (c_1, d_2), (a_2, b_2), (c_2, d_2), \ldots, (a_2, b_n), (c_n, d_2), \\
&\ldots, \\
&(a_n, b_1), (c_1, d_n), (a_n, b_2), (c_2, d_n), \ldots, (a_n, b_n), (c_n, d_n).
\end{aligned}$$

The reduction gives this sequence to the algorithm to preprocess it, and then it inserts edges E_M, in the same order as in the sequence. This concludes the preprocessing phase, and the reduction starts accepting queries.

Upon receiving a pair of vectors $u_i, v_i \in \{0,1\}^{n \times n}$, the reduction first inserts edges from a_i to b-nodes that correspond to ones in u_i, and from c-nodes that correspond to ones in v_i to d_i, i.e.,

$$\{(a_i, b_j) \mid j \in [n], u_i[j] = 1\} \cup \{(c_j, d_i) \mid j \in [n], v_i[j] = 1\}.$$

At this point, the graph contains a path from a_i to d_i if and only if $uMv = 1$, so the reduction asks reachability query (a_i, d_i) and returns the answer. Finally, the reduction inserts remaining edges from a_i to b-nodes and from c-nodes to d_i, i.e.,

$$\{(a_i, b_j) \mid j \in [n], u_i[j] = 0\} \cup \{(c_j, d_i) \mid j \in [n], v_i[j] = 0\},$$

and it is ready to accept the next query.

Note that, both the predicted and actual insertion time for edge (a_i, b_j) are within the range $[|E_m| + 2n(i - 1), |E_m| + 2ni]$, so the prediction error for such edge is at most $2n$. The same is true for edges of the form (c_j, d_i), and the predicted insertion times for edges between b-nodes and c-nodes have no error. Hence, the maximum prediction error is $\eta_\infty \leqslant 2n$, as desired.

The construction proving hardness of the decremental variant of the problem is very similar. The difference is that we start with two full bipartite cliques – one between a-nodes and b-nodes, the other between c-nodes and d-nodes – and edges E_M between b-nodes and c-nodes. Then, in i-th OMv query, we first remove edges going from a_i and to d_i corresponding to zeros in u_i and v_i, respectively; after that we ask the reachability query (a_i, d_i), and finally we remove the remaining edges adjacent to a_i and d_i, which correspond to ones in u_i and v_i. □

4 Dynamic Matrix Inverse with Predictions.

In this section we prove Theorem 4.1 which is our main algebraic data structure. In Section 5 we use this result together with standard reductions from [60, 61, 13] to obtain the graph applications stated in Theorem 5.1.

THEOREM 4.1. *There exists a data structure with the following operations.*

- INITIALIZE *Initialize on given* $\mathbf{M} \in \mathbb{F}^{n \times n}$ *and a queue of* n *rank-1 updates. Complexity* $O(n^\omega)$.

- APPENDUPDATE *Append a rank-1 update (given via two vectors* u, v*) at the end of the queue in* $O(n)$ *worst-case update time.*

- PERFORMUPDATE(η) *Performs the update (i.e.* $\mathbf{M} \leftarrow \mathbf{M} + uv^\top$*) stored at the* η*-th position in the queue, and removes it from the queue. The data structure returns the rank and determinant of* $\mathbf{M}$*. If the matrix is invertible, it also returns the vector* $v^\top \mathbf{M}'^{-1}$ *(where* v *is the vector of the performed rank-1 update and* $\mathbf{M}'$ *is the matrix* $\mathbf{M}$ *from before the update). The worst-case update time is* $O(n^{\omega-1} + \min\{n\eta, n^2\})$*.*

The queue must have at least n *updates at all times. The data structure is randomized and its output is correct with high probability.*

Note that here η describes precisely the prediction error as described in the introduction, i.e. the parameter η describes how much earlier an update occurs than predicted. If all updates occur exactly in the sequence as predicted, we always have $\eta = 1$. If an update occurs η iterations too early, then it is stored at the η-th position in the queue.

REMARK 4.1. *If the matrix is promised to stay invertible throughout all updates, the data structure of Theorem 4.1 can be deterministic.*

4.1 Reducing Rank-1 Updates to Column Updates. We reduce the general rank-1 updates as described in Theorem 4.1 to column updates which are easier to analyze. Here by column update, we mean an update that changes only one column at a time. Such a reduction is not possible in the general setting without predictions. Rank-1 updates without predictions require $\Omega(n^2)$ update time under the OMv hypothesis, but column updates can be performed in $O(n^{1.529})$ time [13]. However, since we are in the offline/prediction settings we can reduce the rank-1 updates to column updates.

The idea is as follows: Given a set of n predicted rank-1 updates $(u_i, v_i)_{i=1,\ldots,n}$, we can construct $\mathbf{U}, \mathbf{V}$ by stacking the vectors next to each other. Then performing the rank-1 updates to some matrix $\mathbf{M}$ could be phrased as follows: Let $\mathbf{D}$ be an initially all-0 matrix and consider $\mathbf{M}' := (\mathbf{M} + \mathbf{UDV}^\top)$. By flipping the diagonal entries of $\mathbf{D}$ from 0 to 1, the matrix $\mathbf{M}'$ is precisely the matrix $\mathbf{M}$ after receiving the rank-1 updates. In particular, if the first k diagonal entries of $\mathbf{D}$ are 1, we have $\mathbf{M}' = \mathbf{M} + \sum_{i=1}^{k} u_i v_i^\top$. Thus a rank-1 update to $\mathbf{M}$ can be seen as a single entry update to $\mathbf{D}$.

Further, it was shown that maintaining the value of any matrix formula $f(\mathbf{M}_1, \ldots, \mathbf{M}_k)$ that consists only of basic matrix operations (addition, subtraction, multiplication, inversion) can be reduced to a single matrix inversion. That is, we can reduce the data structure task of maintaining the value of the formula

$$f(\mathbf{M}, \mathbf{U}, \mathbf{V}^\top, \mathbf{D}) = (\mathbf{M} + \mathbf{UDV}^\top)^{-1}$$

subject to entry updates to $\mathbf{D}$, to a data structure that maintains the inverse of some matrix subject to entry updates (and column updates are just a generalization of entry updates.).

Only if a rank-1 update was not predicted does an entry update to $\mathbf{D}$ not suffice and we must perform an actual rank-1 update to $\mathbf{M}$.

The following reduction is implicit from the following lemma by v.d.Brand [11].

LEMMA 4.1. ([11]) *Given a matrix formula $f(\mathbf{A}_1, \ldots, \mathbf{A}_k)$ consisting of $p \geq k - 1$ matrix operations, there exists a block matrix $\mathbf{B}$ where some blocks are precisely $\mathbf{A}_1, \ldots, \mathbf{A}_k$ and the inverse $\mathbf{B}^{-1}$ contains a block that is $f(\mathbf{A}_1, \ldots, \mathbf{A}_k)$. When each $\mathbf{A}_i$ is at most size $n \times n$, then $\mathbf{N}$ is of size at most $O(pn) \times O(pn)$. The proof is constructive and constructing $\mathbf{N}$ takes time $O((pn)^2)$.*

This reduction from predicted rank-1 updates to column updates motivates the following Lemma 4.2, which can be interpreted as a restriction of Theorem 4.1 to column updates.

LEMMA 4.2. *There exists a data structure with the following operations.*

- INITIALIZE *Initialize on given $\mathbf{M} \in \mathbb{F}^{n \times n}$ and a queue of n rank-1 updates in $O(n^\omega)$ time.*

- APPENDUPDATE *Append a rank-1 update (given via two vectors u, v) at the end of the queue in $O(n)$ worst-case update time.*

- PERFORMUPDATE(η, ISQUERY) *Performs the update (i.e. $\mathbf{M} \leftarrow \mathbf{M} + uv^\top$) stored at the η-th position in the queue, and removes it from the queue. The data structure returns the determinant of $\mathbf{M}$, and $v^\top \mathbf{M}^{-1}$ where v is the vector of the performed rank-1 update.*
  (We can decide to only perform a query, i.e. return these values but do not change $\mathbf{M}$.) Worst-case update time is $O(n^{\omega-1} + \min\{n\eta, n^2\})$ if v was a standard unit-vector, otherwise it is $O(n^2)$.

The queue must have at least n updates at all times. The data structure returns "fail" for the first time $\mathbf{M}$ becomes singular. The data structure can no longer handle any updates after that point.

4.2 Column Updates with Predictions. In this subsection, we prove Lemma 4.2. The main intermediate result is the following Lemma 4.3. At the end of this subsection we prove that Lemma 4.3 implies Lemma 4.2.

LEMMA 4.3. *There is a data structure with the following operations:*

- INITIALIZE *Initialize on given $\mathbf{M} \in \mathbb{F}^{n \times n}$ and non-empty sets $F_0, \ldots, F_{\log n} \subset [n]$ where $|F_i| \leq c \cdot 2^{i+1}$ for $c \geq 1$ and $F_i \subset F_{i+1}$ in $O(n^\omega)$ time. These sets are predictions for the future column updates. Set F_i contains the predicted column indices for the next 2^i column updates.*

- QUERYANDUPDATE *For $u, v \in \mathbb{F}^n$ return $v^\top \mathbf{M}^{-1}$. Optionally, we can decide to set $\mathbf{M} \leftarrow \mathbf{M} + uv^\top$.*

  If v is some standard unit vector e_j and $j \in F_i$ for some i, then this takes $O(cn^{\omega-1} + n2^i)$ time. Otherwise it takes $O(cn^{\omega-1} + n^2)$ time.

  If this is the t-th update, then for all ℓ where 2^ℓ divides t the data structure must also receive new prediction sets $F_1, \ldots, F_\ell$ such that $F_i \subset F_{i+1}$ for all $i = 0, \ldots, \log n$ and $|F_i| \subset c \cdot 2^{i+1}$.

The matrix $\mathbf{M}$ *must stay invertible throughout all updates.*

We briefly discuss why Lemma 4.3 will imply update complexities as stated in Lemma 4.2.

The sets $(F_i)_{i=0,\ldots,\log n}$ are the sets of possible column indices where we predict the future 2^i columns updates to be. For instance, if we are promised a sequence of n column updates with $j_1,\ldots,j_n$ being their column indices (i.e. when given a queue of updates as in Lemma 4.2), then we can pick $F_i = \{j_1,\ldots,j_{2^i}\}$ for $i = 0,\ldots,\log n$. However, note that we do not need such a precise prediction for our data structure to work. It is enough if we have some $O(2^i)$-sized prediction for the future 2^i updates.[4]

Assuming we have the promised sequence of n column updates, but some column update happens η iterations too early (i.e. η as in PERFORMUPDATE in Theorem 4.1), then we can find that column index in some F_k for $k \leqslant \min\{1 + \log \eta, \log n\}$. Thus such an update takes time $O(n^{\omega-1} + n2^k) = O(n^{\omega-1} + \min\{n\eta, n^2\})$ by Lemma 4.3. Note that w.l.o.g. $F_{\log n} = [n]$, so for any column update, we can always assume $\eta \leqslant n$. However, if an update is not a column update (i.e. if we perform a general rank-1 update), the update complexity will be $O(n^2)$ according to Lemma 4.3.

The proof of Lemma 4.3 relies on the following implicit representation of a matrix inverse.

LEMMA 4.4. *Given* $\mathbf{M} \in \mathbb{F}^{n \times n}$ *and a rank-k update* $\mathbf{U}, \mathbf{V} \in \mathbb{F}^{n \times k}$ *we have*

$$(\mathbf{M} + \mathbf{U}\mathbf{V}^\top)^{-1} = \mathbf{M}^{-1}(\mathbf{I} + \mathbf{L}\mathbf{R}),$$

where $\mathbf{R} = \mathbf{V}^\top \mathbf{M}^{-1}$, $\mathbf{L} = \mathbf{U}(\mathbf{I} + \mathbf{U}\mathbf{R})^{-1}$.

Proof. The Woodbury identity [66] states that

$$(\mathbf{M} + \mathbf{U}\mathbf{V}^\top)^{-1} = \mathbf{M}^{-1} - \mathbf{M}^{-1}\mathbf{U}(\mathbf{I} + \mathbf{V}^\top \mathbf{M}^{-1}\mathbf{U})^{-1}\mathbf{V}^\top \mathbf{M}^{-1},$$

which implies

$$(\mathbf{M} + \mathbf{U}\mathbf{V}^\top)^{-1} = \mathbf{M}^{-1}(\mathbf{I} - \mathbf{U}(\mathbf{I} + \mathbf{V}^\top \mathbf{M}^{-1}\mathbf{U})^{-1}\mathbf{V}^\top \mathbf{M}^{-1})$$
$$= \mathbf{M}^{-1}(\mathbf{I} - \mathbf{L}\mathbf{R}).$$

☐

Proof. [Proof of Lemma 4.3]

We start by describing a data structure with *amortized* time complexity and will later extend it to worst-case time.

Let $\mathbf{M}^{(i)}$ initially be the matrix $\mathbf{M}$, but then we update $\mathbf{M}^{(i)} \leftarrow \mathbf{M}$ only every 2^i calls to QUERYANDUPDATE. So we always have $\mathbf{M}^{(0)} = \mathbf{M}$, but $\mathbf{M}^{(i)}$ might be that status of $\mathbf{M}$ some 2^i calls to QUERYANDUPDATE ago. Throughout all updates, we represent the inverses of these matrices in the following implicit form

$$(4.2) \qquad\qquad (\mathbf{M}^{(i)})^{-1} = (\mathbf{M}^{(i+1)})^{-1}(\mathbf{I} + \mathbf{L}^{(i)}\mathbf{R}^{(i)})$$

where $\mathbf{L}^{(i)}, \mathbf{R}^{(i)} \in \mathbb{F}^{n \times 2^i}$. We do this for $i = 0,\ldots,\log n$.

Initialization. We compute $\mathbf{M}^{-1}$ and store this matrix as $\mathbf{M}^{(\log n)}$. Then set $\mathbf{L}^{(i)} = \mathbf{R}^{(i)} = 0$ for all i.

Update. Assume we receive the t-th call to QUERYANDUPDATE where ℓ is the largest integer such that 2^ℓ divides t. We start describing the calculations performed by our data structure. Afterward we will analyze the complexity.

First, we compute $v^\top(\mathbf{M}^{(0)})^{-1}$ where v is one of the vectors describing the rank-1 update given by the current call to QUERYANDUPDATE, as we must return this result. Further, the data structure performs the following operations to update its internal representation of the inverse.

To maintain invariant (4.2) we must update $\mathbf{M}^{(i)}$ for all $i \leqslant \ell$. Note that $\mathbf{M}^{(\ell+1)}$ was last updated 2^ℓ calls to QUERYANDUPDATE ago, so the difference between $\mathbf{M}^{(\ell)}$ and $\mathbf{M}^{(\ell+1)}$ are only the past 2^ℓ updates. By Lemma 4.4, we can choose $\mathbf{L}^{(\ell)}$ and $\mathbf{R}^{(\ell)}$ as follows:

---

[4]This will later be crucial to maintain the rank of $\mathbf{M}$, because the reduction from dynamic matrix rank to dynamic matrix inverse performs adaptive updates that cannot be accurately predicted.

Let $\mathbf{V}$ be the $n \times 2^\ell$ matrix, where the columns are given by the vectors v of the past 2^ℓ calls to QueryAndUpdate. Then set $\mathbf{R}^{(\ell)} = V^\top (\mathbf{M}^{(\ell+1)})^{-1}$.

Let $\mathbf{U}$ be the $n \times 2^\ell$ matrix, where the columns are given by the vectors u of the past 2^ℓ calls to QueryAndUpdate. (Though all u for which we only performed a query but no update, we will set the corresponding column in $\mathbf{U}$ to 0 instead.) Then set $\mathbf{L}^{(\ell)} = \mathbf{U}(\mathbf{I} + \mathbf{U}\mathbf{R})^{-1}$.

Thus by Lemma 4.4 we have

$$(\mathbf{M}^{(\ell)})^{-1} = (\mathbf{M}^{(\ell+1)})^{-1}(\mathbf{I} + \mathbf{L}^{(\ell)}\mathbf{R}^{(\ell)}).$$

For $i < \ell$, we set $\mathbf{L}^{(i)} = \mathbf{R}^{(i)} = 0$ because $\mathbf{M}^{(i)} = \mathbf{M}^{(\ell)}$. In summary, we still satisfy (4.2).

Complexity. For now, let us assume that we only have column updates, i.e. we only change one column of $\mathbf{M}$. In that case we can assume v is a standard unit vector because we are adding $uv^\top = ue_j^\top$ to $\mathbf{M}$ to reflect adding u to the j-th column of $\mathbf{M}$ for some j.

Then computing $\mathbf{V}^\top (\mathbf{M}^{(\ell)})^{-1}$ is equivalent to picking 2^ℓ rows of $(\mathbf{M}^{(\ell)})^{-1}$. Let's assume we can obtain all these rows within some time T_ℓ. Computing $\mathbf{R}^{(\ell)}$ now takes $O(\mathrm{MM}(n, 2^\ell, 2^\ell))$ time. So maintaining the representation (4.2) for this specific ℓ takes $O((T_\ell + \mathrm{MM}(n, 2^\ell, 2^\ell))/2^\ell)$ amortized time.

Now assume we have perfect predictions, i.e. we whenever we update any $\mathbf{M}^{(i)}$, we know precisely which future 2^i rows will be required of $(\mathbf{M}^{(i)})^{-1}$. Then we could precompute these 2^i rows whenever we update $\mathbf{M}^{(i)}$. In that case, $T_i = O(n2^i)$ since we just need to read the precomputed rows from memory.

With this motivation, we always precompute the rows of $(\mathbf{M}^{(\ell)})^{-1}$ with row index in F_ℓ whenever we update the representation (4.2). This costs an additional $O(c\,\mathrm{MM}(n, 2^\ell, 2^\ell))$ that amortizes over the next 2^ℓ updates. (Note that computing rows of $(\mathbf{M}^{(\ell)})^{-1}$ requires the same rows of $(\mathbf{M}^{(\ell+1)})^{-1}$ but by $F_\ell \subset F_{\ell+1}$ these rows of $(\mathbf{M}^{(\ell+1)})^{-1}$ have been precomputed already.)

In summary, if all updates occur as predicted, i.e. all future 2^i updates affect only the columns with index in F_i for all $i = 0, \ldots, \log n$, then the amortized update time would just be

$$O\Big(\sum_{i=0}^{\log n} c\,\mathrm{MM}(n, 2^i, 2^i)/2^i\Big) = O(c\,\mathrm{MM}(n, 2^{\log n}, 2^{\log n})/2^{\log n}) = O(cn^{\omega-1}).$$

Now assume there is some error in our prediction, i.e. the column index j of an updated column is not in F_i for some i. That is an issue since we need the j-th row of $(\mathbf{M}^{(i)})^{-1}$ but that row was not precomputed. To compute this row, we must spend $O(n2^i)$ time (by dimension of $\mathbf{L}^{(i)}, \mathbf{R}^{(i)}$) and must also compute the j-th row of $(\mathbf{M}^{(i+1)})^{-1}$. Thus by recursion, we spend $O(n2^k)$ time, where $k > i$ is the smallest integer such that F_k contains j[5], because then that row of $(\mathbf{M}^{(k)})^{-1}$ was precomputed during a previous update. Thus our update time increases by an additive $O(n2^k)$.

Now let us focus on the case where v is not a standard unit vector, i.e. we perform a general rank-1 update instead of a column update. In that case, computing $v^\top (\mathbf{M}^{(i)})^{-1}$ takes $O(n2^i)$ time plus the time to compute $v^\top (\mathbf{M}^{(i+1)})^{-1}$. This leads to at most $O(n^2)$ time for $v^\top (\mathbf{M}^{(\log n)})^{-1}$.

Note that we must also return $v^\top (\mathbf{M}^{(0)})^{-1}$ after each update. This is subsumed by the $O(n2^k)$ and $O(n^2)$ cost above, depending on whether v is a standard unit vector or not.

Worst-Case. The worst-case bounds are obtained via a standard technique. The idea is as follows. When constructing $(\mathbf{M}^{(\ell)})^{-1}$ (i.e.$(\mathbf{L}^{(\ell)}, \mathbf{R}^{(\ell)})$, we spread the calculations over next $2^{\ell-1}$ calls to QueryAndUpdate. Thus updating (4.2) for a specific ℓ introduces only $O(\mathrm{MM}(n, 2^\ell, 2^\ell)/2^\ell)$ worst-case cost per call to QueryAndUpdate.

Note that this modification requires us to modify the recursion of (4.2) a bit. $(\mathbf{M}^{(\ell)})^{-1}$ is not immediately accessible as we spread its calculation over several updates. So $(\mathbf{M}^{(\ell-1)})^{-1}$ cannot access $(\mathbf{M}^{(\ell)})^{-1}$ yet. So instead, it will access the old variant of $(\mathbf{M}^{(\ell)})^{-1}$ (the one we are currently replacing). This means that $\mathbf{R}^{(\ell-1)}, \mathbf{L}^{(\ell-1)}$ must be larger by a factor of two (2^ℓ columns instead of $2^{\ell-1}$) because the old version of $(\mathbf{M}^{(\ell)})^{-1}$ represents the matrix $\mathbf{M}$ some 2^ℓ updates ago.

This old version of $(\mathbf{M}^{(\ell)})^{-1}$ is accessible when we refresh $(\mathbf{M}^{(\ell-1)})^{-1}$ because its computation was spread over $2^{\ell-1}$ updates.

---

[5]This parameter k can be seen as the error in our prediction, i.e. if an update happens η iterations too early, then we will have $j \in F_k$ for $k < 1 + \log \eta$.

The same is done recursively. For any $i < \ell$, where ℓ is the largest integer that divides t (during the t-th update), we set
$$(\mathbf{M}^{(i)})^{-1} = (\mathbf{M}'^{(i+1)})^{-1}(\mathbf{I} + \mathbf{L}^{(i)}\mathbf{R}^{(i)}),$$
where $\mathbf{M}'^{(i+1)}$ is either the old version of $\mathbf{M}^{(i+1)}$, or the current version of it, if it has finished its calculation.

Since the dimensions of the $\mathbf{L}, \mathbf{R}$ matrices increases only by a constant factor, the time complexity also increases by only a constant factor, but it is now worst-case. □

We can now prove Lemma 4.2 via Lemma 4.3. We restate Lemma 4.2:

LEMMA 4.2. *There exists a data structure with the following operations.*

- *INITIALIZE Initialize on given $\mathbf{M} \in \mathbb{F}^{n \times n}$ and a queue of n rank-1 updates in $O(n^\omega)$ time.*

- *APPENDUPDATE Append a rank-1 update (given via two vectors u, v) at the end of the queue in $O(n)$ worst-case update time.*

- *PERFORMUPDATE(η, ISQUERY) Performs the update (i.e. $\mathbf{M} \leftarrow \mathbf{M} + uv^\top$) stored at the η-th position in the queue, and removes it from the queue. The data structure returns the determinant of $\mathbf{M}$, and $v^\top \mathbf{M}^{-1}$ where v is the vector of the performed rank-1 update.*
  (We can decide to only perform a query, i.e. return these values but do not change $\mathbf{M}$.) Worst-case update time is $O(n^{\omega-1} + \min\{n\eta, n^2\})$ if v was a standard unit-vector, otherwise it is $O(n^2)$.

The queue must have at least n updates at all times. The data structure returns "fail" for the first time $\mathbf{M}$ becomes singular. The data structure can no longer handle any updates after that point.

Proof. [Proof of Lemma 4.2] Let us quickly recap some terminology. If v of the rank-1 update is a standard unit vector, then the rank-1 update is a "column update" i.e. it changes only one column. We now define "the column index of the update", i.e. we assign some index to each update: If the update is a column update, we select the index of the affected column. If the update is not a column update, we just assign 1 as the column index. Thus "the column index of the update" is well-defined regardless of whether the update is a column or general rank-1 update.

Maintaining sets F_i. We have a queue of updates Q. This queue implies the sequence of sets $F_1, \ldots, F_{\log n}$ required by the data structure from Lemma 4.3. That is, at initialization (and every 2^i updates) F_i contains the column-indices of the next 2^i updates for each $i = 0, \ldots, \log n$.

Initialization. We are given the initial matrix $\mathbf{M}$ and a queue of updates Q. This queue implies the sequence of sets $F_1, \ldots, F_{\log n}$ required by the data structure from Lemma 4.3. We initialize Lemma 4.3 on $\mathbf{M}$ and $(F_i)_{i=0,\ldots,\log n}$.

AppendUpdate. We are given a new update to append to the queue. We store this future update in Q.

Update. We are given a queue position η and must perform the update stored at the η-th position in the queue. If the update was a column update, then the column index of the update will be stored in some F_i for $i \leq \min\{1 + \log \eta, \log n\}$. Thus the update takes $O(n^{\omega-1} + \min\{n\eta, n^2\})$ operations. If this update is not a column update, then by Lemma 4.3 it takes $O(n^2)$ operations. The data structure of Lemma 4.3 returns $v^\top \mathbf{M}^{-1}$.

Determinant. We have $\det(\mathbf{M} + uv^\top) = \det(\mathbf{M}) \cdot (1 + v^\top \mathbf{M}^{-1} u)$. Here $v^\top \mathbf{M}^{-1}$ is given to us after each update, so we can maintain the determinant with an extra $O(n)$ overhead per update, which is subsumed by the cost of an update to Lemma 4.3. We must compute $\det(\mathbf{M})$ during initialization which takes $O(n^\omega)$ operation, which is also subsumed by the initialization cost of Lemma 4.3. □

4.3 Putting Everything Together. We now prove Theorem 4.1 using Lemma 4.2 and the reduction from predicted rank-1 to column updates that we outlined in Section 4.2.

Proof. [Proof of Theorem 4.1] Theorem 4.1 is almost the same as Lemma 4.2, except that

- The update time $O(n^{\omega-1} + \min\{n\eta, n^2\})$ of Lemma 4.2 and Theorem 4.1 matches only for column updates. Rank-1 updates are slower in Lemma 4.2 with $O(n^2)$ update time.

- Theorem 4.1 works on singular matrices and can maintain the rank.

We here describe how to extend Lemma 4.2 to obtain Theorem 4.1.

Rank-1 Update Complexity. We here describe how to reduce predicted rank-1 updates to column updates. Using this reduction, any predicted rank-1 update can be performed via a column update, we obtain $O(n^{\omega-1} + \min\{n\eta, n^2\})$ update time for predicted rank-1 updates.

We can phrase the future rank-1 updates to $\mathbf{M}$ as follows: Consider the formula $\mathbf{V}'^\top(\mathbf{M} + \mathbf{UDV}^\top)^{-1}$. Here $\mathbf{U}, \mathbf{V}, \mathbf{V}'$ are the vectors u, v of the n future rank-1 updates, but $\mathbf{V}'$ contains one extra all-0 column, and $\mathbf{D}$ is a diagonal matrix. Initially $\mathbf{D}$ is all-0, and then one by one we set the diagonal entries to 1 to perform the queued rank 1 updates. Using Lemma 4.1, there is a matrix $\mathbf{B}$ that can be used to maintain the value of this formula. Since the formula consists of 5 operations (1 addition, 3 products and 1 inversion), the matrix $\mathbf{B}$ from Lemma 4.1 is of size $O(n) \times O(n)$ so there will be only a constant complexity blow-up.

Any predicted rank-1 update requires us to just change one entry of $\mathbf{B}$ (i.e. the one entry of the block corresponding to $\mathbf{D}$) so it can be performed via a column update to $\mathbf{B}$ in $O(n^{\omega-1} + \min\{n\eta, n^2\})$ time via Theorem 4.1. A rank-1 update not already stored in $\mathbf{U}, \mathbf{V}$ takes $O(n^2)$ time (by Lemma 4.3) because we perform a typical rank-1 update to $\mathbf{M}$ (and thus a rank-1 update to $\mathbf{B}$). After every update, we output one row of $\mathbf{V}'^\top(\mathbf{M} + \mathbf{UDV}^\top)^{-1}$ which is contained in one row of $\mathbf{B}^{-1}$ (and we know which row of $\mathbf{B}$ that is, by Lemma 4.1 being constructive). In case of an update not stored in $\mathbf{V}$ (and thus also not stored in $\mathbf{V}'$), we set the extra 0 column of $\mathbf{V}'$ to the vector v of the rank-1 update. This, too, takes $O(n^2)$ time.

We restart this dynamic algorithm every $n/2$ updates, thus $\mathbf{U}, \mathbf{V}$ will contain all updates that according to the queue should happen within the next $n/2$ iterations. This takes $O(n^{\omega-1})$ amortized time per update and can be made worst-case via standard techniques. By doing these restarts, any updates at position $\eta \leqslant n/2$ within the queue are stored in $\mathbf{U}, \mathbf{V}$ and thus have update time $O(n^{\omega-1} + n\eta)$. For updates at position $\eta > n/2$, they might not be stored in $\mathbf{U}, \mathbf{V}$ and thus are performed in $O(n^2)$ time.

In summary, we have $O(n^{\omega-1} + \min\{n\eta, n^2\})$ update time when performing an update stored at the η-th position within the queue.

Rank. Sankowski [61] showed the following lemma:

LEMMA 4.5. ([61]) *Given* $\mathbf{M} \in \mathbb{F}^{n \times n}$, *let*

$$\mathbf{N} := \begin{bmatrix} \mathbf{M} & \mathbf{X} & \\ \mathbf{Y} & & \mathbf{I} \\ & \mathbf{I} & \mathbf{I}_k \end{bmatrix} \in \mathbb{F}^{3n \times 3n}$$

where $\mathbf{X}, \mathbf{Y}$ *are* $n \times n$ *matrices where each entry is a uniformly at random sampled number from* $\mathbb{F}$. *Matrix* $\mathbf{I}_k$ *is the partial identity where only the first* k *diagonal entries are* 1 *and the remaining entries are* 0. *Then with probability at least* $1 - O(n/|\mathbb{F}|)$ *we have that* $\mathbf{N}$ *is full rank if and only if* $\mathrm{rank}(\mathbf{M}) \geq n - k$. *Further, if* $k = 0$, *then the top-left* $n \times n$ *block of* $\mathbf{N}^{-1}$ *is precisely* $\mathbf{M}^{-1}$.

We can reduce maintaining the rank of $\mathbf{M}$ to maintaining the determinant of $\mathbf{N}$ (as defined in Lemma 4.5): Initially, compute the rank of $\mathbf{M}$ and let $k = n - \mathrm{rank}(\mathbf{M})$. Then run the our data structure on $\mathbf{N}$. With each update to $\mathbf{M}$, the rank can change by at most 1. If we observe that the rank decreases (because the determinant of $\mathbf{N}$ becomes 0), we unroll the last update, increase k, and then perform the original update again. Alternatively, if the determinant did not become 0, we check if the rank increased by decreasing k by 1 (i.e. one extra update to $\mathbf{N}$) and check if the determinant becomes 0 (if so, revert this update again). Note that thus for each update to $\mathbf{M}$, we will perform up to 2 updates to $\mathbf{N}$. Further, if we look some t updates to $\mathbf{M}$ into the future, then the rank can change at most by t, so we have a range of size $O(t)$ where the future updates to $\mathbf{I}_k$ will occur. So while we do not know the exact location of the future updates, we can still construct the sets $(F_i)_{0 \leq i \leq \log n}$ as required by Lemma 4.3: For any $0 \leq i \leq \log n$, let F_i be the column indices of the next 2^i updates to $\mathbf{M}$ and additionally the $2 \cdot 2^i$ column indices $2n + k + j$ for $-2^i \leq j \leq 2^i$ representing the range within which we may change the $\mathbf{I}_k$ block of $\mathbf{N}$.

The failure probability of Lemma 4.5 is $O(n/|\mathbb{F}|)$. For most of our use-cases, we will have $|\mathbb{F}| = \mathrm{poly}(n)$. However, if $|\mathbb{F}|$ is not polynomial size, we can make the failure probability some small n^{-c} for any arbitrary constant $c > 0$ by instead using some field extension $\mathbb{F}'$ of polynomial size. $\quad\square$

5 Fully Dynamic Graph Algorithms with Predictions.

In this section we prove Theorem 1.3. We restate the result here in a more detailed way, i.e., we define the different operations of the data structure and how the predictions are given to it:

THEOREM 5.1. *There exists a fully dynamic algorithm that solves the following problems under vertex updates with predictions: triangle detection, single-source reachability, strong connectivity, directed cycle detection, maximum matching size, number of vertex disjoint st-paths. The operations of the data structure are as follows*

- INITIALIZE *Initialize on the graph and a queue of n predicted vertex updates in $O(n^\omega)$ time.*

- APPENDUPDATE *Append a vertex update at the end of the queue in $O(n)$ worst-case time.*

- PERFORMUPDATE(η) *Performs the update stored at the t-th position in the queue, and removes it from the queue. The worst-case update time is $O(n^{\omega-1} + n\min\{\eta, n\})$.*

The queue must have at least n updates at all times.

REMARK 5.1. *While we state the queue to need at least n updates, any smaller $\Omega(n)$ also works by repeating each update $O(1)$ times.*

Note that η, the position of an update in the queue, as in Theorem 5.1 matches the definition of η being an error measure of the prediction. The queue can be seen as the predicted sequence of updates, and if all predictions are correct, we always perform the first update in queue, i.e. $\eta = 1$. If the prediction is inaccurate and some update occurs η iteration to early, then that update is stored not at the front of the queue, but at position η.

We now prove Theorem 5.1 via several reductions. Each subsection will present one reduction that solves one of the graph problems stated in Theorem 5.1 by reducing it to Theorem 4.1.

5.1 Triangle Detection. Given graph G and its adjacency matrix $\mathbf{A}$, the number of triangles in G is given by $\sum_v (\mathbf{A}^3)_{v,v}/3$, because $(\mathbf{A}^3)_{v,v}$ is the number of paths from v to v using 3 edges, i.e. the number of triangles containing v.

After performing a vertex update to some $v \in V$, let $\mathbf{A}'$ be the old adjacency matrix of G and $\mathbf{A}$ be the new one. Then the number of triangles in G changes by $(\mathbf{A}^3)_{v,v} - (\mathbf{A}'^3)_{v,v}$. Thus we can maintain the number of triangles by querying only 2 entries of $\mathbf{A}^3$ (one before and one after the update).

We can maintain $\mathbf{A}^3$ via a matrix inverse by

$$
\begin{bmatrix} \mathbf{I} & \mathbf{A} & & \\ & \mathbf{I} & \mathbf{A} & \\ & & \mathbf{I} & \mathbf{A} \\ & & & \mathbf{I} \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{I} & -\mathbf{A} & \mathbf{A}^2 & -\mathbf{A}^3 \\ & \mathbf{I} & -\mathbf{A} & \mathbf{A}^2 \\ & & \mathbf{I} & -\mathbf{A} \\ & & & \mathbf{I} \end{bmatrix}
$$

So we can solve triangle detection by running Theorem 4.1 on the $3n \times 3n$ matrix above. Any vertex update to G corresponds to updating one row and one column of $\mathbf{A}$, so can be implemented via $3 \cdot 2 = 6$ rank-1 updates to the matrix above.

5.2 Directed Cycle Detection.

LEMMA 5.1. ([13]) *Let $G = (V, E)$ be a directed graph and $\mathbf{A} \in \mathbb{F}^{n \times n}$ s.t. each $\mathbf{A}_{u,v}$ for $(u,v) \in E$ is picked independently and uniformly at random from $\mathbb{F}$, and all other entries of $\mathbf{A}$ are 0. Then with probability at least $1 - n/|\mathbb{F}|$ we have $\det(\mathbf{I} - \mathbf{A}) = 1$ if and only if G is acyclic.*

We run Theorem 4.1 on matrix $\mathbf{I} - \mathbf{A}$ as in Lemma 5.1. A vertex update to G corresponds to changing one row and column of $\mathbf{I} - \mathbf{A}$ so it can be performed with two rank-1 updates.

5.3 Single Source Reachability and Strong Connectivity.

LEMMA 5.2. ([60]) *Let $G = (V, E)$ be a directed graphs and $\mathbf{A} \in \mathbb{F}^{n \times n}$ s.t. each $\mathbf{A}_{u,v}$ for $(u,v) \in E$ is picked independently and uniformly at random from $\mathbb{F}$, and all other entries of $\mathbf{A}$ are 0. Then for any $s, t \in V$ with probability at least $1 - n/|\mathbb{F}|$ we have $(\mathbf{I} - \mathbf{A})^{-1}_{s,t} \neq 0$ if and only if s can reach t.*

We run Theorem 4.1 on matrix $\mathbf{I} - \mathbf{A}$ as in Lemma 5.2. A vertex update to G corresponds to changing one row and column of $\mathbf{I} - \mathbf{A}$ so it can be performed with two rank-1 updates. To return the single-source reachability for

some source vertex s, we read the s-th row of $(\mathbf{I} - \mathbf{A})^{-1}$. This can be obtained by using Theorem 4.1 to query $e_s^\top (\mathbf{I} - \mathbf{A})^{-1}$.

To solve strong connectivity, not that a graph is strongly connected if and only if for any one vertex v, every other vertex can reach v and v can reach every other vertex. Thus we can solve strong connectivity by running two data structures for single source reachability.

5.4 Maximum Matching Size and Counting st-Paths.

LEMMA 5.3. ([46]) *Given graph $G = (V, E)$ let $\mathbf{A} \in \mathbb{F}^{n \times n}$ be the randomized Tutte matrix. That is, for each $(u, v) \in E$ let $\mathbf{A}_{u,v} = -\mathbf{A}_{v,u}$ be picked independently and uniformly at random from $\mathbb{F}$. Then for $|\mathbb{F}| = \mathbb{Z}_p$ ($p = \mathrm{poly}(n)$) we have w.h.p. $\mathrm{rank}(\mathbf{A}) = 2 \cdot$ maximum matching size.*

We run Theorem 4.1 on the Tutte matrix $\mathbf{A}$ as in Lemma 5.3. A vertex update to G corresponds to changing one row and column of $\mathbf{A}$ so it can be performed with two rank-1 updates.

Counting the number of vertex disjoint st-paths can be solved via standard reduction to maximum bipartite matching size, see e.g. [51].

6 Fully Dynamic Algorithms with Predicted Deletion Times.

We consider the model in which insertions are arriving online but deletions are based on a predicted sequence, which we refer to as semi-online with prediction setting. We can extend the reduction of [54] that gives a reduction from a fully dynamic *semi-online* data structure, in which the sequence of deletions are offline, to an *insert-only* data structure. In particular, assuming that the insert only data structure has *worst-case* update time Γ, the semi-online data structure of [54] has update time $O(\Gamma \log T)$ for a sequence of T updates. We observe that an adaptation of their result can be used for the predicted deletion model.

We start by sketching their amortized semi-online to worst-case insert-only reduction and then explain how this algorithm can be adapted to handle a deletion with error η_i. Specifically in the rest of this section we argue that the following theorem holds[6]:

THEOREM 6.1. *Consider a sequence of T updates and suppose we are given an incremental dynamic algorithm with worst-case update time Γ. Assume also that at any point in time we have a prediction on the order of deletions of current items, such that for the i inserted item the error η_i indicates the number of elements predicted to be earlier than i-th item that actually arrive later ($\eta_i = 0$ if the prediction is correct or the element arrives later). Then we have a fully-dynamic algorithm with the following guarantees:*

- *An insertion update is performed in $O(\Gamma \log^2 T)$ worst-cast time.*

- *The deletion of the i-th element can be performed in $O((1 + \eta_i) \cdot \Gamma \log^2 T)$ worst-case time.*

The key idea of the reduction in [54] is to order the list of elements in the reverse of deletion at any time and then perform each deletion by rewinding the computation (undo the insertion) in time $O(\Gamma)$ for the first element in this reversed list. Since re-ordering the elements at each update is expensive, they perform an amortization that performs the re-ordering partially for a set of $O(2^j)$ elements in every 2^j updates for each $j = 0, \ldots, \lceil \log T \rceil$. In particular, they keep the elements in $L = \lceil \log T \rceil + 1$ buckets $B_0, B_1, \ldots, B_{\lceil \log T \rceil}$ such that B_j contains the elements indexed in range $[2^j, 2^{j+1})$ in the reverse order of deletion, and hence B_0 contains the next deletion to be performed. At a high-level, for each $j = 0, \ldots, L$, once in every 2^j updates, the algorithm re-orders a set of $O(2^j)$ elements (in $B_0 \cup \cdots \cup B_j$, whose total size is a sum over geometric-sized buckets) and rewinds the computation on these sets in time $O(\Gamma \cdot 2^j)$. The algorithm ensures that B contains the first deletion that needs to be performed. Hence for each $j = 0, \ldots, L$ we get an amortized update time of $O(\Gamma)$. This amortization scheme is similar to the one we use in Section 5, with the difference that here the reverse ordered deletions in buckets are utilized and deletion is performed by a rewind (undo insertion) operation, which also takes $O(\Gamma)$ time in the RAM model.

We get a similar reduction in the predicted deletion setting using the following adaptation: when the update (deletion) of the i-th element e arrives $\eta := \eta_i$ positions *earlier* than predicted, we rewind the computation over all

---

[6] Note that this claim holds for the problems in which the order in which elements are added does not impact the state of the problem – which holds for almost all graph problems studied in the dynamic algorithms literature.

the sequence of elements in these η positions until we get to the correct position of this element in time $O(\eta\Gamma)$ and then re-insert the $\eta - 1$ deleted elements and update the lists as we would with any other insertion. In other words, we can simply maintain the state of the algorithm of [54] by performing $O(\eta)$ rewind operations and $O(\eta)$ re-insert operations, each of which takes $O(\eta\Gamma)$ time.

Note that if an element e arrives later than predicted in the sequence, we simply ignore the deletion e at the predicted arrival time and process it the actual arrival time. This will result in the next elements in the sequence being shifted earlier, and the cost will be incurred to the next elements in the sequence, so that when the element arrives the ordering is corrected by those earlier elements. We can then run the algorithm of [54], and note that the following invariants proven in [54] remain unchanged.

For any time $t \in [T]$, let $\kappa(t)$ denote the largest integer such that t is a multiple of $2^{\kappa(t)}$, let E_t be the set of elements after the t-th update and let $E_{t,r}$ be the set of r elements that are deleted first (if there are less than r elements, we set $E_{t,r} = E_t$). Their algorithm maintains the following invariants:

LEMMA 6.1. ([54]) *After t updates the algorithm maintains:*

- *For each $j = 0, \ldots, \kappa(t)$ we have $E_{t,2^{j+1}} \subseteq B_0 \cup \cdots \cup B_j$;*

- $|B_0 \cup \cdots \cup B_{\kappa(t)+1}| \leqslant O(2^{\kappa(t)})$.

Hence, after performing the $O(\eta\Gamma)$ rewinds and re-insertion we can use Lemma 6.1 to get the amortized update time over T updates. At each time $t \in [T]$ we need to process the union of buckets $B_0 \cup .. \cup B_{\kappa(t)}$, and by summing over the bucket sizes we have:

$$\frac{1}{T}\sum_{t=1}^{T} O(2^{\kappa(t)}\Gamma + \eta\Gamma) = O\left(\left(\frac{1}{T}\sum_{\kappa(t)=0}^{L} 2^{\kappa(t)} \cdot \frac{T}{2^{\kappa(t)}}\right)\Gamma + \eta L\Gamma\right) = O((\eta+1)\Gamma\log T)$$

This algorithm is then de-amortized to get a worst-case bound with an additional log factor. At a high-level, the goal is to slowly perform the longer sequence of re-ordering and re-inserting operation over the updates. The challenge is that the upcoming insertions interleave the scheduled re-orderings. To handle this they distribute and preprocess the re-ordering tasks to $O(\log T)$ threads that further divide the updates into smaller geometrically decreasing sized *epochs*. We refer the readers to [54] for further details on the de-amortization details. They show that we can perform each update (without error) in worst-case $O(\Gamma\log^2 T)$ time. Our adaptation adds $O(\eta)$ rewind operations that take $O(\eta\Gamma)$ time, and introduces additional $O(\eta)$ insertions, which also take in total $O(\Gamma\log^2 T)$ time, and thus the worst-case time is $O(\Gamma\log^2 T(\eta+1))$.

6.1 Application to All-Pairs Shortest Paths with Vertex Updates. We first observe that we can use the reduction of [54], combined with an incremental APSP algorithm based on Floyd-Warshall, first observed by Thorup [63], to get a fully dynamic semi-online APSP algorithm with $O(n^2\log^2 n)$ worst-case update time. Note that here we can bound $T \leq n$ since at any point we can restrict our attention to a set of at most n vertices inserted.

OBSERVATION 6.1. ([63]) *Given an edge-weighted directed graph undergoing online vertex insertions, there is a deterministic algorithm that maintains exact all-pairs shortest paths in this graph with $O(n^2)$ worst-case update time.*

COROLLARY 6.1. ([54, 63]) *Given an edge-weighted directed graph undergoing online vertex insertions and offline (known) vertex deletions, there is a deterministic algorithm that maintains exact all-pairs shortest paths in this graph with $O(n^2\log^2 n)$ worst-case update time.*

We can extend this to the setting in which the insertions are fully online, but the deletions are predicted, with the error measure described using Theorem 1.4.

THEOREM 6.2. *Given a weighted and directed graph undergoing online vertex insertions and predicted vertex deletions, we can maintain exact weighted all-pairs shortest paths with the following guarantees:*

- *An insertion update can be performed $O(n^2\log^2 n)$ worst-cast time.*

- *A deletion of the i-th inserted vertex v_i can be performed in $O(n^2(\eta_i \log^2 n + 1))$ worst-case time, where error $\eta_i \in [0, n]$ indicates how many vertices were predicted to be deleted before v_i that are actually deleted after v_i.*

References

[1] Amir Abboud. Personal communication, 2022.

[2] Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014*, pages 434–443. IEEE Computer Society, 2014. `doi:10.1109/FOCS.2014.53`.

[3] Ittai Abraham, Shiri Chechik, and Sebastian Krinninger. Fully dynamic all-pairs shortest paths with worst-case update-time revisited. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 440–452. SIAM, 2017.

[4] Antonios Antoniadis, Christian Coester, Marek Eliáš, Adam Polak, and Bertrand Simon. Online metric algorithms with untrusted predictions. *ACM Trans. Algorithms*, 19(2):19:1–19:34, 2023. `doi:10.1145/3582689`.

[5] Yossi Azar, Stefano Leonardi, and Noam Touitou. Flow time scheduling with uncertain processing time. In *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 1070–1080. ACM, 2021. `doi:10.1145/3406325.3451023`.

[6] Yossi Azar, Stefano Leonardi, and Noam Touitou. Distortion-oblivious algorithms for minimizing flow time. In *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, SODA 2022*, pages 252–274. SIAM, 2022. `doi:10.1137/1.9781611977073.13`.

[7] Yossi Azar, Debmalya Panigrahi, and Noam Touitou. Online graph algorithms with predictions. In *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, SODA 2022*, pages 35–66. SIAM, 2022. `doi:10.1137/1.9781611977073.3`.

[8] Étienne Bamas, Andreas Maggiori, Lars Rohwedder, and Ola Svensson. Learning augmented energy minimization via speed scaling. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020*, 2020. URL: `https://proceedings.neurips.cc/paper/2020/hash/af94ed0d6f5acc95f97170e3685f16c0-Abstract.html`.

[9] Surender Baswana, Ramesh Hariharan, and Sandeep Sen. Improved decremental algorithms for maintaining transitive closure and all-pairs shortest paths. *Journal of Algorithms*, 62(2):74–92, 2007. Announced at STOC 2002. `doi:10.1016/j.jalgor.2004.08.004`.

[10] Aaron Bernstein. Maintaining shortest paths under deletions in weighted directed graphs: [extended abstract]. In *Symposium on Theory of Computing Conference, STOC'13*, pages 725–734. ACM, 2013. `doi:10.1145/2488608.2488701`.

[11] Jan van den Brand. Unifying matrix data structures: Simplifying and speeding up iterative algorithms. In *SOSA*, pages 1–13. SIAM, 2021.

[12] Jan van den Brand, Yin Tat Lee, Danupon Nanongkai, Richard Peng, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Bipartite matching in nearly-linear time on moderately dense graphs. In *FOCS*, pages 919–930. IEEE, 2020.

[13] Jan van den Brand, Danupon Nanongkai, and Thatchaphol Saranurak. Dynamic matrix inverse: Improved algorithms and matching conditional lower bounds. In *FOCS*, pages 456–480. IEEE Computer Society, 2019.

[14] Karl Bringmann, Marvin Künnemann, and André Nusser. Fréchet distance under translation: Conditional hardness and an algorithm via offline dynamic grid reachability. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2019)*, pages 2902–2921. SIAM, 2019. `doi:10.1137/1.9781611975482.180`.

[15] Timothy M Chan. Three problems about dynamic convex hulls. In *Proceedings of the twenty-seventh annual symposium on Computational geometry*, pages 27–36, 2011.

[16] Shiri Chechik and Tianyi Zhang. Faster deterministic worst-case fully dynamic all-pairs shortest paths via decremental hop-restricted shortest paths. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 87–99. SIAM, 2023.

[17] Justin Y. Chen, Sandeep Silwal, Ali Vakilian, and Fred Zhang. Faster fundamental graph algorithms via learned predictions. In *International Conference on Machine Learning, ICML 2022*, volume 162 of *Proceedings of Machine Learning Research*, pages 3583–3602. PMLR, 2022. URL: `https://proceedings.mlr.press/v162/chen22v.html`.

[18] Li Chen, Gramoz Goranci, Monika Henzinger, Richard Peng, and Thatchaphol Saranurak. Fast dynamic cuts, distances and effective resistances via vertex sparsifiers. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020*, pages 1135–1146. IEEE, 2020. `doi:10.1109/FOCS46700.2020.00109`.

[19] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. In *FOCS*, pages 612–623. IEEE, 2022.

[20] Camil Demetrescu and Giuseppe F. Italiano. Fully dynamic transitive closure: Breaking through the o(n^2) barrier. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS 2000)*, pages 381–389, 2000. `doi:10.1109/SFCS.2000.892126`.

[21] Camil Demetrescu and Giuseppe F. Italiano. Fully dynamic all pairs shortest paths with real edge weights. *Journal of Computer and System Sciences*, 72(5):813–837, 2006. Announced at FOCS 2001. `doi:10.1016/j.jcss.2005.05.005`.

[22] Michael Dinitz, Sungjin Im, Thomas Lavastida, Benjamin Moseley, and Sergei Vassilvitskii. Faster matchings via learned duals. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021*, pages 10393–10406, 2021. URL: `https://proceedings.neurips.cc/paper/2021/hash/5616060fb8ae85d93f334e7267307664-Abstract.html`.

[23] Ran Duan and Seth Pettie. Fast algorithms for (max, min)-matrix multiplication and bottleneck shortest paths. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009*, pages 384–391. SIAM, 2009. `doi:10.1137/1.9781611973068.43`.

[24] Ran Duan, Hongxun Wu, and Renfei Zhou. Faster matrix multiplication via asymmetric hashing. In *64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023*. IEEE, 2023.

[25] David Eppstein. Offline algorithms for dynamic minimum spanning tree problems. *J. Algorithms*, 17(2):237–250, 1994. `doi:10.1006/jagm.1994.1033`.

[26] Jon C. Ergun, Zhili Feng, Sandeep Silwal, David P. Woodruff, and Samson Zhou. Learning-augmented k-means clustering. In *The Tenth International Conference on Learning Representations, ICLR 2022*. OpenReview.net, 2022. URL: `https://openreview.net/forum?id=X8cLTHexYyY`.

[27] Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. On the performance of learned data structures. *Theor. Comput. Sci.*, 871:107–120, 2021. `doi:10.1016/j.tcs.2021.04.015`.

[28] Buddhima Gamlath, Silvio Lattanzi, Ashkan Norouzi-Fard, and Ola Svensson. Approximate cluster recovery from noisy labels. In *Conference on Learning Theory*, volume 178 of *Proceedings of Machine Learning Research*, pages 1463–1509. PMLR, 2022. URL: `https://proceedings.mlr.press/v178/gamlath22a.html`.

[29] Maximilian Probst Gutenberg and Christian Wulff-Nilsen. Fully-dynamic all-pairs shortest paths: Improved worst-case time and space bounds. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2562–2574. SIAM, 2020.

[30] David P. Helmbold and Manfred K. Warmuth. Learning permutations with exponential weights. *J. Mach. Learn. Res.*, 10:1705–1736, 2009. `doi:10.5555/1577069.1755841`.

[31] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015*, pages 21–30. ACM, 2015. `doi:10.1145/2746539.2746609`.

[32] Monika Henzinger, Andrea Lincoln, Barna Saha, Martin P Seybold, and Christopher Ye. On the complexity of algorithms with predictions for dynamic graph problems. *arXiv preprint arXiv:2307.16771*, 2023.

[33] Petter Holme and Jari Saramäki. Temporal networks. *Physics reports*, 519(3):97–125, 2012.

[34] Toshihide Ibaraki and Naoki Katoh. On-line computation of transitive closures of graphs. *Information Processing Letters*, 16(2):95–97, 1983. `doi:10.1016/0020-0190(83)90033-9`.

[35] Giuseppe F. Italiano. Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science*, 48(3):273–281, 1986. `doi:10.1016/0304-3975(86)90098-8`.

[36] Zhihao Jiang, Debmalya Panigrahi, and Kevin Sun. Online algorithms for weighted paging with predictions. *ACM Trans. Algorithms*, 18(4):39:1–39:27, 2022. `doi:10.1145/3548774`.

[37] Adam Karczmarz and Jakub Lacki. Fast and simple connectivity in graph timelines. In *Algorithms and Data Structures - 14th International Symposium, WADS 2015*, volume 9214 of *Lecture Notes in Computer Science*, pages 458–469. Springer, 2015. `doi:10.1007/978-3-319-21840-3\_38`.

[38] Telikepalli Kavitha. Dynamic matrix rank with partial lookahead. *Theory Comput. Syst.*, 55(1):229–249, 2014.

[39] Misha Khodak, Maria-Florina Balcan, Ameet Talwalkar, and Sergei Vassilvitskii. Learning predictions for algorithms with predictions. In *NeurIPS*, 2022. URL: `http://papers.nips.cc/paper_files/paper/2022/hash/17061a94c3c7fda5fa24bbdd1832fa99-Abstract-Conference.html`.

[40] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018*, pages 489–504. ACM, 2018. `doi:10.1145/3183713.3196909`.

[41] Jakub Lacki. Improved deterministic algorithms for decremental reachability and strongly connected components. *ACM Transactions on Algorithms*, 9(3):27:1–27:15, 2013. Announced at SODA 2011. `doi:10.1145/2483699.2483707`.

[42] Jakub Lacki and Piotr Sankowski. Reachability in graph timelines. In *Proceedings of the 4th conference on Innovations in Theoretical Computer Science*, pages 257–268, 2013.

[43] Honghao Lin, Tian Luo, and David P. Woodruff. Learning augmented binary search trees. In *International Conference on Machine Learning, ICML 2022*, volume 162 of *Proceedings of Machine Learning Research*, pages 13431–13440.

PMLR, 2022. URL: https://proceedings.mlr.press/v162/lin22f.html.

[44] Alexander Lindermayr and Nicole Megow. Algorithms with predictions. https://algorithms-with-predictions.github.io, 2022. Accessed 8 July 2023.

[45] Quanquan C Liu and Vaidehi Srinivas. The predicted-deletion dynamic model: Taking advantage of ml predictions, for free. *arXiv preprint arXiv:2307.08890*, 2023.

[46] László Lovász. On determinants, matchings, and random algorithms. In *FCT*, pages 565–574. Akademie-Verlag, Berlin, 1979.

[47] Thodoris Lykouris and Sergei Vassilvitskii. Competitive caching with machine learned advice. *J. ACM*, 68(4):24:1–24:25, 2021. doi:10.1145/3447579.

[48] Xiao Mao. Fully-dynamic all-pairs shortest paths: Likely optimal worst-case update time. *arXiv preprint arXiv:2306.02662*, 2023.

[49] Michael Mitzenmacher and Sergei Vassilvitskii. Algorithms with predictions. In Tim Roughgarden, editor, *Beyond the Worst-Case Analysis of Algorithms*, pages 646–662. Cambridge University Press, 2020. doi:10.1017/9781108637435.037.

[50] Michael Mitzenmacher and Sergei Vassilvitskii. Algorithms with predictions. *Commun. ACM*, 65(7):33–35, 2022. doi:10.1145/3528087.

[51] Ketan Mulmuley, Umesh V. Vazirani, and Vijay V. Vazirani. Matching is as easy as matrix inversion. In *STOC*, pages 345–354. ACM, 1987.

[52] Thy Dinh Nguyen, Anamay Chaturvedi, and Huy L. Nguyen. Improved learning-augmented algorithms for k-means and k-medians clustering. In *The Eleventh International Conference on Learning Representations, ICLR 2023*. OpenReview.net, 2023. URL: https://openreview.net/pdf?id=dCSFiAl_VO3.

[53] Mihai Pătraşcu. Towards polynomial lower bounds for dynamic problems. In Leonard J. Schulman, editor, *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010*, pages 603–610. ACM, 2010. doi:10.1145/1806689.1806772.

[54] Binghui Peng and Aviad Rubinstein. Fully-dynamic-to-incremental reductions with known deletion order (eg sliding window). In *Symposium on Simplicity in Algorithms (SOSA)*, pages 261–271. SIAM, 2023.

[55] Richard Peng, Bryce Sandlund, and Daniel Dominic Sleator. Optimal offline dynamic 2, 3-edge/vertex connectivity. In *Algorithms and Data Structures - 16th International Symposium, WADS 2019*, volume 11646 of *Lecture Notes in Computer Science*, pages 553–565. Springer, 2019. doi:10.1007/978-3-030-24766-9_40.

[56] Johannes A. La Poutré and Jan van Leeuwen. Maintenance of transitive closures and transitive reductions of graphs. In *Proceedings of the International Workshop on Graph-Theoretic Concepts in Computer Science Graph-Theoretic Concepts in Computer Science (WG 1987)*, volume 314, pages 106–120, 1987. doi:10.1007/3-540-19422-3_9.

[57] Manish Purohit, Zoya Svitkina, and Ravi Kumar. Improving online algorithms via ML predictions. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018*, pages 9684–9693, 2018. URL: https://proceedings.neurips.cc/paper/2018/hash/73a427badebe0e32caa2e1fc7530b7f3-Abstract.html.

[58] Liam Roditty and Uri Zwick. Improved dynamic reachability algorithms for directed graphs. *SIAM Journal on Computing*, 37(5):1455–1471, 2008. Announced at FOCS 2002. doi:10.1137/060650271.

[59] Liam Roditty and Uri Zwick. On dynamic shortest paths problems. *Algorithmica*, 61(2):389–401, 2011. Announced at ESA 2004. doi:10.1007/s00453-010-9401-5.

[60] Piotr Sankowski. Dynamic transitive closure via dynamic matrix inverse (extended abstract). In *FOCS*, pages 509–517. IEEE Computer Society, 2004.

[61] Piotr Sankowski. Faster dynamic matchings and vertex connectivity. In *SODA*, pages 118–126. SIAM, 2007.

[62] Piotr Sankowski and Marcin Mucha. Fast dynamic transitive closure with lookahead. *Algorithmica*, 56(2):180–197, 2010. doi:10.1007/s00453-008-9166-2.

[63] Mikkel Thorup. Worst-case update times for fully-dynamic all-pairs shortest paths. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '05, page 112–119, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1060590.1060607.

[64] Virginia Vassilevska, Ryan Williams, and Raphael Yuster. All-pairs bottleneck paths for general graphs in truly sub-cubic time. In *Proceedings of the 39th Annual ACM Symposium on Theory of Computing*, pages 585–589. ACM, 2007. doi:10.1145/1250790.1250876.

[65] Virginia Vassilevska Williams and R. Ryan Williams. Subcubic equivalences between path, matrix, and triangle problems. *Journal of the ACM*, 65(5):27:1–27:38, 2018. Announced at FOCS 2010. doi:10.1145/3186893.

[66] Max A Woodbury. *Inverting modified matrices.* Statistical Research Group, 1950.