Proceedings of Machine Learning Research vol 247:1–60, 2024 37th Annual Conference on Learning Theory

The Predicted-Updates Dynamic Model: Offline, Incremental, and Decremental to Fully Dynamic Transformations

Quanquan C. Liu QUANQUAN.LIU@YALE.EDU
Yale University, New Haven, CT, USA

Vaidehi Srinivas VAIDEHI@U.NORTHWESTERN.EDU
Northwestern University, Evanston, IL, USA[*]

Editors: Shipra Agrawal and Aaron Roth

Abstract

The main bottleneck in designing efficient dynamic algorithms is the unknown nature of the update sequence. In particular, there are problems where the separation in runtime between the best offline or partially dynamic solutions and the best fully dynamic solutions is polynomial, sometimes even exponential. In this paper, we formulate the *predicted-updates dynamic model*, one of the first *beyond-worst-case* models for dynamic algorithms, which generalizes a large set of well-studied dynamic models including the offline dynamic, incremental, and decremental models to the fully dynamic setting when given predictions about the update times of the elements. Our paper models real world settings, in which we often have access to side information that allows us to make coarse predictions about future updates. We formulate a framework that bridges the gap between fully and offline/partially dynamic, leading to greatly improved runtime bounds over the state-of-the-art dynamic algorithms for a variety of important problems such as triconnectivity, planar digraph all pairs shortest paths, k-edge connectivity, and others, for prediction error of reasonable magnitude. Our simple framework avoids heavy machinery, potentially leading to a new set of dynamic algorithms that are implementable in practice.

Keywords: dynamic algorithms, algorithms with predictions, learning-augmented algorithms, beyond worst case analysis

1. Introduction

Learning-augmented algorithms and traditional dynamic algorithms have fundamentally similar goals: to efficiently provide up-to-date solutions to problems in the face of rapidly changing data. While learning-augmented algorithms typically use machine learning methods to take advantage of structure in data, traditional dynamic algorithms consider worst-case adversarial inputs, and algorithms are designed to be efficient in spite of such worst-case, potentially unstructured, instances. In this paper, we combine these two paradigms under ***one framework*** that encompasses the three major settings traditionally studied within dynamic algorithms: the *offline dynamic*, *incremental*, and *decremental* settings. Specifically, we show that we can *transform* broad classes of offline, incremental, and decremental algorithms into *fully dynamic* algorithms provided predictions of the update times of dynamic events such as insertions and/or deletions. Not only does this allow us to achieve (sometimes exponential) improvements in runtime for certain problems, but it also allows us to transfer algorithmic techniques between models in novel ways.

Algorithms with predictions or *learning-augmented algorithms* is a very rich area (see Mitzenmacher and Vassilvitskii (2021) and references therein) that in recent times has seen many interesting and important results. In this paradigm, an algorithm solicits *predictions* from an untrusted

[*] Equal contribution

source (e.g. a machine learning model) to help make decisions. The goal is to design algorithms that achieve the following three desiderata (Lykouris and Vassilvitskii (2021)): *Consistency* ensures that if the predictions are of high quality, the algorithm performs much better than a worst-case algorithm. *Competitiveness* states that if the predictions are of low quality, the algorithm does not perform any worse than a worst-case algorithm. And finally, *robustness* guarantees that the performance of the algorithm degrades gracefully as a function of the prediction error. Additionally, we want to solicit predictions that can be reasonably obtained in practice. An example is edge insertion/deletion predictions for dynamic graphs which is gaining great interest in the machine learning community, with a set of promising results including Kumar et al. (2019); Nguyen et al. (2018); Poursafaei et al. (2022); Quach et al. (2021); Rossi et al. (2020); Zhang and Chen (2018); Wang et al. (2021).

These desiderata present a challenge. While we would like to achieve all three for some reasonable notion of prediction, it is not a priori clear that such a guarantee is even possible. In *online algorithms with predictions*, for example, it is often not possible to achieve all three simultaneously (see Kumar et al. (2018); Gollapudi and Panigrahi (2019); Wei and Zhang (2020)). Even when it is possible to achieve all three desiderata, solutions can often require large and problem specific predictions. One such example is the setting of *warm starts*, where we design algorithms for static problems that take advantage of (large) predicted solutions (see Dinitz et al. (2021); Chen et al. (2022a); Davies et al. (2023)). We provide more detailed comparisons with our work in Appendix G.

Thus, it is particularly interesting that for dynamic algorithms, it is indeed possible to achieve the best of online algorithms and warm starts: we only need predictions of future events (instead of future solutions) and we *simultaneously achieve all three desiderata* of algorithms with predictions. Our model has the added benefit that the requested predictions are not problem-specific, and the same predictions could be used multiple times to solve different unrelated problems (e.g. one set of predictions about a dynamic graph can be used to solve a collection of graph problems).

Our framework expands upon the traditional model used for dynamically changing data. The main bottleneck of designing traditional *dynamic algorithms* is the unknown nature of the update sequence. This is demonstrated by the large gap in efficiency between algorithms for the offline and partially dynamic settings and the fully dynamic (online) setting for many graph problems (see e.g. Abraham et al. (2017); Chen et al. (2020); Goranci (2019); Holm and Rotenberg (2020) and references therein). To address this, there has been much interest in using the techniques from the offline and partially dynamic settings to obtain fully dynamic algorithms for various problems (see Chan (2011); Peng and Rubinstein (2023)), However, prior to our work, no general framework for lifting offline dynamic to the fully dynamic settings were known. We demonstrate the first such framework using a novel data structure called the *random partition-tree* that leverages the power of predictions. One can intuitively and informally think of the random partition-tree as a type of randomized segment tree; hence, we reduce fully dynamic operations to operations on an offline, randomized segtree with a rebuild operation for fixing errors dynamically.

In this paper, we define the *predicted-updates model* to bring *beyond-worst-case analysis* Roughgarden (2021) to the study of dynamic algorithms. In the offline dynamic setting, the entire update sequence is provided to the algorithm at once, and the algorithm's task is to compute all of the solutions corresponding to each update in the sequence. In the incremental setting, an algorithm only needs to handle element insertions and not deletions. Similarly, in the decremental setting, an algorithm only needs to handle element deletions. Our framework can convert an offline dynamic algorithm into a fully-dynamic algorithm using predicted timestamps for all types of updates. Our framework can also be extended to convert an incremental algorithm into a fully-dynamic algorithm, with only predictions of deletion events, and a decremental algorithm into a fully-dynamic algorithm, with only predictions of insertion events. Our framework provides an entire new set of algorithms, previously not known, for the offline and decremental "0 error" setting and generalizes the the known-deletions result of Peng and Rubinstein (2023) for incremental algorithms and concurrent work by van den Brand et al. (2023).

Much of the study of dynamic algorithms has, thus far, focused on worst-case analysis with many fundamental problems hitting boundaries in efficiency. State-of-the-art dynamic solutions are often quite complex and require the use of heavy machinery. The framework we introduce in this paper is fundamentally *simple*, *does not* use heavy machinery, but is applicable to a *broad* range of problems and settings. With the help of predictions, we are able to lift *simpler*, more *implementable*, and *faster* algorithms from the offline and partially dynamic settings to the fully dynamic setting, hence leveraging the power of modern ML predictions to solve previously believed difficult problems.

Problem	Best Fully Dynamic Runtimes		New Predicted-Update Runtimes (Theorems 27 to 29)	
Planar Digraph APSP	$\widetilde{O}\left(n^{2/3}\right)$	Fakcharoenphol and Rao (2006) Klein (2005)	$\widetilde{O}(\sqrt{n})$	Das et al. (2022)
Triconnectivity	$O(n^{2/3})$	Galil et al. (1999)	$\widetilde{O}\left(1\right)$	Holm and Rotenberg (2020) Peng et al. (2017)
k-Edge Connectivity	$n^{o(1)}$	Jin and Sun (2022)	$\widetilde{O}(1)$	Chalermsook et al. (2021)
Dynamic DFS Tree	$\widetilde{O}\left(\sqrt{mn}\right)$	Baswana et al. (2019)	$\widetilde{O}\left(n\right)$	Baswana et al. (2019) Chen et al. (2018)
APSP	$\left(\frac{256}{k^2}\right)^{4/k}$-Approx $\widetilde{O}\left(n^k\right)$ update $\widetilde{O}(n^{k/8})$ query	Forster et al. (2023)	$(2r-1)^k$-Approx $\widetilde{O}\left(m^{1/(k+1)}n^{k/r}\right)$	Chen et al. (2020)
AP Maxflow/Mincut	$O(\log(n)\log\log n)$-Approx $\widetilde{O}\left(n^{2/3+o(1)}\right)$	Chen et al. (2020)	$O\left(\log^{8k}(n)\right)$-Approx. $\widetilde{O}\left(n^{2/(k+1)}\right)$	Goranci (2019) Goranci et al. (2019)
MCF	$(1+\varepsilon)$-Approx $\widetilde{O}(1)$ update $\widetilde{O}(n)$ query	Chen et al. (2020)	$O(\log^{8k}(n))$-Approx. $\widetilde{O}\left(n^{2/(k+1)}\right)$ update $\widetilde{O}(P^2)$ query	Goranci (2019) Goranci et al. (2019)
Uniform Sparsest Cut	$2^{O(\log^{5/6}(n))}$-Approx $2^{O(\log^{5/6}(n))}$ update $O(\log^{1/6}(n))$ query	Goranci et al. (2021)	$O\left(\log^{8k}(n)\right)$-Approx $\widetilde{O}\left(n^{2/(k+1)}\right)$ $O(1)$ query	Goranci (2019) Goranci et al. (2019)
Submodular Max	$1/4$-Approx $\widetilde{O}(k^2)$	Dütting et al. (2023)	0.3178-Approx $\widetilde{O}\left(\text{poly}(k)\right)$	Feldman et al. (2022)

Table 1: Table of the best fully dynamic update runtimes for a variety of problems vs. our update times obtained via our framework assuming $||\text{error}||_1 = \widetilde{O}(T)$. $||\text{error}||_1$ is the sum of the absolute value difference between predicted and real timestamps. Our query times match the fully dynamic query times for every problem. The acronyms are as follows: APSP: all-pairs shortest paths, DFS: depth-first search, AP: all-pairs, MCF: multi-commodity concurrent flows. The variable P is the number of queried pairs in the multi-commodity flow result. If only one runtime is shown, the same runtime holds for both update and query. Problem definitions are provided in Appendix F. Here, m is the maximum number of edges in the graph at any time.

Our framework gives polynomial, sometimes even exponential, improvements in runtime over the best fully dynamic algorithm for APSP, triconnectivity, dynamic DFS, maxflow/mincut, submodular maximization, k-edge connectivity, and others (see Table 1). Our techniques are also inherently parallel (over small $\mathrm{poly}(\log n)$ depth) and may have additional implications in scalable models like the work-depth and distributed models and provide motivation for a new *noisy setting* in dynamic algorithms, with connections to differential privacy Dwork et al. (2006) and sensitivity analysis Varma and Yoshida (2023); Kumabe and Yoshida (2022). Our results are shown in Table 1. We give an abridged version of all of our technical details in Appendix B. Below, we first provide a description of our model in Definition 1.

Definition 1 (Predicted-Updates Dynamic Model) *In the* ***predicted-updates dynamic model****, we consider a ground set $\mathcal{S}$ of size $|\mathcal{S}|$. We are given updates for three different types of algorithms:*

1. *Offline Dynamic Algorithms: We are given a* predicted sequence of dynamic updates *P consisting of tuples $(e, type, day, i)$ where $e \in \mathcal{S}$ is the element that the event is performed on, $type$ is the type of event, day is the predicted day of the event,*[1] *and i (initially set to 0) is a counter for the number of times the prediction for this event is updated (i is a parameter that is used in our algorithms). The predictions can be given to us, online, in sets $P_1, P_2, \dots, P_{\log_2(T)}$ where $P_i \subseteq P_{i+1}$ and $|P_{i+1}| = 2 \cdot |P_i|$ as we see more real events.* [2] *More than one event can be predicted for a day, but only one real event occurs each day.*

2. *Incremental Partially Dynamic Algorithms: On each day exactly one of the following occurs:*

   *(a) An element $e \in \mathcal{S}$ is inserted (a real insertion), and reports a prediction, $(e, \text{`}delete', day, i)$, of the day on which it will be deleted (*a predicted deletion*).*

   (b) A previously inserted element is deleted (a real deletion).

3. *Decremental Partially Dynamic Algorithms: From the outset, the algorithm is given a set P of all the elements that are predicted to ever appear in the system, and predictions for when each of the elements in P will be inserted. Then, on each day, one of the following occurs:*

   (a) An element e, either in P or not, is inserted,

   (b) A previously inserted element is deleted, and provides a prediction of when it will be reinserted (if ever).

   As in Item 1 (with the same restrictions), the original insertion predictions can also be given to us in sets $P_1, P_2, \dots, P_{\log_2(T)}$, online, as we see more real deletions.

An algorithm computes a function $f(\cdot)$, which is a solution to a problem $\mathcal{P}$, in the predicted-updates dynamic model, if on every day $t \in [T]$, the algorithm outputs $f(S)$, where $S \subseteq \mathcal{S}$ is the subset of elements (from the universe $\mathcal{S}$ of elements) induced by the true (not predicted) sequence of element insertions and deletions that occurred on all days $t' \leq t$.

Our main result is given informally in Theorem 1.1. The corresponding formal results can be found in Theorem 2.1, Corollary 11, and Corollary 12.

1. A *day* is our chosen unit of time for describing updates. One can, of course, use another unit of time as long as real events occur on unique timestamps.
2. In other words, as we see more real events, we receive more predictions. The only requirement on the sets $P_1, \dots, P_{\log_2(T)}$ is that the predicted days for new events cannot be earlier than the day each set of predictions is given. In particular, it is possible that multiple events are predicted for the same day.

Theorem 1.1 (Predicted-Updates Dynamic Model Framework (Informal))
Given a balanced, offline divide-and-conquer algorithm, $\mathcal{A}$ (defined in Definition 3), which performs $\widetilde{O}(|W|^c)$ work per subproblem W of size $|W|$, we can construct an algorithm in the predicted-updates model that, over T real events, does total work,

$$\widetilde{O}(T + ||\text{error}||_1), \text{ when } c \leq 1, \qquad \text{and } \widetilde{O}((T + ||\text{error}||_1) \cdot T^{c-1}), \text{ when } c > 1,$$

in expectation and with high probability, where $||\text{error}||_1$ is the sum over all elements of the absolute difference between the predicted deletion day and the actual deletion day.

Given an incremental or decremental algorithm, $\mathcal{A}$, with worst-case update time $\text{update}(\mathcal{A})$, we can construct an algorithm in the predicted-updates model that, over T real events, does total work, in expectation and with high probability,

$$\widetilde{O}\left((T + ||\text{error}||_1) \cdot \text{update}(\mathcal{A})\right).$$

Furthermore, given a fully-dynamic algorithm B for the problem, that does at most $R_B(T)$ total work by day T, we can get an algorithm that, over T updates, does total work

$$\widetilde{O}\left(\min\left\{(T + ||\text{error}||_1) \cdot \text{update}(\mathcal{A}),\ R_B(T)\right\}\right),$$

in expectation and with high probability (where $\text{update}(\mathcal{A})$ depends on c, see above, in the offline case). [3]

The above theorem require worst-case incremental and decremental algorithms (but not worst-case offline algorithms) which was shown to be necessary even in the "0-error" case by Peng and Rubinstein (2023); thus, such a requirement is necessary for our setting also.

Comparison with Recent Concurrent, Related, Independent Work Recent concurrent work of van den Brand et al. (2023) and Henzinger et al. (2023) also study algorithms in dynamic graph models with prediction. Henzinger et al. (2023) focus on lower bounds in their paper under different prediction models from our work. van den Brand et al. (2023) also study different types of prediction models including a model very similar to our predicted-deletions model, where instead of using the ℓ_1-error of a prediction, they instead look at the number of element-wise inversions between the predicted *deletion* sequence and the real sequence. They give a deterministic algorithm for incremental algorithms with predicted deletions based on Peng and Rubinstein (2023). Their reduction maintains the state of an incremental algorithm, in which elements are inserted in approximately reverse deletion order. In this paper, we give a reduction from the decremental setting with predicted *insertions* to the incremental setting that can also be applied to their result. The reduction applied to their construction can be interpreted as doing essentially the reverse: it maintains the state of a decremental algorithm, in which elements are deleted in approximately reverse insertion order.

Their framework does not handle offline to fully dynamic transformations for two reasons: 1) it is unclear how to simultaneously insert elements in both approximately reverse deletion order *and* reverse insertion order when given noisy predictions, and 2) their reduction is tied to incremental algorithms where it is even unclear what types of algorithms to ask for in the offline setting. We sidestep both issues in our framework with the key data structure used in our solution: the random *partition-tree* (Definition 2).

3. We use $\widetilde{O}(\cdot)$ to hide polylogarithmic factors in $|\mathcal{S}|$ (the universe of elements) and T (total number of real events).

Recent independent and concurrent work of Agarwal and Balkanski (2023) studies the problem of *dynamic submodular function maximization with cardinality constraints* in a model similar to our predicted update model. Their update time is given in terms of two parameters: w which constitutes the magnitude of a "small" error, and η which is the number of elements with error larger than w. Their update time is then polylogarithmic in η and w. This form of update time is incomparable to runtimes in the form of our result. They also study a different form of submodular maximization, we consider matroid constraints as opposed to cardinality.

2. Offline Divide-and-Conquer to Fully-Dynamic Transformation

This section presents our main framework for offline dynamic to fully dynamic transformations.

2.1. Preliminaries

We denote the *total* number of updates in our update sequence by T. We use $[t]$ to denote the set of positive integers up to t, ie. $\{1, \dots, t\}$. We use $\widetilde{O}(\cdot)$ to hide $\mathrm{poly}(\log(|\mathcal{S}| \cdot T))$ factors where $\mathcal{S}$ is our ground set for the elements and T is the total number of updates. Often, it holds that $|\mathcal{S}| = \mathrm{poly}(T)$.

An overview of Algorithm 2.1 that converts an offline algorithm to a predicted-update algorithm is given in Section 2.2. Below, we first define the ***partition-tree***, which is generated randomly in our algorithm, and is the main data structure in our framework.

Definition 2 (Partition-Tree) *A partition-tree of a sequence of days is a binary tree such that every node in the tree is associated with a window (interval of time). Henceforth, we refer to all nodes in the partition-tree as* ***windows****. The tree has the following properties:*

1. *The root window of the tree is the full sequence of T days.*
2. *For each internal (non-leaf) window W, its children windows partition W. We denote the parent of W by* $\mathrm{parent}(W)$.
3. *Each leaf window is associated with a single day.*

Our framework works for any c***-divide-and-conquer*** algorithm defined below. For our offline to fully dynamic transformations, we show a number of interesting algorithms; in fact all of the currently known offline dynamic graph algorithms fall under this definition. For our incremental and decremental to fully dynamic transformations, we show that *any* incremental or decremental algorithm with *worst-case* update time can be framed as a c-divide-and-conquer algorithm and can, hence, be used in our framework.

Definition 3 (Divide-and-conquer algorithm) *A* c-divide-and-conquer *algorithm $\mathcal{A}$, for some constant $c > 0$, is one that can be computed over divide-and-conquer tree T (where each node is a window) using a function $f(\cdot)$ (to solve a problem $\mathcal{P}$) such that:*

1. *A polynomial amount of read-write memory* $\mathbf{M}$ *is given to the root window;*
2. *Algorithm $\mathcal{A}$ computes $f(W, \mathbf{M}_{\mathrm{parent}(W)})$ for each window W of T, where the computation is given the* unordered *set of events occurring in W, and read-write access to* $\mathbf{M}_{\mathrm{parent}(W)}$ *which is the state of* $\mathbf{M}$ *after* $\mathrm{parent}(W)$ *has terminated.* $\mathbf{M}$ *is passed down and mutated by windows along the path from the root, but is* not *mutated by any sibling windows.*[4]

4. Consider the k-edge connectivity problem, an example of objects stored in read-write memory are the contracted components formed by edges that have insertion and deletion days outside of the current window. Along a root to leaf path, more components in $\mathbf{M}$ become contracted and this new set of (newly contracted) components is the read-write $\mathbf{M}$ that is passed down to children.

3. *The* expected *work done at window W is $O(\Gamma \cdot |W|^c)$, where Γ represents* poly $\log(|\mathcal{S}| \cdot T)$ *factors or update$(\mathcal{A})$, the worst-case incremental/decremental update time of algorithm $\mathcal{A}$.*[5]

4. *The solution to $\mathcal{P}$ is obtained from the windows in the tree and the memory,* $\mathbf{M}$.

Item 3 ensures that the work done by the divide-and-conquer algorithm is evenly distributed across the tree. This is the analogue of our condition for the incremental/decremental algorithms to have worst-case (not amortized) bounds.

2.2. Algorithm Description

We obtain two separate categories of inputs to the algorithm: offline inputs received during preprocessing and online inputs. Algorithm 2.1 uses the partition-tree $\mathcal{T}$, the divide-and-conquer algorithm $\mathcal{A}$ that computes $f(\cdot)$, and the predicted sequence of dynamic updates P. The predicted sequence is given as tuples $(e, type, day, i)$ where e is the element, $type$ is the type of update (an insertion or deletion), day is the predicted day of the update, and i is the number of times the event has been rescheduled. Initially, all $i = 0$ for all events in the prediction sequence. Then, the actual set of updates is given as an online sequence of dynamic updates U. The algorithm outputs an answer to the function $f(\cdot)$ after each update on each day $t \in [T]$ where U_t is the updates in prefix t of U.

Given a raw set of update predictions P, we must first convert it into a feasible sequence of events in our model with error comparable to the original set of update predictions. In particular, the error of our converted sequence of predictions has error at most $O(\log(T))$ factor worse than the original set P. In Line 1, we do this preprocessing (with pseudocode for PREPROCESSPREDICTIONS(P) in Appendix C.1). To do this, we first convert P into an instance of metric online bipartite matching where P represents the requests and the set of days $t \in [T]$ are our servers. We produce a matching such that at most one event in P is matched to each $t \in [T]$ by using the online harmonic algorithm of Gupta and Lewi (2012) (that we observe can be run as a fast approximation algorithm). Then, finally, we do a post-processing on the produced matching. In the order of the days, we perform a linear scan to check if any deletions of events occur before their respective insertions. For any such deletions, we move the event to the same day as the (later) insertion. After this post-processing, we have a feasible sequence of events $\overline{P}$ which consists of at most two events per day: at most one insertion and at most one deletion. This procedure can also be applied to any constant number of event types with precedence constraints (i.e. one event type must be performed before another).

Algorithm 2.1 iterates through each day $t \in [T]$ (Line 17) and checks for each *real* (Line 18) and *predicted* event (Line 23) assigned to day t. A *real* event is an event that is assigned to day t by the update sequence. A *predicted* event is one that is assigned by our algorithm (i.e. an event that is predicted to occur on day t). For each real event of type $type$ and on element e (Line 19), we find the corresponding prediction for this event in $\overline{P}$ (Line 19). If this event is not in $\overline{P}$, then we assign the default prediction of $(e, type, \infty, 0)$ where ∞ indicates that we predict the event to occur at the very end of the update stream. If our current day t is less than the predicted day t_{predict} (Line 19), then we process the event as one that occurs *earlier* than predicted. We use the procedure PROCESSEVENTEARLIERTHANPREDICTION$(E, t, t_{\text{predict}})$ (Line 20), explained below.

After iterating through the real events, we now iterate through the *predicted* events on day t (Line 23). If the predicted event $E = (e, type)$ was not a real event that occurred on day t (Line 24), then, we update our prediction since the real event will occur on a later day. To update our prediction,

5. For simplicity of expression in our proofs, we omit the factor Γ since it falls under our $\widetilde{O}(\cdot)$ bound for the offline transformations.

Algorithm 2.1: Fully Dynamic Algorithms with Predictions from Offline Divide-and-Conquer Algorithms

```
Input: Offline (during preprocessing): partition-tree $\mathcal{T}$, divide-and-conquer algorithm $\mathcal{A}$ that computes $f(\cdot)$,
       and predicted sequence of dynamic updates $P$. Online: Online sequence of dynamic updates $U =
       [E_1, \dots, E_T]$ where each event $E = (e, type)$ is a real event.
Output: After each day $t \in [T]$, output $f(U_t)$.
$\overline{P} \leftarrow$ PreprocessPredictions($P$)          // Converts $P$ into a feasible set of
  predictions
Insert all events in $\overline{P}$ as predicted events on their corresponding days
Procedure Retrigger($t_1, t_2$):
    Find $W$, the smallest window in $\mathcal{T}$ that contains both $t_1$ and $t_2$
    $S \leftarrow \{W\}$                                  // Set of windows to process.
    while $S \neq \emptyset$ do
        Remove a window $W'$ from $S$
        for each child $C$ of $W'$ do
            ProcessEvents($C$)
            $S \leftarrow S \cup \{C\}$
        end
    end
Procedure ProcessEventEarlierThanPrediction($E, t, t_{predict}$):
    Remove predicted event $E$ from day $t_{\text{predict}}$
    Add $E$ to day $t$ as a real event
    Retrigger($t, t_{predict}$)
Procedure ProcessEventLaterThanPrediction($E, i, t$):
    Remove predicted event $E$ from day $t$
    Add $E$ to day $(t + 2^i)$ as a predicted event
    Change the prediction for $E = (e, type)$ in $\overline{P}$ to $(e, type, t + 2^i, i)$
    if $e$ has a corresponding predicted deletion event that occurs earlier than $(t + 2^i)$ then
        Remove $e$'s predicted deletion event from the day $t'$ that it was scheduled for
        Add $e$'s predicted deletion event to day $(t + 2^i)$
        Let $(e, delete, t', i_{del})$ be the entry for $e$'s deletion event in $P$
        Change $e$'s deletion event prediction in $\overline{P}$ to $(e, delete, (t + 2^i), i_{del} + 1)$
    end
    Retrigger($t, t + 2^i$)
for day $t \in [T]$ do
    for each real event $E = (e, type)$ on day $t$ do
        Suppose event $E$ is an event of type $type$ on element $e$
        Find the corresponding prediction $(e, type, t_{\text{predict}}, i) \in \overline{P}$ for event $E$
        if $t < t_{predict}$ then
            ProcessEventEarlierThanPrediction($E, t, t_{predict}$)
        end
    end
    for each predicted event $E = (e, type)$ on day $t$ do
        if event $E$ wasn't a real event on day $t$ then
            ProcessEventLaterThanPrediction($E, i + 1, t$)
        end
    end
    Return Output($t$)
end
```

we call the procedure, PROCESSEVENTLATERTHANPREDICTION, for handling events that occur *later* than the predicted day. Finally, we call the function OUTPUT that is obtained from $\mathcal{A}$ for computing the output of our dynamic algorithm at t (Line 28).

We now describe each of our individual procedures that is called within our main algorithm:

RETRIGGER(t_1, t_2)**:** This procedure retriggers the processing of all events in *every descendant* of the smallest window W in $\mathcal{T}$ that contains both t_1 and t_2. The processing of the events uses the procedure PROCESSEVENTS (Line 7) that is obtained from $\mathcal{A}$.

ProcessEventEarlierThanPrediction$(E, t, t_{\textbf{predict}})$**:** This procedure processes a real event that occurs on a day t prior to its predicted day t_{predict}. First, we delete the predicted event E from day t_{predict} (Line 11). Then, we add E to day t as a real event (Line 11). Finally, we find the smallest window W in $\mathcal{T}$ that contains both t and t_{predict} (Line 4). We then call retrigger on W (Line 11).

ProcessEventLaterThanPrediction(E, i, t)**:** This procedure reschedules events which occur later than the prediction; specifically, the event is rescheduled to a future day. We first remove the predicted event E from day t (Line 13). We then increase the predicted day for the corresponding event from t to $t + 2^i$ where i is the number of times it has been rescheduled before (Line 13). Then, we find the smallest window W that contains both t and $(t + 2^i)$ (Line 4) and change the corresponding prediction to reflect the new prediction (Line 13). We now need to consider multiple possible updates associated with an element and ordering constraints among these updates. In the case of edge insertions/deletions, the insertion of an edge e must happen before the deletion of the edge (such is an example of an ordering constraint). We look for all corresponding predicted deletion events for e that occurs on an earlier day than $(t + 2^i)$ (Line 13) and reschedule them to day $(t + 2^i)$ (Line 14). Note that the conditional statement given in Line 13 can apply for any ordering constraints on any type of updates, not only insertions and deletions. We perform retrigger on W (Line 16).

2.3. Preprocessing and Scheduling of Predictions

Lemma 2.1 (Initial Scheduling Quality and Runtime) *Let* $\mathbf{p}$ *be the vector of predictions, mapping each event to a day in* $[T]$*. Let* $\mathbf{r}$ *be the (unknown to the algorithm) true vector of real events, mapping each event to the day in* $[T]$ *that it actually occurs.*

Given $\mathbf{p}$*, we can compute in time* $O(T \log^*(T))$ *an assignment vector* $\mathbf{a}_0$ *such that* $\mathbf{a}_0$ *assigns at most one insertion event and at most one deletion event to each day, assigns all insertions events before deletion events, and has error*

$$\mathbf{E}\left[||\mathbf{a}_0 - \mathbf{r}||_1\right] = O\left(||\mathbf{p} - \mathbf{r}||_1 \cdot \log(T)\right).$$

The formal proof of this lemma is given in Appendix C.1. We now compute the work of constructing our partition-tree, $\mathcal{T}$, using $\overline{P}$ and prove the depth of the constructed tree. We first prove the work to compute the partition-tree.

Lemma 4 (Work to Compute Initial Divide-and-Conquer Solutions) *The full divide-and-conquer algorithm over the partition-tree can be computed in expected time* $O(T^c \cdot \log^3(T) \cdot \log\log(T \cdot |\mathcal{S}|))$ *when* $c > 1$*, and* $O(T \cdot \log^3(T) \cdot \log\log(T \cdot |\mathcal{S}|))$ *when* $c \leq 1$.

Lemma 4 is a special case of Lemma 8. We now bound the depth of the partition-tree over the predictions in P, to bound the work of the divide-and-conquer. As the below analysis draws inspiration from the analysis of randomized quicksort, we relegate its proof to Appendix C.1.

Lemma 5 (Partition-Tree has $O(\log(T))$ Depth) *The depth of a random partition-tree drawn over T days is $O(\log(T))$ in expectation. Furthermore, for any constant $k \geq 36$, the depth is*

$$\leq k\ln(T) \qquad \textit{with probability} \qquad \geq 1 - T^{-\frac{k}{24}}.$$

Note that we do not need knowledge of T when computing the aforementioned bounds. We show in Algorithm C.3 how to remove the assumption on the value of T.

Finally, we give the work of maintaining the schedule after predictions are rescheduled based on the online dynamic sequence of updates. Below (in Lemma 6 with proof deferred to Appendix C.1) we show the maximum total number of times all predicted events are rescheduled by Algorithm 2.1 which combined with the work necessary to call RETRIGGER (which we analyze in Section 2.4) gives the total work of our algorithm.

Lemma 6 (Work to Maintain Schedule) *Algorithm 2.1 performs at most $O(T\log T)$ reschedules to the predictions of updates and maintains the predicted schedule of events over the course of the online dynamic updates.*

2.4. Work of RETRIGGER

The main workhorse of our algorithm is the RETRIGGER(t_1, t_2) operation, which recomputes the subtree of the partition-tree rooted at the smallest window W that contains t_1 and t_2. This operation recomputes the part of the partition-tree that can be affected by rearranging events that are scheduled between t_1 and t_2. We highlight the following lemma that forms the basis of our algorithm.

Lemma 7 (Random Partition-Tree Preserves Lengths in Expectation) *Consider a random partition-tree drawn according to Definition 2 for time sequence $[T]$. Then, for any $t_1, t_2 \in [T]$ where $t_1 \neq t_2$, the expected size of the smallest window that strictly contains both t_1 and t_2 is $O(|t_2 - t_1| \cdot \log(T))$.*

Proof We prove this lemma via a coupling argument where we show another way to generate the same distribution of binary tree partitions that will be easier to analyze. For each possible divider $d \in [T-1]$ of the original sequence, we associate d with a rank r_d drawn uniformly at random from the uniform distribution over $[0, 1]$. The ranks are drawn independently for each d.

Using the ranks, we assign the tree structure from the top down. At the top level, the divider with the lowest rank is used to split the sequence into the left child and the right child. Iteratively, for each new window of the tree, we use the lowest ranked divider of its subsequence to split the sequence. This results in the same distribution over partition-trees as Definition 2, as at each level, each of the possible dividers is equally likely to be chosen next.

Now, we see that a contiguous subsequence $[t_{\text{start}}, t_{\text{end}}]$ of $[T]$ is a window in the partition-tree if and only if the divider directly preceding t_{start} and the divider directly following t_{end} both have lower ranks than all of the dividers between t_{start} and t_{end}, where we can consider the endpoints of the original sequence over all days in $[T]$ to be dividers with rank 0. This is illustrated in Figure 1.

Without loss of generality let $t_1 < t_2$. Let W be the smallest window that strictly contains t_1 and t_2. We have that the size of W (in days) is $|W| = (t_2 - t_1 + 1) + L + R$, where $t_2 - t_1 + 1$ are the number days between the dividers bordering t_1 and t_2. L is a random variable representing the number of days before t_1 until we reach one that is bordering a divider with strictly smaller rank than the $t_2 - t_1 + 2$ dividers drawn between t_1 and t_2 and the dividers bordering t_1 and t_2. Symmetrically, R is the random variable for days after t_2 until we reach a divider with smaller

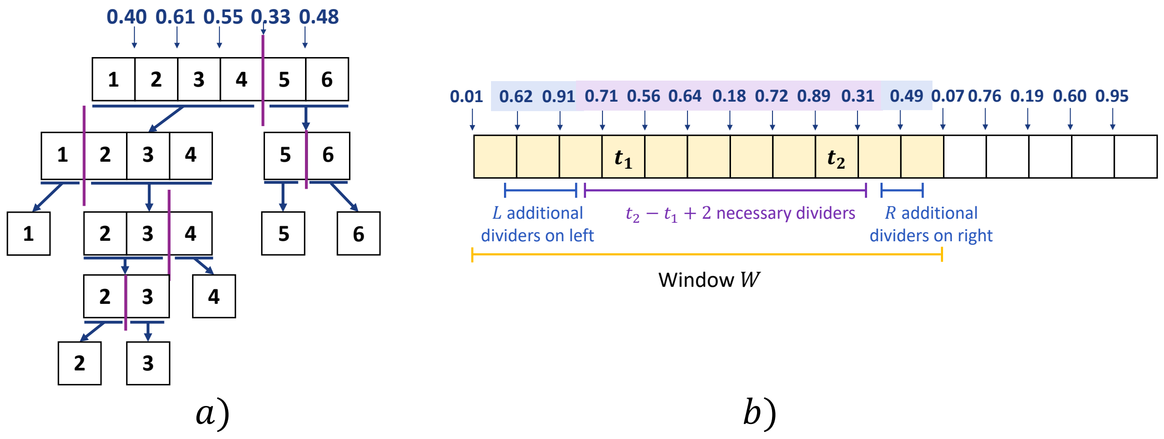

Figure 1: In Fig. a), dividers are drawn uniformly at random from $[0, 1]$ and shown in blue above the figure. This example contains 5 dividers since $T = 6$. $[2, 4]$ is a window in the tree because $0.40, 0.33 < 0.61, 0.55$. $[2, 5]$ is not a window in the tree because $0.48 \not< 0.33$. In Fig. b), W is the smallest window containing $[t_1, t_2]$. The random variables L and R capture the number of additional days added to $[t_1, t_2]$ to satisfy the property in a). The dividers bordering W must have smaller value than all dividers in W. The dividers in L and R have *larger* values than the dividers bordering $[t_1, t_2]$ and the dividers contained within $[t_1, t_2]$; otherwise, W would not be the smallest window containing $[t_1, t_2]$.

rank. This is illustrated in Figure 1. L and R must each contain dividers with ranks *larger* than the dividers between t_1 and t_2; otherwise, W would not be the smallest window strictly containing t_1, t_2, and all days in between.

First, we analyze $\mathbf{E}[L]$. Let $Z = \min\{r_d : d \in [t_1 - 1, t_2 + 1]\}$ be the minimum rank of the dividers between t_1 to t_2 and including the ones bordering t_1 and t_2. We can compute $\mathbf{E}[L]$ by conditioning on different values of Z. Let $p_Z(\cdot)$ be the p.d.f. of Z.

$$\mathbf{E}[L] = \int_{z=0}^{1} \mathbf{E}[L|Z = z] \cdot p_Z(z)dz \quad \le 1 + \int_{z=1/T}^{1} \frac{1}{z} \cdot p_Z(z)dz$$

Now, we can find the p.d.f. of Z via its c.d.f. Let $F_Z(\cdot)$ be the c.d.f. of Z.

$$F_Z(z) = \mathbf{P}[Z < z] = 1 - (1 - z)^{t_2 - t_1 + 2}$$
$$p_Z(z) = \frac{d}{dz}(F_Z(z)) \quad = (t_2 - t_1 + 2)(1 - z)^{t_2 - t_1 + 1}.$$

This allows us to conclude the following since $\int_{z=1/T}^{1} \frac{1}{z}dz = \ln(T)$:

$$\mathbf{E}[L] \le 1 + \int_{z=1/T}^{1} \frac{(t_2 - t_1 + 2)(1 - z)^{t_2 - t_1 + 1}}{z}dz \quad \le 1 + (t_2 - t_1 + 2)\ln(T).$$

Symmetrically, by the same argument, $\mathbf{E}[R] \le 1 + \ln(T)$ as well. So by linearity of expectation, we have $\mathbf{E}[W] = (t_2 - t_1 + 1) + \mathbf{E}[L] + \mathbf{E}[R] = O(|t_2 - t_1| \cdot \log(T))$. ■

This lemma is morally similar to a randomized tree-embedding scheme, in the vein of Fakcharoenphol et al. (2004). We discuss this connection in Appendix C.5. Now, we are ready to bound the work of the RETRIGGER operation. The work is defined in terms of a c-divide-and-conquer algorithm as defined in Definition 3.

Lemma 8 (Expected Work of RETRIGGER Operation) *Provided a c-divide-and-conquer algorithm (Definition 3), the expected work of* RETRIGGER(t_1, t_2) *is bounded by*

$$\widetilde{O}\left(|t_2 - t_1| \cdot T^{c-1}\right) \quad \textit{for } c > 1, \qquad \textit{and } \widetilde{O}\left(|t_2 - t_1|\right) \quad \textit{for } c \leq 1.$$

The proof follows from using Lemma 7 to bound the width of the subtree that is retriggered, and then applying a standard divide-and-conquer argument to bound the total work. The formal proof is deferred to Appendix C.2.

Lemma 9 (Expected Work Over All Calls to RETRIGGER) *The expected work done by Algorithm 2.1 over all calls to* RETRIGGER *is*

$$\widetilde{O}(|\mathbf{p} - \mathbf{r}|_1 \cdot T^{c-1}) \quad \textit{when } c \geq 1, \qquad \textit{and } \widetilde{O}(|\mathbf{p} - \mathbf{r}|_1) \quad \textit{when } c < 1.$$

The proof follows from observing that the total work triggered by one event can be bounded by a guess-and-double sequence that blows up the work by at most a constant factor. Then the work can be aggregated over all calls to RETRIGGER by linearity of expectation. The formal proof is given in Appendix C.2.

2.5. Putting Everything Together: the Final Theorem

We observe that the guarantees from Lemma 2.1 and Lemma 9, and therefore Lemma 13 as well, require the algorithm to have access to the size of the time horizon T. We show how to set this via a guess-and-double procedure assuming we receive sets of updates $P_1, P_2, \ldots, P_T$ of successively larger sizes where $P_i \subseteq P_{i+1}$ and $|P_{i+1}| = 2 \cdot |P_i|$. Together, this gives us Algorithm C.3 (which composes Algorithm 2.1 and Algorithm C.2 and is given in Appendix C.4) and our final theorem Theorem 2.1. The proof follows from the previous lemmas, a standard guess-and-double argument, and a standard boosting argument (described in Appendix C.3), and is given in Appendix C.4.

Theorem 2.1 (Offline Divide-and-Conquer to Fully Dynamic with Predictions Reduction) *Given a c-divide-and-conquer algorithm $\mathcal{A}$ that computes the solution to a problem $\mathcal{P}$, sets of predictions $P_1, P_2, \ldots, P_{\log_2(T)}$, and online sequence of events $U = [E_1, \ldots, E_t]$, an improved Algorithm 2.1 correctly outputs the solution to $f(U_t)$ after each $t \in [T]$ and uses total work, with high probability,*

$$\widetilde{O}\left(T^{c-1} \cdot (T + ||\mathbf{p} - \mathbf{u}||_1)\right) \quad \textit{when } c > 1, \qquad \textit{and } \widetilde{O}\left(T + ||\mathbf{p} - \mathbf{u}||_1\right) \quad \textit{when } c \leq 1.$$

Furthermore, our algorithm never performs worse than the best-known fully dynamic algorithm for problem $\mathcal{P}$. Here, $\mathbf{p}$ is our predicted sequence of days and $\mathbf{u}$ is the real sequence.

Finally, we observe that this framework gives a stronger result in the update/query model, when we reduce to an algorithm that has worst-case query time. That is, an algorithm that handles updates *without* information about queries, can continue to do so in this model. This is in contrast to some offline divide-and-conquer algorithms that use information about what elements will be queried to make decisions. The proof is given in Appendix C.4.

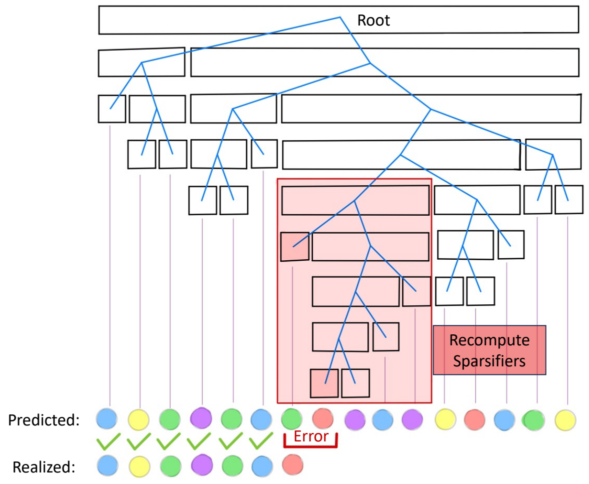

Figure 2: The black rectangles represent the windows of the random partition tree. The red rectangle denotes the part of the partition tree data structure that must be recomputed due to the difference between the predicted and realized updates.

Corollary 10 (Bound for Update/Query problems) *Given a c-divide-and-conquer algorithm $\mathcal{A}$ for $f(\cdot)$ with worst-case query time $\mathrm{query}(A)$, we can construct an algorithm, such that with high probability with respect to T, that has total work*

$$\widetilde{O}\left(T^{c-1}\cdot(T+||\mathbf{p}-\mathbf{d}||_1)\right)\ \textit{when } c>1, \qquad \textit{and } \widetilde{O}\left(T+||\mathbf{p}-\mathbf{d}||_1\right)\ \textit{when } c\leq 1,$$

and the worst-case query time is always bounded by $\mathrm{query}(A)$. *Furthermore, our algorithm never performs worse than the best-known fully dynamic algorithm for problem $\mathcal{P}$.*

2.6. Example Application of Framework for 3-Edge Connectivity

The best-known algorithm for 3-edge connectivity in the fully dynamic setting achieves $n^{o(1)}$ time per update while, with our framework, we can obtain $\tilde{O}(1)$ time per update. We describe the algorithm that our framework provides for 3-edge connectivity in our framework, as an example, to motivate that this algorithm is indeed simple and implementable.

Assume for simplicity that the time horizon T is known. Initialize a partition tree over T days as follows. The root node is associated with the full sequence of T days. Choose a day t uniformly at random between 1 and $T-1$. Associate the left child of the root with the sequence $[1, t-1]$ and the right child with $[t, T-1]$. Continue to recursively split the sequences to create the rest of the nodes in the tree, until the leaves, which are associated with singleton days, are reached. We will think of each node in this partition tree as a recursive subproblem in our divide-and-conquer algorithm. We call the updates associated with each node “windows”. See Figure 2.

We begin by solving the offline dynamic problem that we get by assuming that the predicted events will be perfectly correct. To do this, we start at the root of the partition-tree and move down. In each window, we maintain a sparsifier where the sparsifier has size equal to the number of non-permanent updates/queries within the window. A sparsifier is a sparser graph consisting of subsets of edges/vertices of the original graph. The sparsifier in each window contracts as many of the permanent edges as possible into *cactus graphs* (graphs of edge-disjoint cycles). Our framework directly uses the procedure given by Peng et al. (2017) to produce the cactus graph sparsifier in each window. For more information about the cactus, we refer the reader to their paper.

After we have finished the preprocessing step, i.e. solving the offline version of the problem, we move to the online phase of the algorithm. On each day t, an edge insertion or deletion is revealed to us. Say for example, edge e is inserted. If e is not predicted to occur on day t, add e's insertion to the list of events scheduled for t, and delete e's insertion from the schedule for the day it was (incorrectly) predicted. There may be other events that we scheduled for day t, that we now know do not happen on day t. We reschedule these events to future days using a doubling trick (the first time an event is rescheduled, we move it up by one day, then two days, then 4 days, etc.).

Now that the schedule has been updated, we correct the partition tree. Let $\mathcal{T}$ be the set of all days that had at least one scheduled event that changed in this update. Find the smallest window w in the tree that contains all of the days in $\mathcal{T}$. Using the 3-edge connectivity sparsifier that is stored at this node, we recompute the cactus sparsifiers for all of the descendants of w, using the newly updated schedule of events. Finally, we output the solution that is associated with the leaf containing day t.

2.7. Extensions to Incremental and Decremental Settings

This same framework also allows us to lift incremental and decremental algorithms to the fully dynamic setting, given predictions of only deletion times for the incremental setting, and only insertion times for the decremental setting. Algorithms and proofs are given in Appendix D and Appendix E. Concurrent works also provide results for incremental to fully dynamic reductions and we give an in-depth comparison in Appendix D.1.

Corollary 11 (Incremental to Fully-Dynamic Transformation) *Given an incremental algorithm, $\mathcal{A}$, with expected worst-case update time* $\text{update}(\mathcal{A})$*, we can construct an algorithm in the predicted-deletion model (Definition 15) such that the total work done by the algorithm is*

$$\widetilde{O}\left(\text{update}(\mathcal{A}) \cdot (T + ||\mathbf{p} - \mathbf{d}||_1)\right)$$

with high probability, where $||\mathbf{p} - \mathbf{d}||_1$ is the ℓ_1 error of the deletion-time predictions.

Corollary 12 (Decremental to Fully Dynamic Transformation) *Consider a decremental algorithm, $\mathcal{A}$, which takes time* $\text{initialize}_{\mathcal{A}}(T)$ *to initialize a state containing up to T elements and has worst-case update time* $\text{update}(\mathcal{A})$*. Given such an algorithm, we can construct an algorithm for the predicted-insertion model (Definition 21) that does total work*

$$\widetilde{O}\left(\text{initialize}_{\mathcal{A}}(T) \cdot K + \text{update}(\mathcal{A}) \cdot (T + ||\mathbf{p} - \mathbf{i}||_1 + TK)\right),$$

with high probability, where $||\mathbf{p} - \mathbf{i}||_1$ is the ℓ_1 error of the insertion-time predictions of predicted elements, and K is the number of elements that were never predicted but are inserted.

Acknowledgments

We thank Gramoz Goranci and anonymous reviewers for providing very helpful references for application problems we address in this paper. We thank Pattara Sukprasert for pointing us to references on online bipartite matching. We also thank Aravindan Vijayaraghavan and Aditya Bhaskara for valuable feedback on the presentation and direction of this project.

References

Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 434–443. IEEE, 2014.

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.

Arpit Agarwal and Eric Balkanski. Learning-augmented dynamic submodular maximization, 2023. URL https://arxiv.org/abs/2311.13006.

Cezar-Mihail Alexandru, Pavel Dvořák, Christian Konrad, and Kheeran K. Naidu. Improved Weighted Matching in the Sliding Window Model. In Petra Berenbrink, Patricia Bouyer, Anuj Dawar, and Mamadou Moustapha Kanté, editors, *40th International Symposium on Theoretical Aspects of Computer Science (STACS 2023)*, volume 254 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:21, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-266-2. doi: 10.4230/LIPIcs.STACS.2023.6. URL https://drops.dagstuhl.de/opus/volltexte/2023/17658.

Spyros Angelopoulos, Christoph Dürr, Shendan Jin, Shahin Kamali, and Marc Renault. Online Computation with Untrusted Advice. In Thomas Vidick, editor, *11th Innovations in Theoretical Computer Science Conference (ITCS 2020)*, volume 151 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 52:1–52:15, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-134-4. doi: 10.4230/LIPIcs.ITCS.2020.52. URL https://drops.dagstuhl.de/opus/volltexte/2020/11737.

Antonios Antoniadis, Christian Coester, Marek Elias, Adam Polak, and Bertrand Simon. Online metric algorithms with untrusted predictions. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 345–355. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/antoniadis20a.html.

Etienne Bamas, Andreas Maggiori, Lars Rohwedder, and Ola Svensson. Learning augmented energy minimization via speed scaling. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 15350–15359. Curran Associates, Inc., 2020a. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/af94ed0d6f5acc95f97170e3685f16c0-Paper.pdf.

Etienne Bamas, Andreas Maggiori, and Ola Svensson. The primal-dual method for learning augmented algorithms. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20, Red Hook, NY, USA, 2020b. Curran Associates Inc. ISBN 9781713829546.

Nikhil Bansal, Niv Buchbinder, Anupam Gupta, and Joseph (Seffi) Naor. An o(log2k)-competitive algorithm for metric bipartite matching. In Lars Arge, Michael Hoffmann, and Emo Welzl, editors, *Algorithms – ESA 2007*, pages 522–533, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-75520-3.

Surender Baswana, Shreejit Ray Chaudhury, Keerti Choudhary, and Shahbaz Khan. Dynamic dfs in undirected graphs: Breaking the o(m) barrier. *SIAM Journal on Computing*, 48(4):1335–1363, 2019.

Leyla Biabani, Mark de Berg, and Morteza Monemizadeh. Maximum-Weight Matching in Sliding Windows and Beyond. In Hee-Kap Ahn and Kunihiko Sadakane, editors, *32nd International Symposium on Algorithms and Computation (ISAAC 2021)*, volume 212 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 73:1–73:16, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-214-3. doi: 10.4230/LIPIcs.ISAAC.2021.73. URL https://drops.dagstuhl.de/opus/volltexte/2021/15506.

V. Braverman, P. Drineas, C. Musco, C. Musco, J. Upadhyay, D. P. Woodruff, and S. Zhou. Near optimal linear algebra in the online and sliding window models. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 517–528, Los Alamitos, CA, USA, nov 2020. IEEE Computer Society. doi: 10.1109/FOCS46700.2020.00055. URL https://doi.ieeecomputersociety.org/10.1109/FOCS46700.2020.00055.

Parinya Chalermsook, Syamantak Das, Yunbum Kook, Bundit Laekhanukit, Yang P Liu, Richard Peng, Mark Sellke, and Daniel Vaz. Vertex sparsification for edge connectivity. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1206–1225. SIAM, 2021.

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

Justin Chen, Sandeep Silwal, Ali Vakilian, and Fred Zhang. Faster fundamental graph algorithms via learned predictions. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 3583–3602. PMLR, 17–23 Jul 2022a.

Justin Y Chen, Talya Eden, Piotr Indyk, Honghao Lin, Shyam Narayanan, Ronitt Rubinfeld, Sandeep Silwal, Tal Wagner, David Woodruff, and Michael Zhang. Triangle and four cycle counting with predictions in graph streams. In *International Conference on Learning Representations*, 2022b. URL https://openreview.net/forum?id=8in_5gN9I0.

Li Chen, Gramoz Goranci, Monika Henzinger, Richard Peng, and Thatchaphol Saranurak. Fast dynamic cuts, distances and effective resistances via vertex sparsifiers. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1135–1146. IEEE, 2020.

Lijie Chen, Ran Duan, Ruosong Wang, Hanrui Zhang, and Tianyi Zhang. An improved algorithm for incremental DFS tree in undirected graphs. In David Eppstein, editor, *16th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2018, June 18-20, 2018, Malmö, Sweden*, volume 101 of *LIPIcs*, pages 16:1–16:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi: 10.4230/LIPIcs.SWAT.2018.16. URL https://doi.org/10.4230/LIPIcs.SWAT.2018.16.

Edith Cohen, Ofir Geri, and Rasmus Pagh. Composable sketches for functions of frequencies: Beyond the worst case. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th*

International Conference on Machine Learning, volume 119 of *Proceedings of Machine Learning Research*, pages 2057–2067. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/cohen20a.html.

Michael S. Crouch, Andrew McGregor, and Daniel Stubbs. Dynamic graphs in the sliding-window model. In Hans L. Bodlaender and Giuseppe F. Italiano, editors, *Algorithms – ESA 2013*, pages 337–348, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-40450-4.

Debarati Das, Maximilian Probst Gutenberg, and Christian Wulff-Nilsen. A near-optimal offline algorithm for dynamic all-pairs shortest paths in planar digraphs. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3482–3495. SIAM, 2022.

Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM J. Comput.*, 31:1794–1813, 09 2002. doi: 10.1137/S0097539701398363.

Sami Davies, Benjamin Moseley, Sergei Vassilvitskii, and Yuyan Wang. Predictive flows for faster ford-fulkerson. In *Proceedings of the 40th International Conference on Machine Learning*, Jul 2023.

Michael Dinitz, Sungjin Im, Thomas Lavastida, Benjamin Moseley, and Sergei Vassilvitskii. Faster matchings via learned duals. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021. URL https://openreview.net/forum?id=kB8eks2Edt8.

Paul Dütting, Silvio Lattanzi, Renato Paes Leme, and Sergei Vassilvitskii. Secretaries with advice. In *Proceedings of the 22nd ACM Conference on Economics and Computation*, EC '21, page 409–429, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385541. doi: 10.1145/3465456.3467623. URL https://doi.org/10.1145/3465456.3467623.

Paul Dütting, Federico Fusco, Silvio Lattanzi, Ashkan Norouzi-Fard, and Morteza Zadimoghaddam. Fully dynamic submodular maximization over matroids. *arXiv preprint arXiv:2305.19918*, 2023. To appear in ICML 2023.

Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Proceedings of the Third Conference on Theory of Cryptography*, TCC'06, page 265–284, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3540327312. doi: 10.1007/11681878_14. URL https://doi.org/10.1007/11681878_14.

Talya Eden, Piotr Indyk, Shyam Narayanan, Ronitt Rubinfeld, Sandeep Silwal, and Tal Wagner. Learning-based support estimation in sublinear time. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=tilovEHA3YS.

Alessandro Epasto, Silvio Lattanzi, Sergei Vassilvitskii, and Morteza Zadimoghaddam. Submodular optimization over sliding windows. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, page 421–430, Republic and Canton of Geneva, CHE, 2017. International World Wide Web Conferences Steering Committee. ISBN 9781450349130. doi: 10.1145/3038912.3052699. URL https://doi.org/10.1145/3038912.3052699.

Alessandro Epasto, Mohammad Mahdian, Vahab Mirrokni, and Peilin Zhong. *Improved Sliding Window Algorithms for Clustering and Coverage via Bucketing-Based Sketches*, pages 3005–3042. 2022. doi: 10.1137/1.9781611977073.117. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611977073.117.

David Eppstein. Offline algorithms for dynamic minimum spanning tree problems. *Journal of Algorithms*, 17(2):237 – 250, 1994. ISSN 0196-6774. doi: https://doi.org/10.1006/jagm.1994.1033. URL http://www.sciencedirect.com/science/article/pii/S0196677484710339.

Jittat Fakcharoenphol and Satish Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *Journal of Computer and System Sciences*, 72(5):868–889, 2006.

Jittat Fakcharoenphol, Satish Rao, and Kunal Talwar. A tight bound on approximating arbitrary metrics by tree metrics. *Journal of Computer and System Sciences*, 69(3):485–497, 2004. ISSN 0022-0000. doi: https://doi.org/10.1016/j.jcss.2004.04.011. URL https://www.sciencedirect.com/science/article/pii/S0022000004000637. Special Issue on STOC 2003.

Moran Feldman, Paul Liu, Ashkan Norouzi-Fard, Ola Svensson, and Rico Zenklusen. Streaming submodular maximization under matroid constraints. In Mikolaj Bojanczyk, Emanuela Merelli, and David P. Woodruff, editors, *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, July 4-8, 2022, Paris, France*, volume 229 of *LIPIcs*, pages 59:1–59:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi: 10.4230/LIPIcs.ICALP.2022.59. URL https://doi.org/10.4230/LIPIcs.ICALP.2022.59.

Sebastian Forster, Gramoz Goranci, Yasamin Nazari, and Antonis Skarlatos. Bootstrapping dynamic distance oracles. *Proceedings of the 2023 Annual European Symposium on Algorithms (ESA)*, 2023.

Zvi Galil, Giuseppe F. Italiano, and Neil Sarnak. Fully dynamic planarity testing with applications. *J. ACM*, 46(1):28–91, jan 1999. ISSN 0004-5411. doi: 10.1145/300515.300517. URL https://doi.org/10.1145/300515.300517.

Sreenivas Gollapudi and Debmalya Panigrahi. Online algorithms for rent-or-buy with expert advice. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2319–2327. PMLR, 09–15 Jun 2019. URL https://proceedings.mlr.press/v97/gollapudi19a.html.

Gramoz Goranci. *Dynamic Graph Algorithms and Graph Sparsification: New Techniques and Connections*. PhD thesis, University of Vienna, 2019.

Gramoz Goranci, Monika Henzinger, and Thatchaphol Saranurak. Fast incremental algorithms via local sparsifiers. Manuscript, 2019.

Gramoz Goranci, Harald Räcke, Thatchaphol Saranurak, and Zihan Tan. The expander hierarchy and its applications to dynamic graph algorithms. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2212–2228. SIAM, 2021.

Anupam Gupta and Kevin Lewi. The online metric matching problem for doubling metrics. In Artur Czumaj, Kurt Mehlhorn, Andrew Pitts, and Roger Wattenhofer, editors, *Automata, Languages, and Programming*, pages 424–435, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

Kathrin Hanauer, Monika Henzinger, and Christian Schulz. Recent advances in fully dynamic graph algorithms – a quick reference guide. *ACM J. Exp. Algorithmics*, 27, dec 2022. ISSN 1084-6654. doi: 10.1145/3555806. URL https://doi.org/10.1145/3555806.

Monika Henzinger, Barna Saha, Martin P. Seybold, and Christopher Ye. On the complexity of algorithms with predictions for dynamic graph problems, 2023.

Monika Rauch Henzinger and Valerie King. Maintaining minimum spanning trees in dynamic graphs. In Pierpaolo Degano, Roberto Gorrieri, and Alberto Marchetti-Spaccamela, editors, *Automata, Languages and Programming, 24th International Colloquium, ICALP'97, Bologna, Italy, 7-11 July 1997, Proceedings*, volume 1256 of *Lecture Notes in Computer Science*, pages 594–604. Springer, 1997. doi: 10.1007/3-540-63165-8_214. URL https://doi.org/10.1007/3-540-63165-8_214.

Jacob Holm and Eva Rotenberg. Worst-case polylog incremental spqr-trees: Embeddings, planarity, and triconnectivity. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2378–2397. SIAM, 2020.

Chen-Yu Hsu, Piotr Indyk, Dina Katabi, and A. Vakilian. Learning-based frequency estimation algorithms. In *International Conference on Learning Representations*, 2018.

Piotr Indyk, Ali Vakilian, and Yang Yuan. Learning-based low-rank approximations. In *NeurIPS 2019 Workshop on Solving Inverse Problems with Deep Networks*, 2019. URL https://openreview.net/forum?id=S1l5s7298H.

Piotr Indyk, Frederik Mallmann-Trenn, Slobodan Mitrovic, and Ronitt Rubinfeld. Online page migration with ml advice. In Gustau Camps-Valls, Francisco J. R. Ruiz, and Isabel Valera, editors, *Proceedings of The 25th International Conference on Artificial Intelligence and Statistics*, volume 151 of *Proceedings of Machine Learning Research*, pages 1655–1670. PMLR, 28–30 Mar 2022. URL https://proceedings.mlr.press/v151/indyk22a.html.

Rajesh Jayaram, David P. Woodruff, and Samson Zhou. Truly perfect samplers for data streams and sliding windows. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '22, page 29–40, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392600. doi: 10.1145/3517804.3524139. URL https://doi.org/10.1145/3517804.3524139.

Tanqiu Jiang, Yi Li, Honghao Lin, Yisong Ruan, and David P. Woodruff. Learning-augmented data stream algorithms. In *International Conference on Learning Representations*, 2020a. URL https://openreview.net/forum?id=HyxJ1xBYDH.

Zhihao Jiang, Debmalya Panigrahi, and Kevin Sun. Online Algorithms for Weighted Paging with Predictions. In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *47th International Colloquium on Automata, Languages, and Programming (ICALP 2020)*, volume 168 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 69:1–69:18, Dagstuhl, Germany, 2020b. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. ISBN 978-3-95977-138-2. doi: 10.4230/LIPIcs.ICALP.2020.69. URL https://drops.dagstuhl.de/opus/volltexte/2020/12476.

Wenyu Jin and Xiaorui Sun. Fully dynamic st edge connectivity in subpolynomial time. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 861–872. IEEE, 2022.

B. Kalyanasundaram and K. Pruhs. Online weighted matching. *Journal of Algorithms*, 14(3): 478–488, 1993. ISSN 0196-6774. doi: https://doi.org/10.1006/jagm.1993.1026. URL https://www.sciencedirect.com/science/article/pii/S0196677483710266.

Sanjeev Khanna, Rajeev Motwani, and Randall H. Wilson. On certificates and lookahead in dynamic graph problems. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '96, page 222–231, USA, 1996. Society for Industrial and Applied Mathematics. ISBN 0898713668.

Samir Khuller, Stephen G. Mitchell, and Vijay V. Vazirani. On-line algorithms for weighted bipartite matching and stable marriages. In Javier Leach Albert, Burkhard Monien, and Mario Rodríguez Artalejo, editors, *Automata, Languages and Programming*, pages 728–738, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg. ISBN 978-3-540-47516-3.

Philip N Klein. Multiple-source shortest paths in planar graphs. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 146–155, 2005.

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 '18, page 489–504, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450347037. doi: 10.1145/3183713.3196909. URL https://doi.org/10.1145/3183713.3196909.

Soh Kumabe and Yuichi Yoshida. Lipschitz continuous algorithms for graph problems. *arXiv preprint arXiv:2211.04674*, 2022.

Ravi Kumar, Manish Purohit, and Zoya Svitkina. Improving online algorithms via ml predictions. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper_files/paper/2018/file/73a427badebe0e32caa2e1fc7530b7f3-Paper.pdf.

Srijan Kumar, Xikun Zhang, and Jure Leskovec. Predicting dynamic embedding trajectory in temporal interaction networks. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 1269–1278, 2019.

Silvio Lattanzi, Thomas Lavastida, Benjamin Moseley, and Sergei Vassilvitskii. *Online Scheduling via Learned Weights*, pages 1859–1877. 2020. doi: 10.1137/1.9781611975994.114. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611975994.114.

Thomas Lavastida, Benjamin Moseley, R. Ravi, and Chenyang Xu. Learnable and Instance-Robust Predictions for Online Matching, Flows and Load Balancing. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms (ESA 2021)*, volume 204 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 59:1–59:17, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-204-4. doi: 10.4230/LIPIcs.ESA.2021.59. URL https://drops.dagstuhl.de/opus/volltexte/2021/14640.

Thodoris Lykouris and Sergei Vassilvitskii. Competitive caching with machine learned advice. *J. ACM*, 68(4), jul 2021. ISSN 0004-5411. doi: 10.1145/3447579. URL https://doi.org/10.1145/3447579.

Andrés Muñoz Medina and Sergei Vassilvitskii. Revenue optimization with approximate bid predictions. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 1856–1864, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964.

Michael Mitzenmacher. A model for learned bloom filters, and optimizing by sandwiching. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 462–471, Red Hook, NY, USA, 2018. Curran Associates Inc.

Michael Mitzenmacher. *Queues with Small Advice*, pages 1–12. 2021. doi: 10.1137/1.9781611976830.1. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611976830.1.

Michael Mitzenmacher and Sergei Vassilvitskii. *Algorithms with Predictions*, page 646–662. Cambridge University Press, 2021. doi: 10.1017/9781108637435.037.

Giang Hoang Nguyen, John Boaz Lee, Ryan A Rossi, Nesreen K Ahmed, Eunyee Koh, and Sungchul Kim. Continuous-time dynamic network embeddings. In *Companion proceedings of the the web conference 2018*, pages 969–976, 2018.

Odysseas Papapetrou, Minos Garofalakis, and Antonios Deligiannakis. Sketching distributed sliding-window data streams. *The VLDB Journal*, 24(3):345–368, jun 2015. ISSN 1066-8888. doi: 10.1007/s00778-015-0380-7. URL https://doi.org/10.1007/s00778-015-0380-7.

Binghui Peng and Aviad Rubinstein. Fully-dynamic-to-incremental reductions with known deletion order (e.g. sliding window). In *2023 Symposium on Simplicity in Algorithms (SOSA)*, pages 261–271, 2023. doi: 10.1137/1.9781611977585.ch24. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611977585.ch24.

Richard Peng, Bryce Sandlund, and Daniel Dominic Sleator. Optimal offline dynamic 2, 3-edge/vertex connectivity. In *Workshop on Algorithms and Data Structures*, 2017.

Enoch Peserico and Michele Scquizzato. Matching on the Line Admits No $o(\sqrt{\log n})$-Competitive Algorithm. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*, volume 198 of

Leibniz International Proceedings in Informatics (LIPIcs), pages 103:1–103:3, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-195-5. doi: 10.4230/LIPIcs.ICALP.2021.103. URL https://drops.dagstuhl.de/opus/volltexte/2021/14172.

Farimah Poursafaei, Shenyang Huang, Kellin Pelrine, and Reihaneh Rabbany. Towards better evaluation for dynamic link prediction. *Advances in Neural Information Processing Systems*, 35: 32928–32941, 2022.

Kha Gia Quach, Pha Nguyen, Huu Le, Thanh-Dat Truong, Chi Nhan Duong, Minh-Triet Tran, and Khoa Luu. Dyglip: A dynamic graph model with link prediction for accurate multi-camera multiple object tracking. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13784–13793, 2021.

Sharath Raghvendra. A Robust and Optimal Online Algorithm for Minimum Metric Bipartite Matching. In Klaus Jansen, Claire Mathieu, José D. P. Rolim, and Chris Umans, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2016)*, volume 60 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:16, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-018-7. doi: 10.4230/LIPIcs.APPROX-RANDOM.2016.18. URL http://drops.dagstuhl.de/opus/volltexte/2016/6641.

Sharath Raghvendra. Optimal Analysis of an Online Algorithm for the Bipartite Matching Problem on a Line. In Bettina Speckmann and Csaba D. Tóth, editors, *34th International Symposium on Computational Geometry (SoCG 2018)*, volume 99 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 67:1–67:14, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-066-8. doi: 10.4230/LIPIcs.SoCG.2018.67. URL http://drops.dagstuhl.de/opus/volltexte/2018/8780.

David Reitblat. Sliding-window streaming algorithms for graph problems and ℓ_p-sampling. Master's thesis, Rehovot, Israel, 2019.

Liam Roditty. Decremental maintenance of strongly connected components. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '13, page 1143–1150, USA, 2013. Society for Industrial and Applied Mathematics. ISBN 9781611972511.

Dhruv Rohatgi. *Near-Optimal Bounds for Online Caching with Machine Learned Advice*, pages 1834–1845. 2020. doi: 10.1137/1.9781611975994.112. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611975994.112.

Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. Temporal graph networks for deep learning on dynamic graphs. *arXiv preprint arXiv:2006.10637*, 2020.

Tim Roughgarden, editor. *Beyond the Worst-Case Analysis of Algorithms*. Cambridge University Press, 2021. doi: 10.1017/9781108637435.

Piotr Sankowski and Marcin Mucha. Fast dynamic transitive closure with lookahead. *Algorithmica*, 56:180–197, 2010.

Kapil Vaidya, Eric Knorr, Michael Mitzenmacher, and Tim Kraska. Partitioned learned bloom filters. In *International Conference on Learning Representations*, 2021.

Jan van den Brand, Danupon Nanongkai, and Thatchaphol Saranurak. Dynamic matrix inverse: Improved algorithms and matching conditional lower bounds. *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 456–480, 2019.

Jan van den Brand, Sebastian Forster, Yasamin Nazari, and Adam Polak. On dynamic graph algorithms with predictions. *CoRR*, abs/2307.09961, 2023. doi: 10.48550/ARXIV.2307.09961. URL https://doi.org/10.48550/arXiv.2307.09961.

Nithin Varma and Yuichi Yoshida. Average sensitivity of graph algorithms. *SIAM J. Comput.*, 52(4):1039–1081, 2023. doi: 10.1137/21M1399592. URL https://doi.org/10.1137/21m1399592.

Yanbang Wang, Yen-Yu Chang, Yunyu Liu, Jure Leskovec, and Pan Li. Inductive representation learning in temporal networks via causal anonymous walks. *arXiv preprint arXiv:2101.05974*, 2021.

Alexander Wei. Better and Simpler Learning-Augmented Online Caching. In Jarosław Byrka and Raghu Meka, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2020)*, volume 176 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 60:1–60:17, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. ISBN 978-3-95977-164-1. doi: 10.4230/LIPIcs.APPROX/RANDOM.2020.60. URL https://drops.dagstuhl.de/opus/volltexte/2020/12663.

Alexander Wei and Fred Zhang. Optimal robustness-consistency trade-offs for learning-augmented online algorithms. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20, Red Hook, NY, USA, 2020. Curran Associates Inc. ISBN 9781713829546.

Zhewei Wei, Xuancheng Liu, Feifei Li, Shuo Shang, Xiaoyong Du, and Ji-Rong Wen. Matrix sketching over sliding windows. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1465–1480, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450335317. doi: 10.1145/2882903.2915228. URL https://doi.org/10.1145/2882903.2915228.

David P. Woodruff and Samson Zhou. Tight bounds for adversarially robust streams and sliding windows via difference estimators. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1183–1196, 2022. doi: 10.1109/FOCS52979.2021.00116.

Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. *Advances in neural information processing systems*, 31, 2018.

Appendix A. Table of Contents of Main Paper and Appendix

Contents

Appendix B. Technical Overview

First, we define and motivate the *predicted-updates dynamic model*. Then, we introduce and provide a technical overview of our framework to design algorithms for this model. Finally, we describe specific problems for which our framework is able to outperform state-of-the-art fully dynamic algorithms.

B.1. The Predicted-Updates Dynamic Model

We introduce the *predicted-updates dynamic model*, which is a general model for algorithms with predictions in the dynamic setting that applies to a wide range of problems in the traditional offline, incremental, and decremental dynamic settings. In this paper, we refer to *days* to indicate an ordering to the update operations. The updates can easily be presented as an ordered sequence instead of being tied to days. However, we find that speaking in terms of *days* is a useful abstraction for describing our results. Throughout, we use *time* and *timestamp* interchangeably with *day*. We define an ***event*** to be an update of a certain type on an element $e \in \mathcal{S}$ where $\mathcal{S}$ is the ground set of all possible elements in the dataset. We differentiate between ***real*** and ***predicted*** events to be updates that actually occur on a day versus an update that is predicted to occur on a day.

Definition 1 (Predicted-Updates Dynamic Model) *In the **predicted-updates dynamic model**, we consider a ground set $\mathcal{S}$ of size $|\mathcal{S}|$. We are given updates for three different types of algorithms:*

1. *Offline Dynamic Algorithms: We are given a* predicted sequence of dynamic updates *P consisting of tuples $(e, type, day, i)$ where $e \in \mathcal{S}$ is the element that the event is performed on, $type$ is the type of event, day is the predicted day of the event,*[6] *and i (initially set to 0) is a counter for the number of times the prediction for this event is updated (i is a parameter that is used in our algorithms). The predictions can be given to us, online, in sets $P_1, P_2, \dots, P_{\log_2(T)}$ where $P_i \subseteq P_{i+1}$ and $|P_{i+1}| = 2 \cdot |P_i|$ as we see more real events.* [7] *More than one event can be predicted for a day, but only one real event occurs each day.*

2. *Incremental Partially Dynamic Algorithms: On each day exactly one of the following occurs:*

   *(a) An element $e \in \mathcal{S}$ is inserted (a real insertion), and reports a prediction, $(e, \text{`}delete', day, i)$, of the day on which it will be deleted (*a predicted deletion*).*

   (b) A previously inserted element is deleted (a real deletion).

3. *Decremental Partially Dynamic Algorithms: From the outset, the algorithm is given a set P of all the elements that are predicted to ever appear in the system, and predictions for when each of the elements in P will be inserted. Then, on each day, one of the following occurs:*

   (a) An element e, either in P or not, is inserted,

   (b) A previously inserted element is deleted, and provides a prediction of when it will be reinserted (if ever).

6. A *day* is our chosen unit of time for describing updates. One can, of course, use another unit of time as long as real events occur on unique timestamps.

7. In other words, as we see more real events, we receive more predictions. The only requirement on the sets $P_1, \dots, P_{\log_2(T)}$ is that the predicted days for new events cannot be earlier than the day each set of predictions is given. In particular, it is possible that multiple events are predicted for the same day.

As in Item 1 (with the same restrictions), the original insertion predictions can also be given to us in sets $P_1, P_2, \ldots, P_{\log_2(T)}$, online, as we see more real deletions.

An algorithm computes a function $f(\cdot)$, which is a solution to a problem $\mathcal{P}$, in the predicted-updates dynamic model, if on every day $t \in [T]$, the algorithm outputs $f(S)$, where $S \subseteq \mathcal{S}$ is the subset of elements (from the universe $\mathcal{S}$ of elements) induced by the true (not predicted) sequence of element insertions and deletions that occurred on all days $t' \leq t$.

Note that an algorithm in this model *must* compute each day's output *correctly*. The runtime of the algorithm, however, may depend on the prediction error. While there is significant recent work surrounding algorithms with predictions, this, along with concurrent, independent works van den Brand et al. (2023); Henzinger et al. (2023), to the best of our knowledge, are the first to study the dynamic model.[8] We motivate the predicted-updates dynamic model by examining some key features.

Modularity. In many settings of the algorithms-with-predictions domain, the requested predictions are problem-specific. This is necessary for some problems, where the purpose of the prediction is to provide a partial solution. However, this requires algorithm designers to provide a custom prediction model, error metric, and algorithm tailored to each new problem. In our model, our requested predictions of element update times are not problem-specific, and are instead applicable to any dynamic problem. Hence, techniques can translate across wide classes of problems, as demonstrated by our framework. We also emphasize that our model makes no distributional assumptions about the input. Rather, our guarantees are in the form of *worst-case analysis* with respect to a new parameter.

Practical and efficient machine learning. A main motivation of algorithms-with-predictions is to take advantage of the predictive power of machine learning models. Our model and framework allows us to train a model once e.g., to predict insertion/deletion times of edges in a graph (as a vector of values), and use it many times for a wide range of problems e.g., a collection of dynamic graph problems. Additionally, our framework can handle predictions that are ill-formed, i.e. not feasible update sequences, so we do not have to take into account possibly complicated feasibility requirements in this induced learning problem. These properties allow us to avoid the costly process of designing and training models that predict problem-specific parameters for each new problem. Additionally, since our requested predictions do not depend on the solutions to a specific problem $f(\cdot)$, we *do not* need to compute $f(\cdot)$ on the dynamic graphs we encounter in the training stage. This could be a very large saving, as computing $f(\cdot)$ on each training instance, in many cases, takes time superlinear in the size of the instance. Finally, we note that a rapidly growing area of empirical research shows that it is possible to use machine learning to generate high-quality predictions of edge insertions and deletions in graphs Kumar et al. (2019); Nguyen et al. (2018); Poursafaei et al. (2022); Quach et al. (2021); Rossi et al. (2020); Wang et al. (2021); Zhang and Chen (2018).

Interpolation. This model provides a *beyond-worst-case paradigm* for dynamic algorithms that interpolates between the offline and partially dynamic settings (with zero prediction error) and fully dynamic setting (with large prediction error). Beyond-worst-case models can give us insight into what bottlenecks are inherent to a problem and model, and inspire new algorithmic techniques that

8. Specifically, dynamic problems that compute functions over a subset S of some ground set $\mathcal{S}$, where S is subject to insertions and deletions. See e.g. this recent survey Hanauer et al. (2022) for examples of such algorithms.

could transfer between models (see Roughgarden (2021) for an in-depth discussion). We hope that studying the predicted-updates dynamic model can provide insights that inspire better fully dynamic algorithms and lower bounds.

B.2. Framework for Predicted-Updates Dynamic Algorithms

In conjunction with our model, we design an algorithmic framework that "lifts" offline, incremental, and decremental algorithms to the predicted-updates, fully dynamic setting. In the technical section of our paper, we use the term *work* to denote *runtime*.

Theorem 1.1 (Predicted-Updates Dynamic Model Framework (Informal))

Given a balanced, offline divide-and-conquer algorithm, $\mathcal{A}$ (defined in Definition 3), which performs $\widetilde{O}(|W|^c)$ work per subproblem W of size $|W|$, we can construct an algorithm in the predicted-updates model that, over T real events, does total work,

$$\widetilde{O}(T + ||\text{error}||_1),\ \textit{when } c \leq 1, \qquad \textit{and } \widetilde{O}((T + ||\text{error}||_1) \cdot T^{c-1}),\ \textit{when } c > 1,$$

in expectation and with high probability, where $||\text{error}||_1$ is the sum over all elements of the absolute difference between the predicted deletion day and the actual deletion day.

Given an incremental or decremental algorithm, $\mathcal{A}$, with worst-case update time $\text{update}(\mathcal{A})$, we can construct an algorithm in the predicted-updates model that, over T real events, does total work, in expectation and with high probability,

$$\widetilde{O}\left((T + ||\text{error}||_1) \cdot \text{update}(\mathcal{A})\right).$$

Furthermore, given a fully-dynamic algorithm B for the problem, that does at most $R_B(T)$ total work by day T, we can get an algorithm that, over T updates, does total work

$$\widetilde{O}\left(\min\left\{(T + ||\text{error}||_1) \cdot \text{update}(\mathcal{A}),\ R_B(T)\right\}\right),$$

in expectation and with high probability (where $\text{update}(\mathcal{A})$ depends on c, see above, in the offline case). [9]

Surprisingly, we are able to achieve all three desiderata of algorithms-with-predictions: consistency, competitiveness, and robustness, simultaneously. Note also that this guarantee holds *in hindsight*. That is, the algorithm does not require an estimate of the magnitude of $||\text{error}||_1$ to achieve this goal. Additionally, the potential to include a backstop means that, asymptotically, one can design algorithms that can take advantage of predictions *for free*, without losing the assurance of a worst-case guarantee. We believe that this framework is practically implementable, and does not rely on any algorithmic "heavy machinery." This is particularly attractive, as the intention of the model is to provide algorithms that perform well in practice. Furthermore, it is often the case that offline, incremental, and decremental algorithms are simpler than their fully dynamic counterparts, allowing us to utilize these algorithms in real-world settings.

9. We use $\widetilde{O}(\cdot)$ to hide polylogarithmic factors in $|\mathcal{S}|$ (the universe of elements) and T (total number of real events).

The ℓ_1 Error Metric The results of Theorem 1.1 give runtime bounds that depend on the ℓ_1 *norm* of the prediction error. We use this as a shorthand to denote the sum of absolute differences between the predicted time and realized time of each event. One way to interpret this, is that if $\mathbf{u}$ is a vector indexed by possible events that contains the true update times of each event, and $\mathbf{p}$ is another vector indexed by possible events that contains the predicted update times of each event, the ℓ_1 error of the prediction is $||\mathbf{u} - \mathbf{p}||_1$. If an event occurs but is never predicted, or vice versa, that event contributes T to the ℓ_1 error. Also, for events that occur multiple times, it suffices to consider the closest matching of duplicated events.

The ℓ_1 error metric is a natural one that has been well-studied in applications to warm-starts Dinitz et al. (2021); Davies et al. (2023). To contextualize the bound, consider predictions that, on average, are within $\text{poly}(\log(T))$ factor of days of the true event times. Then, the ℓ_1 error will be near-linear in T, and asymptotically, we achieve the runtime of the offline algorithm (up to $\text{poly}(\log(T))$ factors). On the other hand, the ℓ_1 error of any prediction is $O(T^2)$. So in the worst case, the algorithm will take runtime equivalent to solving T independent offline instances (which can be improved by providing the algorithm a worst-case backstop).

Another related error metric that we could have considered is bounding the runtime by a factor of $T \cdot ||\mathbf{u} - \mathbf{p}||_\infty$, where $||\mathbf{u} - \mathbf{p}||_\infty$ is the largest absolute error over all predictions. This bound is always at least the ℓ_1 error and is sometimes comparable in magnitude. In general, however, it is a significantly weaker bound. As an example, the ℓ_1 error can be (near-)linear in T even when there are a polylogarithmic number of events, which occur, that were never predicted. In this same case, $T \cdot ||\mathbf{u} - \mathbf{p}||_\infty$ would be $\Omega(T^2)$, giving a very different bound. Thus, a main focus of our work is getting this tighter dependence on ℓ_1 error rather than $T \cdot \ell_\infty$.

Random Partition-Tree Framework Our main technical contribution is the ***random partition-tree*** data structure (Definition 2). Our random partition-tree data structure is a tree drawn uniformly-at-random over the predicted event sequence; such a structure crucially mimicks a divide-and-conquer data structure. We show that such a simple data structure allows us to *confine the effects of prediction errors "locally."* Specifically, we present a way to *fix* the divide-and-conquer data structure on the fly, as prediction errors become apparent. We fix the structure by recomputing nodes (representing recursive subproblems in the divide-and-conquer) in the partition-tree and their descendants. Splitting the recursive subproblems *randomly* guarantees that the expected amount of recomputation that is triggered by prediction errors is $\widetilde{O}(|t' - t|)$, where t' is the predicted time and t is the real time of the event. Hence, we perform as much work per element as the prediction error, $|t' - t|$, associated with that element. This, fundamentally, allows our total ℓ_1 prediction error to be as large as the number of real updates, $\widetilde{O}(T)$, while maintaining work matching that of the corresponding offline, incremental, or decremental algorithm. Our result also makes an interesting connection to *metric tree-embeddings* where our metric of interest is time. We describe this connection in more detail in Appendix C.5.

Other Technical Contributions of the Framework In addition to our main contribution, we also solve a variety of other technical challenges listed below.

- *(Early deletion problem):* an element's true deletion time is earlier than its predicted deletion time. That is, we arrive at day t to see that element e is deleted, but we expected e to be active until some later day $t' > t$. This means that some subproblems in our divide-and-conquer recursion tree were based on an incorrect list of events, and are therefore not correct. These subproblems and their descendants need to be recomputed. Our random partition-tree, as described above, allows us to perform the computation in work linearly proportional to the prediction error, with small overhead.

- *(Late update problem):* an element's true update time is later than its predicted update time. That is, we reach a day t when we expect to see an update to element e, but element e is not updated on that day (i.e. there is no real event on element e).

  We address this by reinserting e as a prediction for a later time. We do this via a "guess-and-double" procedure. That is, the first time e is reinserted, we reinsert it for 1 day in the future, then 2 days in the future, then 4 days, etc. Then, we can again fix the data structure to account for this rescheduling using the random partition-tree, like the early deletion problem.

- *(Overscheduling problem):* the predicted event times in P could schedule arbitrarily many events for one day $t \in [T]$. This breaks our divide-and-conquer framework, which distributes work based on the crucial assumption that a small number of events happens per day. This problem is exacerbated by the repeated rescheduling we do to address the early deletion problem.

  We show how to reassign the scheduled element deletions, in a preprocessing step, so that not too many are scheduled for one day. We do this by framing our scheduling problem as an instance of *online metric matching* for the line metric (representing time), and using a known competitive algorithm of Gupta and Lewi (2012). Finally, we account for the rescheduling by assigning a ***batch*** of $O(\log T)$ events to each day, and accounting for these in our total work.

- *(Boosting and backstopping for competitiveness):* our work bounds are proven in expectation and our framework may perform worse than state-of-the-art *without backstopping*. We show how to boost our expected work bounds to work bounds with high probability by running $O(\log(T))$ independent instances of our algorithm. Furthermore, we show how to "backstop" the algorithm by observing that we can compose two dynamic algorithms A and B to get one algorithm that, asymptotically, achieves the minimum amortized runtime of both A and B. We compose our framework with a traditional fully dynamic algorithm using our backstop procedure to get a composite algorithm that has runtime comparable to the state-of-the-art.

- *(Handling unknown T):* T is often unknown in real-life sequences. We give another "guess-and-double" procedure in Algorithm C.3 that allows us to handle *unknown* T, with high probability, by guessing an upper bound on T in successive powers of 2.

The only use of randomness in our framework is for preprocessing our inputs and for constructing our random partition-tree. Implications of using randomness, and potential avenues to derandomize the framework are discussed in Appendix C.6.

Reducing Incremental to Divide-and-Conquer Using an incremental algorithm, we show that our framework can be used to design algorithms for a stronger setting, in which elements are inserted fully arbitrarily, and our algorithm only receives predictions about deletion times. That is, when an element is inserted, it is tagged with a predicted deletion time. Utilizing our offline framework, we begin by using the incremental algorithm to design an offline dynamic algorithm for this problem. Our previous theorem shows that we can lift such an algorithm to the setting where we are given predictions for both insertion and deletion events. We then make two main observations that allow us to drop the need for predictions of insertion events.

- In our original offline to fully-dynamic reduction, we prepare a tentative schedule of events during preprocessing. This is no longer possible, because we do not have access to any predicted information before the algorithm starts. However, we achieved this using an *online* algorithm. Thus, we can use the online algorithm to create the schedule, as we learn about new events.

- In our original offline to fully-dynamic reduction, we run a tentative version of the divide-and-conquer algorithm during preprocessing. This is no longer possible, again because we do not have access to the predicted events. We observe that, because of the specific structure of the offline algorithms that arise from incremental algorithms, we can run the computations associated with the nodes of the divide-and-conquer tree using a *just-in-time* approach.

This extension of our framework is described in detail in Appendix D.

Reducing Decremental to Divide-and-Conquer We also apply our framework to lifting decremental algorithms to the fully-dynamic setting, given predictions of insertion events. We do this via a simple reduction to the incremental case. In particular, we reinterpret a decremental algorithm over elements as an incremental algorithm over "anti-elements." Then, we are able to directly apply the previous result. This is described in detail in Appendix E.

Appendix C. Missing Proofs and Additional Discussion from Section 2: Offline Divide-and-Conquer to Fully Dynamic Transformation

C.1. Preprocessing Proofs

In this section, we provide the proofs for preprocessing that we omitted in the main body of our paper. We first recall Lemma 2.1.

Lemma 2.1 (Initial Scheduling Quality and Runtime) *Let* $\mathbf{p}$ *be the vector of predictions, mapping each event to a day in* $[T]$*. Let* $\mathbf{r}$ *be the (unknown to the algorithm) true vector of real events, mapping each event to the day in* $[T]$ *that it actually occurs.*

Given $\mathbf{p}$*, we can compute in time* $O(T\log^*(T))$ *an assignment vector* $\mathbf{a}_0$ *such that* $\mathbf{a}_0$ *assigns at most one insertion event and at most one deletion event to each day, assigns all insertions events before deletion events, and has error*

$$\mathbf{E}\left[||\mathbf{a}_0 - \mathbf{r}||_1\right] = O\left(||\mathbf{p} - \mathbf{r}||_1 \cdot \log(T)\right).$$

Proof We iterate through the events in P in arbitrary order; the reason is that the online metric matching algorithm can handle requests that occur in any order. At each predicted event, we match it to an available day assignment, where only one event can be scheduled on each day. Then, this problem becomes an instance of online metric matching, where the predicted events are requests, and there is one available server at each day. In particular, our problem instances are over the line metric.

We implement the harmonic algorithm for online metric matching on line metrics, due to Gupta and Lewi Gupta and Lewi (2012). In this algorithm, a request is matched to either its closest open server on the left, or the closest open server on the right. Let d_{left} be the distance from the request to the closest open server to the left, and d_{right} be the distance from the request to the closest open server on the right. Then the request is assigned to the left server with probability $\frac{1/d_{\text{left}}}{1/d_{\text{left}}+1/d_{\text{right}}}$, and to the right server otherwise.

We note that this algorithm can be implemented in amortized $O(\log^*(T))$ time per update, as shown in Algorithm C.1. We maintain a union-find data structure that keeps track of contiguous blocks of assigned servers where initially each $t \in [T]$ is assigned to a different disjoint set. For each block, we maintain metadata with the first open server to the left and the first open server to the right. Then, when a new request arrives, finding the closest open servers can be done by looking up the request's representative element in the union-find data structure. One request is assigned per arrival, so at most three sets must be merged for each arrival. Since union-find can be implemented with amortized $O\left(\log^*(T)\right)$ update time, and we use a constant number of calls to union-find (with constant additional work), we can implement this algorithm with amortized $O\left(\log^*(T)\right)$ update time.

Now we argue that it meets our correctness guarantees. The harmonic algorithm is $O(\log(T))$ competitive, for T requests. Let $\mathbf{r}$ be the vector of the real update days of each event, $\mathbf{p}$ be the predicted update times of each event, and $\mathbf{a}_0$ be the assigned update days of each event. We have

$$\begin{aligned}||\mathbf{a}_0 - \mathbf{r}||_1 &\leq ||\mathbf{a}_0 - \mathbf{p}||_1 + ||\mathbf{p} - \mathbf{r}||_1 \\ &\leq O(\log(T)) \cdot ||\mathbf{c} - \mathbf{p}||_1 + ||\mathbf{p} - \mathbf{r}||_1,\end{aligned}$$

where $\mathbf{c}$ is the feasible assignment with the lowest ℓ_1 distance to $\mathbf{p}$, in hindsight. Since we know that $\mathbf{r}$ is also a feasible assignment, we have that

$$||\mathbf{c} - \mathbf{p}||_1 \leq ||\mathbf{p} - \mathbf{r}||_1$$

Thus,

$$||\mathbf{a}_0 - \mathbf{r}||_1 \leq O(\log(T)) \cdot ||\mathbf{p} - \mathbf{r}||_1.$$

Finally, to resolve the ordering constraints among events on the same element e, we iterate through all of our assigned days and maintain a set of all elements for which we have seen a deletion event but not an insertion event. Suppose that for element e we see an insertion event on day $t_{e,i}$ *after* its deletion event on day $t_{e,d}$, we reassign the day for the deletion event to $t_{e,i}$. If there are multiple deletion events that occur before an insertion event, we reassign each deletion event to the next available insertion event and to $T+1$ otherwise. Since we are guaranteed at most one event per day before our reassignment of events and our assignment procedure ensures that we reassign at most one deletion event for each element e to its corresponding insertion event, the maximum number of events assigned to each day $t \in [T]$ is two and there is at most one insertion event and one deletion event assigned to t. We produce a new vector $\mathbf{a}_0'$ after this final processing.

We now prove that our new $\mathbf{a}_0'$ also satisfies our error bounds. We show the following inequality since all insertion events are assigned the same days as $\mathbf{a}_0$ and only deletions are potentially reassigned:

$$||\mathbf{a}_0' - \mathbf{r}||_1 \leq ||\mathbf{a}_0 - \mathbf{r}||_1 + ||\mathbf{a}_{0,\text{del}}' - \mathbf{r}_{\text{del}}||_1,$$

where $\mathbf{a}_{0,\text{del}}'$ and $\mathbf{r}_{\text{del}}$ are the assigned and real days, respectively, of the deletion events. We now show that $||\mathbf{a}_{0,\text{del}}' - \mathbf{r}_{\text{del}}||_1 \leq ||\mathbf{a}_0 - \mathbf{r}||_1$ through several cases. Let $\hat{t}_{i,e,1} < \hat{t}_{i,e,2} < \cdots$ be the insertion times for element e in $\mathbf{a}_0$, $\hat{t}_{d,e,1} < \hat{t}_{d,e,2} < \cdots$ be the deletion times for element e in $\mathbf{a}_0$, $t_{i,e,1} < t_{i,e,2} < \cdots$ be the insertion times for element e in $\mathbf{r}$, and $t_{d,e,1} < t_{d,e,2} < \cdots$ be the deletion times for element e in $\mathbf{r}$. We assign $t_{i,e,j} = T+1$ for any updates where we have a greater number of predicted events than real events. Then, we show, for any j:

1. If $\mathbf{\hat{t}_{i,e,j}} < \mathbf{t_{i,e,j}} < \mathbf{t_{d,e,j}}$ or $\mathbf{t_{i,e,j}} < \mathbf{\hat{t}_{i,e,j}} < \mathbf{t_{d,e,j}}$, then $|t_{d,e,j} - \hat{t}_{i,e,j}| < |\hat{t}_{d,e,j} - t_{d,e,j}|$.

2. Otherwise, if $\mathbf{t_{i,e,j}} < \mathbf{t_{d,e,j}} < \mathbf{\hat{t}_{i,e,j}}$, then $|t_{d,e,j} - \hat{t}_{i,e,j}| < 2 \cdot |\hat{t}_{i,e,j} - t_{i,e,j}|$.

Hence, via the casework above, we show that

$$||\mathbf{a}_0' - \mathbf{r}||_1 \leq 2 \cdot ||\mathbf{a}_{0,\text{ins}} - \mathbf{r}_{\text{ins}}||_1 + ||\mathbf{a}_{0,\text{del}} - \mathbf{r}_{\text{del}}||_1 \leq 2 \cdot ||\mathbf{a}_0 - \mathbf{r}||_1,$$

where $\mathbf{a}_{0,\text{ins}}$ and $\mathbf{r}_{\text{ins}}$ are the assigned and real days, respectively, of the insertion events. This proof can be extended beyond insertions and deletions to any constant number of ordering constraints of events on elements. Such a reassignment to produce $\mathbf{a}_0'$ requires $O(T)$ additional work. ■

Lemma 5 (Partition-Tree has $O(\log(T))$ Depth) *The depth of a random partition-tree drawn over T days is $O(\log(T))$ in expectation. Furthermore, for any constant $k \geq 36$, the depth is*

$$\leq k \ln(T) \qquad \textit{with probability} \qquad \geq 1 - T^{-\frac{k}{24}}.$$

Proof Consider an arbitrary day $t \in [T]$ which is a leaf in the partition-tree as defined in Definition 2. We analyze the depth of the leaf containing t in the partition-tree for every $t \in [T]$. Consider the path from the root of the partition-tree to this leaf, that is the path of all windows containing t.

Consider some non-leaf window W of size S that contains t; the size of window W is a random variable with value S conditioned on a set of events. We know that with probability at least $\frac{1}{2}$, the random partition of W results in both children of W having size at most $\frac{3S}{4}$. We call this the "good event." (Consider the center $\lfloor \frac{S}{2} \rfloor$ elements of W. There are strictly more dividers that border one of these center elements than there are dividers that do not border these elements. If any of these dividers is chosen, the smaller child has size at least $\frac{S}{4}$, thus both children have size at most $\frac{3S}{4}$.)

Good events can occur at most $\log_{\frac{4}{3}}(T) = \frac{\ln(T)}{\ln \frac{4}{3}} < 4\ln(T)$ times over the windows on the root-to-leaf path to t before there is one element left and the path ends.

Now we bound the probability that t's path in the partition-tree has length more than $k\ln(T)$, for some constant $k \geq 36$. For this to occur, it is necessary that in the partitioning for the first $k\ln(T)$ windows on t's path, the good event occurred fewer than $4\ln(T)$ times.

We can bound the probability that good events occur fewer than $4\ln(T)$ times with the Chernoff-Hoeffding inequality. Furthermore, for this analysis, we assume that the number of windows along t's path is at least $k\ln(T)$; otherwise, we have our desired property. Let

$$X = X_1 + \cdots + X_{k\ln(T)},$$

where X_j is the indicator random variable that the good event occurred at the jth window of i's path. We have $\mathbf{E}[X_j] \geq \frac{1}{2}$ and $\mathbf{E}[X] \geq \frac{k\ln(T)}{2}$. Furthermore, each good event is independent of

```
Algorithm C.1: Fast Randomized Online Matching
Input: Predicted sequence of updates $P$.
Output: Sequence of updates $\overline{P}$ with at most two events on each day.
30 Initialize a union-find data structure over $T = \max\left(|P|, \max_{(e,type,t,i)\in P}(t)\right)$ elements with procedures FIND and UNION. All $t \in [T]$ are in disjoint sets.
   for $t \in T$ do
31     $\mathbf{left}_t := t - 1$ if $t > 1$, null else.
       $\mathbf{right}_t := t + 1$ if $t < T$, null else.
       $\mathbf{assigned}_t :=$ false
32 for event $E = (e, t) \in P$ do
33     $t' := \text{FIND}(t)$
       $d_{\text{left}} := t - \mathbf{left}_{t'}$
       $d_{\text{right}} := \mathbf{right}_{t'} - t$
       goleft := true with probability $\frac{1/d_{\text{left}}}{1/d_{\text{left}} + 1/d_{\text{right}}}$, false else.
       if goleft then
34         $\mathbf{assigned}_{\mathbf{left}_{t'}} :=$ true
           if $\textit{assigned}_{(\textit{left}_{t'} - 1)}$ then
35             $\text{UNION}(\mathbf{left}_{t'} - 1, \mathbf{left}_{t'})$
               $\text{UNION}(\mathbf{left}_{t'}, t)$
               $t'' := \text{FIND}(t)$
               $\mathbf{left}_{t''} := \mathbf{left}_{(\mathbf{left}_{t'} - 1)}$
               $\mathbf{right}_{t''} := \mathbf{right}_{t'}$
36         else
37             $\text{UNION}(\mathbf{left}_{t'}, t)$
               $t'' := \text{FIND}(t)$
               $\mathbf{left}_{t''} := \mathbf{left}_{t'} - 1$
               $\mathbf{right}_{t''} := \mathbf{right}_{t'}$
38     else
39         if $\textit{assigned}_{(\textit{right}_{t'} + 1)}$ then
40             $\text{UNION}(\mathbf{right}_{t'} + 1, \mathbf{right}_{t'})$
               $\text{UNION}(\mathbf{right}_{t'}, t)$
               $t'' := \text{FIND}(t)$
               $\mathbf{left}_{t''} := \mathbf{left}_{t'}$
               $\mathbf{right}_{t''} := \mathbf{right}_{(\mathbf{right}_{t'} + 1)}$
41         else
42             $\text{UNION}(\mathbf{right}_{t'}, t)$
               $t'' := \text{FIND}(t)$
               $\mathbf{left}_{t''} := \mathbf{left}_{t'}$
               $\mathbf{right}_{t''} := \mathbf{right}_{t'} + 1$
```

previous good events. Thus,

$$\begin{aligned}
\mathbf{Pr}\left[X \le \mathbf{E}[X] - t\right] &\le e^{-\frac{t^2}{3\mathbf{E}[X]}} \\
\mathbf{Pr}\left[X \le 4\ln(T)\right] &= \mathbf{Pr}\left[X \le \mathbf{E}[X] - (\frac{k}{2} - 4)\ln(T)\right] \\
&\le e^{-\frac{((\frac{k}{2}-4)\ln(T))^2}{\frac{3k\ln(T)}{2}}} \\
&= e^{-(\frac{k}{6}-\frac{8}{3}+\frac{32}{3k})\ln(T)} \\
&\le e^{-\frac{k}{12}\ln T} && \text{for } k \ge 28 \\
&= T^{-\frac{k}{12}}.
\end{aligned}$$

This shows that any fixed day t has depth at most $k\ln T$ in the random partition-tree, with probability at least $1 - T^{-\frac{k}{12}}$.

Now, we can take a union bound over the T days, and say that all T days have depth at most $k\ln T$ in the random partition-tree with probability

$$\ge 1 - T\cdot T^{-\frac{k}{12}} = 1 - T^{-\frac{k}{12}+1} \ge 1 - T^{-\frac{k}{24}}.$$

To bound the expected depth of the tree, we can multiply the probability of the depth being low by the depth, and, otherwise, multiply the maximum possible depth of T by the remaining probability. Using the above bound for $k = 96$, we get

$$\mathbf{E}[\text{depth}] \le 96\ln T\cdot(1 - T^{-4}) + T\cdot T^{-4} = O(\log(T)).$$

■

Lemma 6 (Work to Maintain Schedule) *Algorithm 2.1 performs at most $O(T\log T)$ reschedules to the predictions of updates and maintains the predicted schedule of events over the course of the online dynamic updates.*

Proof Lemma 2.1 guarantees that at the end of preprocessing, we have at most two events scheduled per day. Call this assignment of events $\mathbf{a}_0$. Over the course of the algorithm, events are rescheduled via a doubling search procedure using PROCESSEVENTLATERTHANPREDICTION. For an event e, call this sequence of reschedulings $\mathbf{a}_1(e), \mathbf{a}_2(e), \ldots$, where $\mathbf{a}_i(e) = \mathbf{a}_0(e) + 2^i$, if it exists. Thus the set of existent $\mathbf{a}_i(e)$, for event e, is some subset of the events in $\mathbf{a}_0$, shifted by exactly 2^i. Thus, each $\mathbf{a}_i$ contributes at most a constant number of scheduled events on each day.

Now, we note that since there are only T possible days, an event can only be rescheduled $O(\log T)$ times. Thus, $\mathbf{a}_i(e)$ only exist for $i \in [k\cdot\log T)]$ for some constant $k \ge 1$. Together, this results in at most $O(\log T)$ events (re)scheduled for any given day. There are T days, resulting in $O(T\log T)$ reschedules.

Now, consider the work done by our algorithm to maintain the schedule on a given day t. The algorithm must potentially reschedule the prediction of one real event for day t using PROCESSEVENTEARLIERTHANPREDICTION, and reschedule (via the guess and double procedure in PROCESSEVENTLATERTHANPREDICTION) the predicted events that were scheduled for day t but did not occur. Rescheduling a predicted event using PROCESSEVENTEARLIERTHANPREDICTION is done once per real event; since there are T real events, this procedure requires $O(T)$ reschedules. By our previous accounting, there are at most $O(T \log T)$ events that are rescheduled by PROCESSEVENTLATERTHANPREDICTION. Hence, our algorithm does $O(T \log T)$ reschedules in total. ■

C.2. Work of RETRIGGER

Lemma 8 (Expected Work of RETRIGGER Operation) *Provided a c-divide-and-conquer algorithm (Definition 3), the expected work of* RETRIGGER(t_1, t_2) *is bounded by*

$$\widetilde{O}\left(|t_2 - t_1| \cdot T^{c-1}\right) \quad \textit{for } c > 1, \qquad \textit{and } \widetilde{O}\left(|t_2 - t_1|\right) \quad \textit{for } c \leq 1.$$

Proof Consider a call to RETRIGGER(t_1, t_2). We first show the following observation that our algorithm given in Algorithm 2.1 assigns at most $O(\log(T))$ (real or predicted) events to each day. First, by Lemma 2.1, after preprocessing, we assign at most two predicted events to each day. Then, predicted events become reassigned by either PROCESSEVENTEARLIERTHANPREDICTION or PROCESSEVENTLATERTHANPREDICTION. An event gets processed by PROCESSEVENTEARLIERTHANPREDICTION when a real event happens on a day earlier than the prediction. By the guarantee that at most one real event occur on any day, at most one predicted event can be moved to a day $t \in [T]$ by PROCESSEVENTEARLIERTHANPREDICTION. Then, PROCESSEVENTLATERTHANPREDICTION reassigns a predicted event to a later day when an event occurs later than predicted. By our procedure for reassigning such events, we keep track of a counter i for each predicted event, denoting the number of times it has been reassigned. Because $\mathbf{a}_0'$ contains at most 2 events per day and PROCESSEVENTLATERTHANPREDICTION reassigns events in increments of powers of 2, each day $t \in [T]$ gets assigned at most two events at a distance of 2^i away for all $i \in [\log(T)]$. Hence, the number of events assigned to any day is $O(\log(T))$. We call this set of events, on day t, a ***batch*** of events and denote it as $\mathcal{B}_t$.

First, we fix the size of the smallest window containing $[t_1, t_2]$ and let $S = |W_{t_1,t_2}|$. Now, we consider the work done in any one level ℓ of the subtree rooted at W_{t_1,t_2}. Let the windows in level ℓ of the subtree have sizes $S_1, \dots, S_k$. If $c > 1$, the work done at this level can be bounded by

$$\sum_{i=1}^{k} S_i^c \leq \sum_{i=1}^{k} S_i \cdot T^{c-1} = ST^{c-1},$$

since $S_i \leq T$. If $c \leq 1$, the work of the subtree rooted at W_{t_1,t_2} is dominated by the root, and so, the work done at level ℓ is bounded by

$$\sum_{i=1}^{k} S_i \leq S.$$

By Lemma 5, if we set $k = 24(c+2)$, we get that the depth of the random partition-tree is $24(c+2) \cdot O(\log(T))$ over T days with probability $\geq 1 - T^{-(c+2)}$ (since c is a fixed constant).[10] Conditioning on this event, and accounting for the $\log(\log(T \cdot |\mathcal{S}|))$ blowup from using persistent memory, we get that the total work is bounded by

$$O\left(S \cdot T^{c-1} \log(T) \log(\log(T \cdot |\mathcal{S}|))\right) \quad \text{when } c > 1 \tag{1}$$
$$O\left(S \log(T) \log(\log(T \cdot |\mathcal{S}|))\right) \quad \text{when } c \leq 1, \tag{2}$$

10. We can choose to set k to be an arbitrarily large constant if we want to increase the probability of success.

where the extra $\log(T)$ comes from multiplying our work per level by $O(\log(T))$ levels.

Each day corresponds to a batch of $O(\log T)$ events, so there are at most $|t_2 - t_1| \log T$ events between t_1 and t_2. By Lemma 7, we can bound the expected size of $S = |W_{t_1,t_2}|$ to be $\mathbf{E}[S] = O(|t_2 - t_1| \log^2(T))$. Thus, if we condition on the event that the depth of the partition-tree is $O(\log(T))$, then the expected work, using Eqs. (1) and (2), denoted by $\text{work}(W_{t_1,t_2})$, is given by

$$\mathbf{E}[\text{work}(W_{t_1,t_2}) \mid \text{ partition-tree has depth } O(\log T)] = \tag{3}$$

$$\begin{cases} O(|t_2 - t_1| \cdot T^{c-1} \log^3(T) \log(\log(T \cdot |\mathcal{S}|))) & \text{when } c > 1 \\ O(|t_2 - t_1| \cdot \log^3(T) \log(\log(T \cdot |\mathcal{S}|))) & \text{when } c \leq 1. \end{cases} \tag{4}$$

In the case where the tree is not $O(\log T)$ depth, which happens with probability at most $T^{-(c+2)}$, we have the trivial bound that the tree can be depth at most T. Thus, the work over the subtree can be bounded as

$$\mathbf{E}[\text{work}(W_{t_1,t_2}) \mid \text{ partition-tree } \textit{does not} \text{ have depth } O(\log T)] =$$

$$\begin{cases} O(|t_2 - t_1| \cdot T^{c} \log^3(T) \log(\log(T \cdot |\mathcal{S}|))) & \text{when } c > 1 \\ O(|t_2 - t_1| \cdot T \log^3(T) \log(\log(T \cdot |\mathcal{S}|))) & \text{when } c \leq 1. \end{cases}$$

We can also trivially bound that $|t_2 - t_1| \leq T$, allowing us to trivially bound the expected work by

$$\mathbf{E}[\text{work}(W_{t_1,t_2}) \mid \text{ partition-tree } \textit{does not} \text{ have depth } O(\log T)] = \tag{5}$$

$$\begin{cases} O(T^{c+1} \log^3(T) \log(\log(T \cdot |\mathcal{S}|))) & \text{when } c > 1 \\ O(T^2 \log^3(T) \log(\log(T \cdot |\mathcal{S}|))) & \text{when } c \leq 1. \end{cases} \tag{6}$$

Using the two cases given in Eqs. (4) and (6) allows us to bound the expected work over this subtree by

$$\begin{aligned} \mathbf{E}[\text{work}(W_{t_1,t_2})] &= \mathbf{E}[\text{work}(W_{t_1,t_2}) \mid \text{depth } O(\log T)] \cdot \mathbf{P}[\text{depth } O(\log T)] \\ &+ \mathbf{E}[\text{work}(W_{t_1,t_2}) \mid \textit{not} \text{ depth } O(\log T)] \cdot \mathbf{P}[\textit{not} \text{ depth } O(\log T)] \\ &= (1 - T^{-(c+2)}) \cdot O((|t_2 - t_1| \cdot \log(T)) \cdot T^{c-1} \log^3(T) \log(\log(T \cdot |\mathcal{S}|))) \\ &+ T^{-(c+2)} \cdot O(T^{c+1} \log^3(T) \log(\log(T \cdot |\mathcal{S}|))) \\ &\quad = O(|t_2 - t_1| \cdot T^{c-1} \log^3(T) \log(\log(T \cdot |\mathcal{S}|))) \qquad \text{when } c > 1 \\ \\ &= (1 - T^{-(c+2)}) \cdot O((|t_2 - t_1| \cdot \log(T)) \cdot \log^3(T) \log(\log(T \cdot |\mathcal{S}|))) \\ &+ T^{-(c+2)} \cdot O(T^2 \log^3(T) \log(\log(T \cdot |\mathcal{S}|))) \\ &\quad = O(|t_2 - t_1| \cdot \log^3(T) \log(\log(T \cdot |\mathcal{S}|))) \qquad \text{when } c \leq 1. \end{aligned}$$

This proves our desired bounds. ■

Lemma 9 (Expected Work Over All Calls to RETRIGGER) *The expected work done by Algorithm 2.1 over all calls to* RETRIGGER *is*

$$\widetilde{O}(|\mathbf{p}-\mathbf{r}|_1 \cdot T^{c-1}) \quad \text{when } c \geq 1, \qquad \text{and } \widetilde{O}(|\mathbf{p}-\mathbf{r}|_1) \quad \text{when } c < 1.$$

Proof First, consider a single event e. Let $\mathbf{a}_0(e)$ be the day that e is assigned to after the preprocessing step (Lemma 2.1). Let $\mathbf{r}(e)$ be the true day on which e occurs.

If $\mathbf{r}(e) \leq \mathbf{a}_0(e)$, then RETRIGGER$(\mathbf{r}(e), \mathbf{a}_0(e))$ is called exactly once for e. Otherwise, there is a sequence of reassignments, following a doubling search procedure, over $\mathbf{a}_1(e), \ldots, \mathbf{a}_k(e)$, where $\mathbf{a}_{i+1}(e) = 2^i + \mathbf{a}_i(e)$, $\mathbf{a}_i(e) \leq \mathbf{r}(e)$ for $i < k$, and the final $\mathbf{a}_k(e) \geq \mathbf{r}(e)$. RETRIGGER is called for each of these reassignments: RETRIGGER$(\mathbf{a}_i(e), \mathbf{a}_{i+1}(e))$ for $i = 0, \ldots, k-1$, and finally called one last time as RETRIGGER$(\mathbf{r}(e), \mathbf{a}_k(e))$. Thus, by linearity of expectation, the total expected work done is

$$\left[\sum_{i=0}^{k-1} \text{work}(\text{RETRIGGER}(\mathbf{a}_i(e), \mathbf{a}_{i+1}(e)))\right] + \text{work}(\text{RETRIGGER}(\mathbf{r}(e), \mathbf{a}_k(e))). \tag{7}$$

Assume for simplicity we are in the $c > 1$ case. (The $c \leq 1$ analysis simply doesn't contain the T^{c-1} term.) We use Lemma 8 to bound Equation (7) by

$$\begin{aligned}
&\left[\sum_{i=0}^{k-1} O\left(|\mathbf{a}_{i+1}(e) - \mathbf{a}_i(e)| \cdot T^{c-1} \log^3(T) \log\log(T \cdot |\mathcal{S}|)\right)\right] \\
&\qquad + O\left(|\mathbf{a}_k(e) - \mathbf{r}(e)| \cdot T^{c-1} \log^3(T) \log\log(T \cdot |\mathcal{S}|)\right)\big) \\
&\leq \left(\sum_{i=0}^{k-1} |\mathbf{a}_{i+1}(e) - \mathbf{a}_i(e)| + |\mathbf{a}_k(e) - \mathbf{r}(e)|\right) \cdot O\left(T^{c-1} \log^3(T) \log\log(T \cdot |\mathcal{S}|)\right) \\
&= O\left(|\mathbf{a}_0(e) - \mathbf{r}(e)| \cdot T^{c-1} \log^3 T \log\log(T \cdot |\mathcal{S}|)\right).
\end{aligned}$$

The final inequality follows by first observing that

$$|\mathbf{r}(e) - \mathbf{a}_k(e)| \leq |\mathbf{r}(e) - \mathbf{a}_0(e)|, \tag{8}$$

by the nature of the doubling search. This is because $\mathbf{a}_k(e)$ only exists if $\mathbf{a}_{k-1}(e) < \mathbf{r}(e)$, and $\mathbf{a}_{k-1}(e) = \frac{1}{2}(\mathbf{a}_0(e) + \mathbf{a}_k(e))$.

This means that

$$\begin{aligned}
\sum_{i=0}^{k-1} |\mathbf{a}_{i+1}(e) - \mathbf{a}_i(e)| = |\mathbf{a}_k(e) - \mathbf{a}_0(e)| \\
&\leq |\mathbf{a}_k(e) - \mathbf{r}(e)| + |\mathbf{r}(e) - \mathbf{a}_0(e)| \\
&\leq 2|\mathbf{a}_0(e) - \mathbf{r}(e)| && \text{by (8)}
\end{aligned}$$

So in total,

$$\sum_{i=0}^{k-1} |\mathbf{a}_{i+1}(e) - \mathbf{a}_i(e)| + |\mathbf{a}_k(e) - \mathbf{r}(e)| \leq 3|\mathbf{a}_0(e) - \mathbf{r}(e)| = O(|\mathbf{a}_0(e) - \mathbf{r}(e)|).$$

This bounds the expected cost of calls to RETRIGGER related to the specific event e. To the bound the work over all events e, recall we denote the vector of input predictions as $\mathbf{p}$. Then we can bound the total work by

$$\begin{aligned}
&\sum_e O\left(|\mathbf{a}_0(e) - \mathbf{r}(e)| \cdot T^{c-1} \log^3(T) \log\log(T \cdot |\mathcal{S}|)\right) \\
&= |\mathbf{a}_0 - \mathbf{r}|_1 \cdot O\left(T^{c-1} \log^3(T) \log\log(T \cdot |\mathcal{S}|)\right) \\
&= O\left(|\mathbf{p} - \mathbf{r}|_1 \cdot T^{c-1} \log^4(T) \log\log(T \cdot |\mathcal{S}|)\right) \qquad \text{by Lemma 2.1, when } c > 1.
\end{aligned}$$

Using the same analysis and Lemma 8, we can bound the total work by

$$O\left(|\mathbf{p} - \mathbf{r}|_1 \cdot \log^4(T) \log\log(T \cdot |\mathcal{S}|)\right)), \qquad \text{when } c < 1.$$

■

C.3. Backstops and Boosting to High Probability

In this section we describe how to compose N dynamic algorithms $\{A_1, \dots, A_N\}$, that achieve amortized guarantees leading to a framework that achieves both *competitiveness* and our desired work bound *with high probability*. Below, we define *one computation step* as one word operation in the word-RAM. The reduction is straightforward, and relies heavily on the fact that the goal is an amortized bound, and not a worst-case update time: we run the algorithms "in parallel," and we return the output of the algorithm that terminates first. We show our desired work bounds in Theorem C.1.

```
Algorithm C.2: Backstop Meta-Algorithm for $N$ Algorithms
Input: Event sequence $E = (e_1, e_2, \dots)$
Output: Sequence of completed computations
43 Initialize algorithms $\{A_1, \dots, A_N\}$
   Initialize event buffers for algorithms $\{A_1, \dots, A_N\}$

44 for day $t$, event $e_t$ occurs do
                                                        // occurs online
45     Add $e_t$ to event buffers for $\{A_1, \dots, A_N\}$
       repeat
46         Iteratively perform one computation step of each algorithm in $\{A_1, \dots, A_N\}$
47     until one algorithm has completed computation for all events in its buffer
48     Output the completed computation
```

Theorem C.1 (Best-of-All-Worlds Backstop) *Fix N algorithms $\{A_1, \dots, A_N\}$, that will see the same, a priori unknown, input sequence over T days. Define $R_{A_i}(t)$ to be the total work that A_i does over the first t days. We can design a meta-algorithm M (Algorithm C.2), such that*

$$\forall t, \quad R_M(t) = O\left(N \cdot \min\{R_{A_1}(t), \dots, R_{A_N}(t)\}\right),$$

where $R_M(t)$ is the total work that M does over the first t days.

Proof At a high level, our meta-algorithm will update all of the algorithms in synchrony, but it will perform computations at the rate of the faster algorithm. Thus, the faster algorithm will be up-to-date with the current timestep and will provide the output computation; each slower algorithm may have a backlog of computations that it has not yet performed. This is described in Algorithm C.2.

We fix a day t and analyze the work done by the meta-algorithm up through the t-th day. Line 45 does $O(t \cdot N)$ work over t updates. We can assume that $R_{A_i}(t) = \Omega(t)$ for all of the algorithms A_i (where the algorithms perform $\Omega(1)$ processing per update), and thus the contribution of $O(t \cdot N)$ to the work will be dominated by faster growing terms.

Line 46 maintains the invariant that each of the algorithms $A_1, \ldots, A_N$ has been run for the same number of computation steps. Thus, the first algorithm to complete the computation for all of events in its buffer after the t-th update will be precisely the algorithm A_i that minimizes $R_{A_i}(t)$. Over the course of all updates up to t, the meta algorithm has run A_i for

$$R_{A_i}(t) = \min\{R_{A_1}(t), \ldots, R_{A_N}(t)\} \text{ steps.}$$

Since it has run each of the N algorithms for the same number of steps, it has done total work

$$O\left(N \cdot \min\{R_{A_1}(t), \ldots, R_{A_N}(t)\}\right).$$

Correctness of the algorithm follows because a solution output on day t is the solution output by A_i upon seeing exactly the stream of events up to day t. Thus, if A_i is correct, the meta-algorithm is also correct. ■

C.3.1. OBTAINING WORK BOUNDS WITH HIGH PROBABILITY

In this section, we show how to modify our partition-tree-based algorithm using our backstop meta-algorithm to obtain our running time bounds with high probability, using a simple "boosting" argument. We first make the observation that if we run $O(\log(T))$ independent instantiations of our algorithm, at least one instantiation will have runtime close to the expectation with high probability.

We can also remove the assumption that T is known, by using an additional guess-and-double argument; such an argument comes in handy for our incremental or decremental transformation to fully dynamic in later sections.

This allows us to compose $O(\log(T))$ independent copies of our algorithm to achieve our expected runtime bound with high probability. The following lemma follows from a standard boosting argument, which is given formally in Appendix C.3.

Lemma 13 (Boosting Argument) *Consider $k \log(T)$ independent instantiations of the predicted-updates algorithm (Algorithm 2.1), for some constant integer $k > 0$. With probability at least $1 - \frac{1}{T^k}$, at least one of these instantiations has runtime*

$$\widetilde{O}\left(T^{c-1} \cdot (T + ||\mathbf{p} - \mathbf{d}||_1)\right) \text{ when } c > 1, \qquad \text{and } \widetilde{O}\left(T + ||\mathbf{p} - \mathbf{d}||_1\right) \text{ when } c \leq 1.$$

Now, using the backstop technique, we take $O(\log(T))$ instantiations of the predicted updates algorithm (Algorithm 2.1) to create a composite algorithm that has runtime that scales with the minimum of the instantiations.

Proof Consider a single instantiation of the predicted-updates algorithm given in Algorithm 2.1. By Lemmas 4 and 9, the total expected work of this algorithm (including preprocessing) is

$$O\left(T^{c-1} \cdot (T + ||\mathbf{p} - \mathbf{d}||_1) \cdot \log^{c+3}(T) \cdot \log\log(T \cdot |\mathcal{S}|)\right) \quad \text{when } c > 1, \tag{9}$$

$$O\left((T + ||\mathbf{p} - \mathbf{d}||_1) \cdot \log^{4}(T) \cdot \log\log(T \cdot |\mathcal{S}|)\right) \quad \text{when } c \leq 1. \tag{10}$$

By Markov's inequality, the probability that the runtime of a single instantiation exceeds two times the expected work is at most $\frac{1}{2}$. Thus, the probability that $k\log(T)$ independent instantiations all simultaneously exceed two times the expected work is

$$\leq \left(\frac{1}{2}\right)^{k\log(T)} = \frac{1}{T^k}.$$

Thus, with probability at least $1 - \frac{1}{T^k}$, one of the $k\log(T)$ instantiations has runtime at two times the expected work given in Eqs. (9) and (10). ■

C.4. Final Theorems

Algorithm C.3: Work Bounds with High Probability

Input: *Offline:* Partition-tree $\mathcal{T}$, algorithm $\mathcal{A}$, predicted sequence P, ground set $\mathcal{S}$. *Online:* Update sequence $U = [E_1, \ldots, E_T]$, prediction sets $P_1, \ldots, P_{\log_2(T)}$.

Output: After each day $t \in [T]$, output $f(U_t)$.

```
// Initialization
49 $\widehat{T} \leftarrow 1$
   $L \leftarrow k \cdot \log(\widehat{T})$ for some fixed constant $k > 0$
   for real event $E_t$ do
50     if $t \geq \widehat{T}$ then
           // Guess-and-double
51         Obtain prediction set $P_{\max(1,\log_2(\widehat{T}))}$
               $\widehat{T} \leftarrow 2 \cdot \widehat{T}$
               $L \leftarrow \max(k \cdot \log(\widehat{T}), \log(|\mathcal{S}|))$ for sufficiently large $k > 0$[11]
               Create $\mathcal{A} := [A_1, \ldots, A_L]$, $L$ independent instantiations of Algorithm 2.1 using $\widehat{T}$ and
               $P_{\max(1,\log_2(\widehat{T}))}$ // Instantiations use independent randomness.
52         $M \leftarrow$ Initialize backstop meta-algorithm (Algorithm C.2) over $\mathcal{A}$
            Use $M$ to process all previously seen events $E_1, \ldots, E_{t-1}$ in buffer $B$
53     Add $E_t$ to $B$
        Pass $E_t$ to $M$, return output of $M$
```

Theorem 2.1 (Offline Divide-and-Conquer to Fully Dynamic with Predictions Reduction) *Given a c-divide-and-conquer algorithm $\mathcal{A}$ that computes the solution to a problem $\mathcal{P}$, sets of predictions $P_1, P_2, \ldots, P_{\log_2(T)}$, and online sequence of events $U = [E_1, \ldots, E_t]$, an improved Algorithm 2.1 correctly outputs the solution to $f(U_t)$ after each $t \in [T]$ and uses total work, with high probability,*

$$\widetilde{O}\left(T^{c-1} \cdot (T + ||\mathbf{p} - \mathbf{u}||_1)\right) \text{ when } c > 1, \qquad \text{and } \widetilde{O}\left(T + ||\mathbf{p} - \mathbf{u}||_1\right) \text{ when } c \leq 1.$$

11. The $\log(|\mathcal{S}|)$ term is necessary to ensure enough copies with high probability.

Furthermore, our algorithm never performs worse than the best-known fully dynamic algorithm for problem $\mathcal{P}$. Here, $\mathbf{p}$ is our predicted sequence of days and $\mathbf{u}$ is the real sequence.

Proof Given our inputs, we construct our set of random partition-trees using Algorithm 2.1. We first prove that our algorithm correctly outputs $f(U_t)$ after every event E_t in the sequence of online events. Algorithm C.3 runs L instantiations of Algorithm 2.1 and outputs the answer from the first instantiation that finishes processing event E_t. By our procedures in Algorithm 2.1, every real update is processed in every window that contains it, and no windows in the union of all windows which do not contain days after t contain any predicted events that have *not* occurred after event E_t is processed. Furthermore, by Definition 3, all windows process all events irrespective of the order of the events. Hence, Algorithm 2.1 correctly returns $f(U_t)$ for every $t \in [T]$.

We now give our high probability bound for the total amount of work performed over all T real events. Consider a day $t = 2^i$ on which Line 50 is triggered, and $\widehat{T}$ is doubled. $\widehat{T}$ is newly set to be 2^{i+1}. Consider the total work done by the new instantiation of M. Specifically, the work between the event at $t = 2^i$ and when $\widehat{T}$ is doubled again at event $t' = 2^{i+1}$. By Theorem C.1, we can bound the work of M by L times the work of the minimum $A_i \in \mathcal{A}$ that is drawn independently in this iteration. Lemma 13 bounds the work of that minimum A_i by

$$\begin{aligned} \widetilde{O}\left(T^{c-1} \cdot (T + ||\mathbf{p} - \mathbf{d}||_1)\right) \quad &\text{when } c > 1, \\ \widetilde{O}\left(T + ||\mathbf{p} - \mathbf{d}||_1\right) \quad &\text{when } c \leq 1, \end{aligned}$$

with probability $\geq \max(1 - \frac{1}{t^k}$ if $\log(t) > \log(|\mathcal{S}|)$ and $\geq 1 - \frac{1}{t^{k \cdot \log(|\mathcal{S}|)}}$, otherwise. The probability is due to the fact that we create $L = \max(k \cdot \log(T), k \cdot \log(|\mathcal{S}|))$ independent instantiations.

Now, can use a union bound to conclude that our work bounds hold for all integers $t \in [T]$ with probability

$$\begin{aligned} &\geq 1 - \left(\sum_{i=0}^{\infty} \frac{1}{2^{k(i+1)}} + \frac{\log(T)}{2^{k \cdot \log |\mathcal{S}|}}\right) \\ &\geq 1 - \frac{2\log(T)}{|\mathcal{S}|^k}, \end{aligned}$$

which is high probability for sufficiently large constant $k > 0$.

This condition allows us to, for a day t, bound the total work done by Algorithm C.2 up through day t. Define i such that $2^i \leq t < 2^{i+1}$. We can bound the work done by Algorithm C.2 up through day t as

$$\sum_{i=0}^{\lfloor \log_2(t) \rfloor} O\left(|\mathbf{p} - \mathbf{r}|_1 \cdot (2^i)^{c-1} \log^{c+3}(2^i) \log\log(2^i)\right) = O\left(|\mathbf{p} - \mathbf{r}|_1 \cdot (t)^{c-1} \log^{c+3}(t) \log\log(t \cdot |\mathcal{S}|)\right),$$

when $c > 1$,

and $O\left(|\mathbf{p} - \mathbf{r}|_1 \cdot \log^4(t) \log\log(t)\right)$ when $c \leq 1$. Substituting T for t into the equations results in our desired work.

Finally, by composing Algorithm C.3 with the best-known algorithm for $\mathcal{P}$, once again using Theorem C.1, we never perform worse than the best-known algorithm for the problem. ■

Corollary 10 (Bound for Update/Query problems) *Given a c-divide-and-conquer algorithm $\mathcal{A}$ for $f(\cdot)$ with worst-case query time* $\mathrm{query}(A)$*, we can construct an algorithm, such that with high probability with respect to T, that has total work*

$$\widetilde{O}\left(T^{c-1}\cdot(T+||\mathbf{p}-\mathbf{d}||_1)\right) \text{ when } c>1, \qquad \text{and } \widetilde{O}\left(T+||\mathbf{p}-\mathbf{d}||_1\right) \text{ when } c\leq 1,$$

and the worst-case query time is always bounded by $\mathrm{query}(A)$. *Furthermore, our algorithm never performs worse than the best-known fully dynamic algorithm for problem $\mathcal{P}$.*

Proof For the first part of the corollary, we can think of an update/query problem as simply being an update problem, where after each update, the algorithm must return a data structure that is compatible with the query algorithm. Using Theorem 2.1 on this update problem, we get a predicted-dynamic algorithm that always returns a data structure that is compatible with the original query algorithm. Thus, the query algorithm is unchanged.

For the second part of the corollary, we use the backstop procedure of Theorem C.1 to backstop the predicted-deletion algorithm with the fully-dynamic algorithm. This provides some data structure for every update. However, our offline to online query algorithm and the fully-dynamic query algorithm, $\mathrm{query}(B)$, may not be expecting the same data structure. Thus, to execute a query on a given day, we run, in parallel, both query algorithms. Thus our query time is bounded by $O\left(\min\{\mathrm{query}(A), \mathrm{query}(B)\}\right)$. ■

C.5. Connection to Metric Tree Embeddings

The local error guarantee of our random partition-tree essentially provides a randomized scheme to embed a particular line metric where the points are at integer increments, into a tree metric of low depth. Precisely, we want a low-depth partition-tree such that, for two points a, b, the size (number of descendant leaves) of the lowest common ancestor of a and b is of size $O(|a-b|\cdot\log T)$ in expectation, where T is the number of points in the metric space. This is morally the same as finding a low-depth tree embedding of this metric space. One direction is seen by setting edge lengths of a low-depth partition-tree such that for a parent node p and its child node c, $\mathrm{dist}(p,c)=\frac{1}{2}(|p|-|c|)$. Thus, the distance between two leaves is precisely the size of the lowest common ancestor in the partition-tree, and this gives a low-depth tree embedding of the space. The other direction is seen by noting that for any embedding of this particular metric space into a low-depth hierarchically well-separated tree (HST), e.g. Fakcharoenphol et al. (2004), the distance between two points a, b in the tree is at least the size of the lowest common ancestor in the tree, and thus the HST gives us a low-depth partition-tree. A line metric is, of course, itself a tree metric, and one can therefore find a "tree embedding" with no distortion. However, for our algorithm, we require an embedding into a tree metric that is low-depth. This can also be achieved by other distributions on trees, e.g. the classic tree-embedding scheme of Fakcharoenphol et al. (2004).

C.6. Discussion on the Use of Randomness

One implication of the way we use randomness in our algorithmic framework is that we *do not* expect this framework to be robust to the strongest form of an *adaptive adversary* where the adversary has access to the random bits used within our algorithm. (Although in many learning-augmented models, the definition of an adaptive adversary is often also not immediately clear.) Thus, it is an interesting question whether such a result can be derandomized, as a deterministic algorithm must be robust to an adaptive adversary.

There are two places where we use randomness in our predicted-deletion framework.

(1) Random partition-tree: this allows us to argue that in effect that two days that are close in time, correspond to leaves in the partition-tree that are close in the tree, in expectation.

(2) Online metric matching: the randomized harmonic algorithm for online metric matching can be implemented quickly.

We discuss challenges and potential avenues to derandomize these components, along with a strategy that is open to an adaptive adversary.

Random partition-tree. The main benefit of a random partition-tree is that it preserves lengths in expectation. That is, Lemma 7 tells us that for $a, b \in [T]$, the expected size of the lowest common ancestor of a and b in a random-partition-tree drawn over $[T]$ is $O(|b-a| \log T)$.

It is worth noting that this guarantee does *not* hold with high probability. In fact, the probability that the size of the lowest common ancestor is T (the root node), is $\frac{|b-a|}{T-1}$ (i.e. this is the probability that the first divider chosen in the tree falls in the range $[a, b]$).

This indicates that this expectation bound is morally fulfilling two functions: it is both avoiding some bad cases, and implicitly accounting for some amortization. To illustrate this, consider the following examples.

Example 1 (Uneven error density) *Let T be 2^i for some even i. Consider $\sqrt{T}$ elements that are added on days $1, 2, \ldots, \sqrt{T}$ with predicted deletion times*

$$T/2+1, T/2+2, \ldots, T/2+\sqrt{T}.$$

Their actual deletion times will be

$$T/2+1-\sqrt{T}, T/2+2-\sqrt{T}, \ldots, T/2.$$

The contribution to the ℓ_1 prediction error is $\sqrt{T}$ for each of $\sqrt{T}$ elements, for a total of T.

The deterministic perfectly balanced partition-tree will make the first split between $T/2$ and $T/2+1$. So the lowest common ancestor of the predicted deletion time and the actual deletion time for each of these elements is the root node. Thus, each of these early deletion events will trigger $O(T)$ recomputation, for a total of $O(T\sqrt{T})$ work. (We are being informal about the exact logarithmic factors here, for illustrative purposes.)

The randomized tree, on the other hand, guarantees that the expected work of triggering recomputations for these events is $O(T \log T)$. This is only a $\log T$ factor off from the actual ℓ_1 error, as opposed to the deterministic tree that is a $\sqrt{T}$ factor off.

Example 1 illustrates that if there is a portion of the sequence of days that is has a relatively high density of error that coincides with a "deep split" of our tree, then the work of our algorithm could be high. The random partition-tree allows us to avoid this by placing the "splits" of the tree randomly.

Now, we consider the counterpart problem.

Example 2 (Even error density) *Let $T = 2i$ for some i. Label the days $\{1, \ldots, T\}$ Consider i elements that such that element e_j for $j \in \{0, \ldots, i-1\}$ is inserted on day $2j+1$ with predicted deletion $2j+5$. e_j's actual deletion time is $2j+2$. The last elements e_{i-2} and e_{i-1} alone will have predicted deletion on the actual deletion days of $T-2$ and T, respectively.*

The contribution to the ℓ_1 prediction error is 3 for each of $\frac{T}{2}-2$ elements, for a total error of $\frac{3T}{2}-6 \in O(T)$.

In this example, except for the first and last possible divider, every other divider splits some actual deletion time from its corresponding deletion time. This says that in a deterministic partition-tree, and in most instances of a randomized partition-tree, there is some error of size 3 that triggers $O(T)$ recomputation.

While this is a large blow up, this does not occur for *most* of the errors, and specifically for Example 2 one can show that both the deterministic tree and the randomized tree will trigger $O(T)$ recomputation total, which is comparable to the size of the error, $O(T)$. This demonstrates the implicit amortization that happens when we consider the *expected* cost associated with a specific error in the randomized partition-tree.

These two examples demonstrate that a deterministic scheme could likely achieve the amortization of the random partition-tree and handle inputs like Example 2. The challenge lies in designing a deterministic scheme that can avoid splitting areas of high error density, as in Example 1.

This also suggests a strategy that an adaptive adversary could use against a random partition-tree that would break the guarantee, even if the adversary does not have direct access to the random tree. An adversary could learn about the structure of the tree from seeing the output solutions of the framework. As an example, consider a divide-and-conquer algorithm that computes some greedy solution over permanent elements. From seeing the daily outputs of the divide-and-conquer algorithm, the adversary could learn about the order in which the elements were given to the divide-and-conquer algorithm. This is highly informative about what level of the tree each element becomes permanent at, and since the adversary knows the input, they learn about where the windows of each level end. Then, to make the algorithm incur high cost, the adversary could choose to delete elements just before "deep splits" in the tree.

Online metric matching. For this framework, we want an algorithm for online metric matching on the line metric that achieves a logarithmic (or polylogarithmic) competitive ratio, and can be implemented in polylogarithmic update time.

In the online metric matching problem, the algorithm is given a set of "servers". Then, "requests" arrive online and report their distances (cost to be matched) to each of the servers. For each request that arrives, the algorithm must immediately match it irrevocably to one of the servers, and that server may not be matched to any future requests. The objective function is the total cost of the final matching. The distance function must be consistent with some metric space.

The online metric matching problem was introduced independently by Khuller, Mitchell, and Vazirani Khuller et al. (1991), and by Kalyanasundaram and Pruhs Kalyanasundaram and Pruhs (1993). Both of these works show a deterministic $(2n-1)$ competitive algorithm, where n is the number of points in the metric space. This is also optimal for deterministic algorithms, there is no deterministic algorithm that achieves competitive ratio better than $(2n-1)$ in all metric spaces. It is possible to circumvent this lower bound using randomization. The best known randomized algorithm for general metric spaces is due to Bansal, Buchbinder, Gupta, and Naor, and achieves a competitive ratio of $O(\log^2 n)$ Bansal et al. (2007).

For line metrics specifically, it is possible to do better. On the randomized side, Gupta and Lewi provide multiple randomized algorithms achieving competitive ratio $O(\log n)$ Gupta and Lewi (2012), one of which we use in our construction. On the deterministic side, Raghvendra shows that the deterministic *robust matching algorithm* works well in a variety of settings, and in particular achieves competitive ratio $O(\log n)$ for line metrics Raghvendra (2016, 2018). In terms of competitive ratio, these results are close to optimal, as Peserico and Scquizzato show that no algorithm, deterministic or randomized, can achieve competitive ratio $o(\sqrt{\log n})$ for the line metric Peserico and Scquizzato (2021).

From the perspective of our framework, we could hope to use Raghvendra's robust matching algorithm to achieve the desired competitive ratio deterministically. However, it is not immediately clear how to implement this algorithm with polylogarithmic update time. The algorithm maintains an offline matching that informs the choices for the online matching. The offline matching is maintained by finding an augmenting path on the associated flow network on each step. Without additional assumptions, finding such an augmenting path can take $O(n^2)$ work. If we were to use the robust matching algorithm as is, it would actually be the dominating factor of our runtime. However, the robust matching algorithm as stated can be applied to any metric space. It is an interesting question of whether, for the line metric specifically, the robust matching algorithm could be implemented much faster.

We note that it is possible to implement deterministic algorithms for our setting that achieve slightly relaxed guarantees. In particular, the greedy algorithm (always assigns a request to the nearest available server) achieves the following guarantee for our problem.

Lemma 14 (Greedy matching guarantee) *Given an element insertion e with predicted deletion time t_{report}, we assign e greedily to the nearest free time slot $\widehat{t_{report}}$. Such an algorithm produces an assignment of elements to time slots such that*

$$\left(\sum_{i=1}^{t} |t^i_{report} - \widehat{t^i_{report}}|\right) \leq 4t \cdot error_{max} \cdot \log(T) + T \cdot error_{max},$$

where $error_{max}$ is the minimum ℓ_∞ error of any feasible solution. That is, over all possible assignments $t_{baseline}$ of deletion times, such that $t^i_{baseline} \neq t^j_{baseline}$ for $i \neq j$,

$$error_{max} = \min_{t_{baseline}} \max_i \left|t^i_{baseline} - t^i_{report}\right|.$$

This is proven in Appendix C.7. Note that we leverage the integral structure of our particular problem to get this guarantee (i.e. the requests and server are at integral positions, and there is exactly one server per day.) This algorithm can be implemented quickly using a union-find data structure in the same way as the harmonic algorithm (see proof of Lemma 2.1.)

In particular, if we were to use this in place of randomized metric matching in our framework, we would achieve expected total work

$$O\left(T\left(1 + ||\mathbf{p} - \mathbf{d}||_\infty \log^2 T\right) \cdot \text{update}(A) \cdot \log^2 T \cdot \log\log(|\mathcal{S}|T)\right),$$

and the only remaining randomized component would be the random partition-tree. Note that $T \cdot ||\mathbf{p} - \mathbf{d}||_\infty$ is always at least $||\mathbf{p} - \mathbf{d}||_1$, and in many instances is much larger.

C.7. Deterministic ℓ_∞-based bound

In this section we show that the greedy allocation algorithm has competitive ratio bounded by a function of the minimum ℓ_∞ norm of any solution.

Let the maximum error between any t_{report} and t_{delete} be denoted as $error_{\text{max}}$. We show that our greedy strategy of assigning each element update event prediction to the nearest free time slot obtains total error at most $t \cdot error_{\text{max}}$ where t is the total number of updates. Each slot can be taken by an event at most $error_{\text{max}}$ away time-steps away. Thus, this means that given any two time slots

t_2 and t_1 where $t_2 > t_1$, it holds that the number of events in $[t_1, t_2]$ is at most $t_2 - t_1 + 1 + 2 \cdot error_{\max}$. Using this observation, we prove the following lemma about the maximum error by our assignment of events to the timeline.

Lemma 14 (Greedy matching guarantee) *Given an element insertion e with predicted deletion time t_{report}, we assign e greedily to the nearest free time slot $\widehat{t_{report}}$. Such an algorithm produces an assignment of elements to time slots such that*

$$\left(\sum_{i=1}^{t} |t^i_{report} - \widehat{t^i_{report}}|\right) \leq 4t \cdot error_{max} \cdot \log(T) + T \cdot error_{max},$$

where $error_{max}$ is the minimum ℓ_∞ error of any feasible solution. That is, over all possible assignments $t_{baseline}$ of deletion times, such that $t^i_{baseline} \neq t^j_{baseline}$ for $i \neq j$,

$$error_{max} = \min_{t_{baseline}} \max_i \left| t^i_{baseline} - t^i_{report} \right|.$$

Proof We first prove the observation that between any two time slots t_2 and t_1, the maximum number of updates with predicted times in $[t_1, t_2]$ is $t_2 - t_1 + 1 + 2 \cdot error_{\max}$. This is the case because in the true update sequence, the at any time slot t' at most $error_{\max}$ updates on the left of t' can have t_{report} equal to t' and at most $error_{\max}$ updates on the right of t' can have t_{report} equal to t'. Now, we make the following potential argument. Give each update $4error_{\max} \cdot \log(T)$ coins when we assign the update to a time slot. Each time we greedily assign a new update to the nearest free time slot, we take a coin from each update that we *pass on the way to the new assigned time slot*. We then need to argue that no update runs out of coins.

For any time slot t', an update passes t' if the shortest distance to a free slot includes t'. We consider two contiguous segments of time of size 2^i for all $i \in [\log_2(T)]$ where t' is on the boundary. In other words, we consider $[t' - 2^i - 1, t']$ and $[t', t' + 2^i + 1]$. We consider such contiguous segments of time because only additional updates in the range $[t' - 2^{i-1}, t']$ (symmetrically $[t', t' + 2^{i-1}]$) will cross t' if there are no free slots in $[t' - 2^i - 1, t' - 2^{i-1}]$ (symmetrically $[t', t' + 2^{i-1}]$. Otherwise, if there are free slots in those segments, then the updates will fill up those free slots first. By our argument above, for any such contiguous segment of time, there are at most $2error_{\max}$ additional updates that must be assigned. Hence, in the contiguous segment of time $[t' - 2^i - 1, t']$ at most $2error_{\max}$ updates are assigned and these additional updates may pass t'. The same holds for the contiguous segment of time $[t', t' + 2^i + 1]$. Thus, a total of at most $4error_{\max}$ updates pass through t' for each $i \in [\log_2(T)]$ and we do not use more than $4error_{\max} \cdot \log_2(T)$ coins from t'. The additional $error_{\max}$ error is for the initial error of the prediction. ■

Appendix D. Incremental to Fully-Dynamic Transformation

In this section we show how our framework that lifts offline algorithms to the fully-dynamic setting, given predictions of all update times, can be adapted to lift an incremental algorithm to the fully-dynamic setting, given predictions of only the deletion times. Formally, we consider the following model.

Definition 15 (Predicted-Deletion Dynamic Model) *In the predicted-deletion dynamic model, we consider a ground set $\mathcal{S}$. On each day exactly one of the following occurs:*

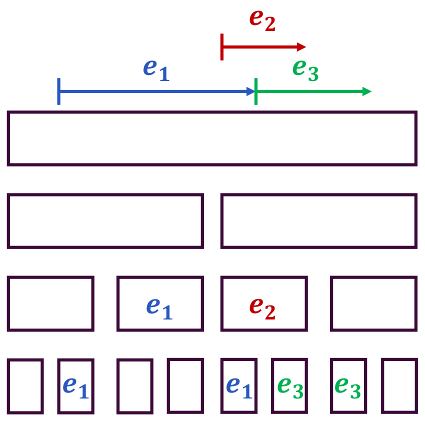

Figure 3: An illustration of which windows consider the elements e_1, e_2, and e_3 permanent

1. *An element* $e \in \mathcal{S}$ *is inserted, and reports a* prediction *of the day on which it will be deleted.*

2. *A previously inserted element is deleted.*

An algorithm computes a function $f(\cdot)$ *in the predicted-deletion dynamic model, if on every day* t*, the algorithm outputs* $f(S)$*, where* $S \subseteq \mathcal{S}$ *is the* working subset *induced by the true (not predicted) sequence of element insertions and deletions that occur in time-steps* $1, \ldots, t$.

This model is stronger than the general predicted-updates model that was presented in the last section. In the previous section, we charged the runtime of the fully-dynamic algorithm to the prediction error in insertions and deletions. In this model, the runtime of the fully-dynamic algorithm should only depend on the predictions of deletion times. This corresponds to the fact that we are starting with an incremental algorithm rather than an offline algorithm. Thus, the algorithm we start with can already handle arbitrary insertions, and our reduction adds functionality only in handling deletions.

First, in what may seem like a step in the wrong direction, we show how to build an *offline dynamic* divide-and-conquer algorithm, using an incremental algorithm with a worst-case guarantee. To do this, we introduce the concept of permanent elements.

Definition 16 (Permanent elements) *An element is* permanent *with respect to a window* W *of a partition-tree if*

1. *the element is inserted on or before the first day of* W*,*

2. *the item is deleted after the last day of* W*,*

3. *and the element is* not *permanent for any ancestor of* W.

This is illustrated in Figure 3. Note that for a day t, the windows containing t perfectly partition the elements that are active on day t. Conversely, we also have that the windows for which an element e is permanent perfectly partition the lifetime of e. This suggests a natural way to convert an (online) incremental algorithm, $\mathcal{A}$, with worst-case update time $\mathrm{update}(\mathcal{A})$ into a divide-and-conquer offline algorithm.

Lemma 17 (Offline-dynamic divide-and-conquer from incremental) *An incremental algorithm, $\mathcal{A}$, with worst-case update time $\mathrm{update}(\mathcal{A})$, can be converted into a offline-dynamic divide-and-conquer algorithm satisfying Definition 3, where the work at each window, W, is*

$$O(\mathrm{update}(\mathcal{A}) \cdot |W|).$$

Proof At the root node, we initialize the state of the incremental algorithm. Then, at each window, we insert the permanent elements associated with that window into the state of the incremental algorithm. The set of permanent elements can be computed from the set of all elements that are active at any time during the parent window, in a linear-time scan. Thus, the work done at a window scales with $\mathrm{update}(\mathcal{A})$ times the number of permanent edges associated with that window.

Note that any element that is permanent for a window W, must have either been inserted or deleted during W's sibling window. Otherwise, this element would actually be permanent for W's parent window. Thus, the number of permanent elements of W is bounded by the size of Ws sibling. For a random partition-tree, this is the same as the size of W in expectation. So this meets the third criterion of Definition 3, where the expected work at a node is linear in the size of the node, with a multiplicative factor of $\mathrm{update}(\mathcal{A})$.

With respect to the second criterion of Definition 3, we need that the computation of the window depends only on the state of the parent window, and the *unordered set* of updates occurring in this window. At first glance, this does not appear to be the case, as calculating the set of permanent elements requires the algorithm to compare the set of updates of its sibling window. However, we can consider the set of updates of the parent window to be included in the state of the parent window. Then, since the number of updates of the parent is, in expectation, a constant factor more than the size of the child window, the child window can reconstruct the necessary information from the state of the parent and its own set of updates.

Thus, this algorithm meets the criteria of Definition 3, where the expected work at each window W is

$$O(\mathrm{update}(\mathcal{A}) \cdot |W|).$$

■

Now, if we directly apply Theorem 2.1, we recover an algorithm that does total work

$$O((T + ||\mathbf{p} - \mathbf{d}||_1) \cdot \mathrm{update}(\mathcal{A}) \cdot \log^3(T) \log\log(T \cdot |\mathcal{S}|)),$$

when given predictions of *all updates* at the outset of the algorithm. However, we wish to consider a setting in which the elements arrive arbitrarily, and an element's predicted deletion time is only provided when the element is inserted.

This leads to two main issues. The first is that we cannot create a tentative schedule of events during preprocessing, because we have no information about the events that are going to occur. The second is that we cannot compute a preliminary version of the divide-and-conquer algorithm during preprocessing, because we do not have a tentative schedule of events to refer to. To address the first issue, we make the following observation.

Observation 18 (Online scheduling) *Even in the setting where the initial schedule of events is computed statically in the preprocessing step, we use an* online *algorithm to assign events to days.*

Thus, as predicted events arrive over time, we can assign each event irrevocably to an initial tentative schedule using this online algorithm, while maintaining the runtime and approximation guarantees of the original algorithm.

Thus, we can schedule events as they arrive, without having to reschedule events to account for later arrivals.

To address the second issue, we show that, while we don't know enough about the schedule to run the entire divide-and-conquer algorithm during preprocessing, we actually can utilize a "just in time" approach.

Lemma 19 (Windows can be computed "just in time") *Consider a offline-dynamic divide-and-conquer algorithm of the form constructed by Lemma 17. Then, the fully-dynamic algorithms with predictions resulting from Theorem 2.1 can be run in a "just in time" fashion where*

1. *the computation associated with a window W beginning on day t is only run on or after day t, and*
2. *on day t, we have enough information to run the computation associated with all windows starting on day t, and*
3. RETRIGGER *is never called for an insertion event.*

Proof First, we note that the output of the fully-dynamic algorithm resulting from Theorem 2.1 on day t only depends on the computations associated with windows that contain t. Thus, the first point of the lemma is true for *any* algorithm of this form. That is, on a day t, we can consider all windows starting after day t to be "inactive," and ignore them when they are part of a RETRIGGER operation. Then, on day t, we run the computations of any windows beginning exactly on day t, for the first time.[12]

For the second point, consider a window W beginning on some day t. It is possible that on day t, there are some events that will occur during W, for which we do not have predicted times. For example, insertions occur entirely arbitrarily, so we do not have any information about the insertions that will occur during W. However, any element that could be permanent for W, must *already* be inserted. This is by definition: for an element to be permanent for W, it must be inserted on or before the first day of W. Thus, on day t, we have enough information to compute all of the permanent elements of W, which is enough to run the divide-and-conquer algorithms constructed by Lemma 17.

Finally, for the third point, note that any window that depends on an insertion event e is only computed for the first time after e occurs. Thus, at this point, e is fully known, and will not be rescheduled, so it will never call RETRIGGER. ■

Altogether, Lemma 17, Observation 18, and Lemma 19, along with Theorem 2.1 give us the following theorem.

12. In fact, while we don't get a better asymptotic runtime bound, running any algorithm of this form in a "just in time" way can only reduce the total work done, as we do not have to retrigger windows that are not yet active.

Theorem 20 (Incremental to Fully-Dynamic) *Given an incremental algorithm, $\mathcal{A}$, with expected worst-case update time* $\text{update}(\mathcal{A})$*, we can construct an algorithm in the predicted-deletion model (Definition 15), such that the total expected work done by the algorithm is*

$$O\left(\text{update}(\mathcal{A}) \cdot (T + ||\mathbf{p} - \mathbf{d}||_1) \cdot \log^4 T \log\log(T \cdot |\mathcal{S}|)\right),$$

where $||\mathbf{p} - \mathbf{d}||_1$ is the ℓ_1 error of the deletion-time predictions.

Using Theorem 2.1, we obtain the following work, with high probability.

Corollary 11 (Incremental to Fully-Dynamic Transformation) *Given an incremental algorithm, $\mathcal{A}$, with expected worst-case update time* $\text{update}(\mathcal{A})$*, we can construct an algorithm in the predicted-deletion model (Definition 15) such that the total work done by the algorithm is*

$$\widetilde{O}\left(\text{update}(\mathcal{A}) \cdot (T + ||\mathbf{p} - \mathbf{d}||_1)\right)$$

with high probability, where $||\mathbf{p} - \mathbf{d}||_1$ is the ℓ_1 error of the deletion-time predictions.

D.1. Comparison to concurrent work of Peng and Rubinstein (2023); van den Brand et al. (2023)

The independent and concurrent work of van den Brand et al. (2023) also provides a reduction that lifts an incremental algorithm to the fully dynamic setting given predictions of deletion times. In this section, we overview their approach and provide a comparison to our reduction.

The reduction of van den Brand et al. (2023) is based on a simple and elegant observation over the previous work of Peng and Rubinstein (2023). In Peng and Rubinstein (2023), they consider a model that they call the *deletion look-ahead model*. In this model, on every day t, the relative deletion order of all elements that are active in the system is known. Another way to think of this model is that, when an element is inserted, it announces which of the existing elements it will be deleted after. This is similar to our predicted-deletion model in the regime where the prediction error is 0. However, it is actually more general, because it only requires *ordinal* information rather than *cardinal* information about the deletion times.

In the language of our work, we can interpret the Peng and Rubinstein (2023) reduction as doing something similar to constructing the offline divide-and-conquer algorithm from Lemma 17, and running it with a just-in-time approach (in the manner of Lemma 19). This is all with respect to a fixed deterministic partition tree (i.e. a perfectly balanced binary tree). Indeed, if we had access to cardinal information (i.e. the actual day on which each edge is predicted to be deleted), this would suffice to recover an amortized guarantee for this problem.

To adapt this to the ordinal setting, Peng and Rubinstein (2023) make the following observation. Over the course of any window W, at most $|W|$ elements can be deleted. Thus, at the start of a window W, any element that is *not* in the $|W|$ earliest-to-be-deleted elements will certainly be permanent over W,[13] and can safely be inserted. At the end of the computation associated with W, there are still up to $|W|$ elements that we did not insert, because we were unsure of whether or not they would be permanent. These elements are passed to the child windows, which process these elements, along with any elements that were inserted in their sibling windows. In all, the total

13. Here, we are abusing our notation of permanent elements from earlier, and referring to an element as permanent if it is inserted on or before the first day of W, and deleted on or after the last day of W.

number of elements that any window has to process is at most a constant factor more than the size of the window.

This adaptation to the ordinal setting leaves the reduction of Peng and Rubinstein (2023) with a very useful property: the state of the incremental algorithm at each leaf of the partition tree has elements inserted in *approximately reverse deletion order*. That is, any element that will be deleted in the next d deletions, was one of the last $c \cdot d$ elements to be inserted, where c is a constant.

Now, in the case where the predicted deletion order is subject to error, van den Brand et al. (2023) observe that this property allows us to fix the data structure with low overhead. In particular, consider an element e that is deleted earlier than predicted. That is, on day t, e was predicted to be the d-th next element to be deleted, but instead it was the first element to be deleted. The approximate reverse deletion order property tells us that e was in the last $c \cdot d$ elements to be inserted into this leaf. This means that it was inserted in some window W relatively close to leaf-level in the partition-tree. Thus, to fix the data structure, they only need to retrigger these few windows that are descendants of W.

Essentially, this approximate reverse deletion order property allows the data structure to be fixed on the fly, with low overhead. Furthermore, their strategy does *not* require a random partition-tree: indeed it is fully deterministic, and can even be adapted to provide a worst-case per update guarantee (as opposed to our work which provides an amortized guarantee).

In comparison to our work, it is not clear if it is possible to extend the Peng and Rubinstein (2023); van den Brand et al. (2023) approach to lift offline algorithms to the fully dynamic setting. In particular, we provide a simple analogue of their approach that can convert a decremental algorithm to the fully-dynamic setting, when given predictions of insertion times. We explore this connection formally in the next section (Appendix E). With respect to the specific reduction of van den Brand et al. (2023), it essentially maintains state of a decremental algorithm, where elements are deleted in *approximately reverse insertion order*.

While this approach can handle predicted deletion times given an incremental algorithm, and predicted insertion times given a decremental algorithm, it is not clear if and how these two guarantees can be combined. In particular, this runs into two issues.

1. It is not clear what kind of algorithm to reduce to, as one direction requires an incremental algorithm, and the other direction requires a decremental algorithm.

2. It is not clear what the analogue of maintaining elements in both reverse insertion *and* reverse deletion order should be.

Our work addresses the first point by making the key observation that we can reduce to an offline divide-and-conquer algorithm. As to the second point, we resolve the issue by using a random partition-tree. This does not require us to maintain any ordering of the elements that we consider.

Thus, the reduction of van den Brand et al. (2023) for the predicted deletion model is based on a simple and elegant extension of the previous framework of Peng and Rubinstein (2023), which lifts incremental algorithms into the deletion look-ahead setting. The reduction that we present in this work for the predicted deletion model, is instead based on a simple extension from our earlier framework that lifts offline divide-and-conquer algorithms into the predicted update setting. Under a single framework, we generalize all three settings: offline, incremental, and decremental.

Appendix E. Decremental to Fully-Dynamic Transformation

In this section, we show how to adapt our framework to convert a decremental algorithm to the fully-dynamic setting, given predictions of only the insertion times. Formally, we consider the following model.

Definition 21 (Predicted-Insertion Dynamic Model) *In the predicted-insertion dynamic model, we consider a set S of all the elements that are predicted to ever appear in the system. At the outset, the algorithm is given S, and predictions for when each of the elements in S will be inserted. Then, on each day, exactly one of the following occurs:*

1. *An element e, either in S or not, is inserted,*
2. *A previously inserted element is deleted, and provides a prediction of when it will be reinserted (if ever).*

An algorithm computes a function $f(\cdot)$ in the predicted-insertion dynamic model, if on every day t, the algorithm outputs $f(S_t)$, where S_t is the working subset *induced by the true (not predicted) sequence of element insertions and deletions that occur in time-steps $1, \ldots, t$.*

A number of works in the past have performed fully dynamic to decremental reductions, starting with the seminal work of Henzinger and King (1997), *for specific problems*. We show that, with the help of predictions, we can generalize such ideas to *any* worst-case decremental algorithm. Now, we show that, given a decremental algorithm, we can design algorithms for the predicted insertion model. We do this via a simple reduction to the predicted insertion case.

Theorem 22 (Decremental to Fully-Dynamic) *Consider a decremental algorithm, $\mathcal{A}$, which takes time $\text{initialize}_{\mathcal{A}}(T)$ to initialize a state containing up to T elements, and then has worst-case update time $\text{update}(\mathcal{A})$. Given such an algorithm, we can construct an algorithm for the predicted-insertion model (Definition 21), that does expected total work*

$$O\left(\text{initialize}_{\mathcal{A}}(T) \cdot K + \text{update}(\mathcal{A}) \cdot (T + ||\mathbf{p} - \mathbf{i}||_1 + TK) \log^4(T) \log\log(T \cdot |\mathcal{S}|)\right),$$

where $||\mathbf{p} - \mathbf{i}||_1$ is the ℓ_1 error of the insertion-time predictions for elements in S, and K is the number of elements that are not in S that are ever inserted.

Proof We reinterpret $\mathcal{A}$ as in incremental algorithm on "anti-elements." That is, suppose we initialize our algorithm to contain the elements in a set S. Then, as a decremental algorithm, the algorithm can handle updates that delete elements of S. We can also think of this as an incremental algorithm, that can handle the insertion of anti-elements, corresponding to elements in S.

Now, consider a series of updates in the predicted-insertion model (Definition 21). From the view of our decremental algorithm, each element starts as not being in the system, then is inserted at some time for which we are given a prediction, then is deleted at an unknown arbitrary time, when it predicts its next insertion.

From the view of the incremental algorithm, each anti-element starts *in* the system, then is deleted at some time for which we have a prediction, and then is reinserted again at an unknown arbitrary time, at which point it predicts its next deletion.

Thus, viewing the decremental algorithm as an incremental algorithm on anti-elements, makes it fit directly into our earlier framework for incremental algorithms, because it converts the decremental algorithm into an incremental algorithm, and it converts insertion events into deletion events and vice versa.

The only remaining issue is that this strategy cannot handle the case where an element that was never part of the predicted set S is inserted. In this case, we must add the new element to S, and run the initialization of the algorithm again from scratch. This contributes total work

$$O\left(\text{initialize}_{\mathcal{A}}(T) \cdot K\right),$$

where K is the number of such elements outside S that are ever inserted. Then, we must retrigger the entire tree, which is equivalent to this new element contributing ℓ_1-error of T. Over all such elements this contributes work

$$O\left(TK \cdot \log^4 T \log\log(T \cdot |\mathcal{S}|)\right).$$

Adding these contributions to the work bound from the incremental reduction, gives us our final bound of

$$O\left(\text{initialize}_{\mathcal{A}}(T) \cdot K + \text{update}(\mathcal{A}) \cdot (T + ||\mathbf{p} - \mathbf{i}||_1 + TK) \log^4 T \log\log(T \cdot |\mathcal{S}|)\right).$$

■

Using Theorem 2.1, we obtain the following work, with high probability.

Corollary 12 (Decremental to Fully Dynamic Transformation) *Consider a decremental algorithm, $\mathcal{A}$, which takes time* initialize$_{\mathcal{A}}(T)$ *to initialize a state containing up to T elements and has worst-case update time* update$(\mathcal{A})$. *Given such an algorithm, we can construct an algorithm for the predicted-insertion model (Definition 21) that does total work*

$$\widetilde{O}\left(\text{initialize}_{\mathcal{A}}(T) \cdot K + \text{update}(\mathcal{A}) \cdot (T + ||\mathbf{p} - \mathbf{i}||_1 + TK)\right),$$

with high probability, where $||\mathbf{p} - \mathbf{i}||_1$ is the ℓ_1 error of the insertion-time predictions of predicted elements, and K is the number of elements that were never predicted but are inserted.

We make the following remark that allows us to obtain a deterministic, worst-case decremental to fully-dynamic transformation using van den Brand et al. (2023).

Remark 23 *The above reduction is not specific to our particular implementation of the incremental to fully dynamic reduction. In particular, our interpretation of anti-elements can be applied to the incremental to fully-dynamic reduction in the concurrent independent work of van den Brand et al. (2023) and achieve their deterministic worst-case per update work bound also in the decremental setting.*

Appendix F. Applications: Offline, Incremental, and Decremental to Fully Dynamic

We apply our framework to the following problems to obtain fully dynamic algorithms in the predicted-deletion model. A summary of our runtimes compared to the best-known fully dynamic algorithms is given in Table 1. We define the problems we study below. Unless specified, the input to the following problems is an input graph $G = (V, E)$.

- **All-Pairs Shortest Paths (APSP):** For any pair of queried vertices u and v, determine the distance from u to v. In the planar diagraph APSP problem, we are given a planar, directed and weighted graph. A c-approximate APSP algorithm returns an approximation that is within c-factor of the distance between the pair.
- **Triconnectivity:** For any pair of queried vertices, determine whether the pair is *3-vertex connected*. Two pairs of vertices u and v are 3-vertex connected if and only if u and v remain connected whenever fewer than two vertices are removed.
- **Dynamic Depth-First Search (DFS) Tree:** A dynamic DFS tree is a DFS tree that is reported after each edge insertion or deletion and is a valid DFS tree for the current graph.
- **All-Pairs Maximum Flow:** For any pair of queried vertices, s and t, return the maximum flow between s and t. A c-approximate maxflow algorithm returns a flow value that is a c-factor approximation of the actual maxflow.
- **All-Pairs Minimum Cut:** For any pair of queried vertices, s and t, return the minimum cut between s and t. A c-approximate maxflow algorithm returns a cut value that is a c-factor approximation of the actual min-cut.
- **Multi-Commodity Concurrent Flow:** For a set of triples $\{(s_1, t_1, d_1), \ldots, (s_P, t_P, d_P)\}$, return the maximum value α where, concurrently for all $i \in [P]$, s_i can send $\alpha \cdot d_i$ units of flow to t_i. A c-approximate multi-commodity concurrent flow algorithm returns a value that is at least α/c.
- **Uniform Sparsest Cut:** For a given graph $G = (V, E, \mathbf{w})$, return $\Phi_G = \min_{S \subset V} \frac{\sum_{(u,v) \in E, u \in S, v \notin S} \mathbf{w(u,v)}}{|S| \cdot |V \setminus S|}$.
- **Monotone Submodular Maximization:** Given a ground set $N = [n]$ and a set function $f : N \to \mathrm{R}^+$, function f is monotone if $f(A) \geq f(B)$ for any $B \subseteq A \subseteq N$. Function f is submodular if $f(A \cup \{u\}) - f(A) \leq f(B \cup \{u\}) - f(B)$ for any $B \subseteq A \subseteq N$ and element u. Under cardinality constraint k, the problem maximizes $\max_{S \subseteq [n], |S| = k}(f(S))$ for some parameter $1 \leq k \leq n$ and under a matroid constraint $\mathcal{M}$, the problem maximizes $\max_{S \subseteq \mathcal{M}}(f(S))$. In the dynamic setting, elements can be inserted and deleted from the ground set, and the goal is to maintain $\max_{S \subseteq \mathcal{M}}(f(S))$. A c-approximate algorithm returns a set S that has value at least $c \cdot \max_{S \subseteq \mathcal{M}}(f(S))$.
- **k-Edge Connectivity:** For any pair of queried vertices $\{u, v\}$, the pair $\{u, v\}$ is k-edge connected if and only if u and v remain connected whenever any set of $k - 1$ edges are removed.
- **Minimum Spanning Forest (MST):** A dynamic minimum spanning forest is an minimum weight spanning forest that is maintained under edge insertions and deletions.

- **Strongly Connected Components (SCC)/Topological Sort:** Given a directed input graph $G = (V, E)$, maintain the strongly connected components and topological sort, respectively, under edge insertions/deletions.

Offline to Fully Dynamic In this section, we instantiate several offline-to-fully dynamic algorithms using our Theorem 2.1. Specifically, we show transformations for triconnectivity, k-edge connectivity, and minimum spanning forest using the divide-and-conquer algorithms of Chalermsook et al. (2021); Eppstein (1994); Peng et al. (2017), achieving exponential time improvements on the running times of the best-known fully dynamic algorithms when $||\mathbf{p} - \mathbf{r}||_1 = \widetilde{O}(T)$ where $\mathbf{p}$ is the predicted sequence of update times and $\mathbf{r}$ are the real update times.

We first prove that the offline algorithms given in Chalermsook et al. (2021); Eppstein (1994); Peng et al. (2017) falls under our definition of c-divide-and-conquer algorithms and specify the parameters for each algorithm. After proving these lemmas, our framework directly gives fully dynamic algorithms for these problems given predicted-updates.

Lemma 24 *There exists a c-divide-and-conquer algorithm for offline triconnecitivity Peng et al. (2017) where $c = 1$.*

Proof Peng et al. (2017) provides a divide-and-conquer algorithm on the sequence of events where each subproblem is half of its parent. In each subproblem, they produce a 3-vertex sparsifier with size equal to the number of non-permanent edges in the subproblem. The permanent edges in each subproblem are used to contract the graph to smaller sizes. For a subproblem S of size $|S|$ (where S contains the non-permanent edges), the sparsification can be done in time $O(|S|)$. Thus, Peng et al. (2017) gives a c-divide-and-conquer algorithm where $c = 1$. ■

Lemma 25 *There exists a c-divide-and-conquer algorithm for offline k-edge connectivity Chalermsook et al. (2021) for any constant $k \geq 1$ where $c = 1$.*

Proof Chalermsook et al. (2021) provides a divide-and-conquer algorithm that divides the events in approximately equal sizes. Then, in each subproblem S of size $|S|$ (where S contains the non-permanent edges), they construct a vertex sparsifier of size $\widetilde{O}(|S|)$ with the terminals being the set of queried vertices falling within the subproblem. Each sparsifier can be constructed in $\widetilde{O}(|S|)$ time; hence, Chalermsook et al. (2021) gives a c-divide-and-conquer algorithm where $c = 1$. ■

Lemma 26 *There exists a c-divide-and-conquer algorithm for offline minimum spanning forest Eppstein (1994) where $c = 1$.*

Proof Eppstein (1994) provides a divide-and-conquer algorithm that recursively divides the sequence into halves. Then, in each subproblem S where $|S|$ denotes the number of non-permanent edges, they sparsify using the permanent and non-permanent edges in the following way. They run the MST algorithm by successively picking edges in non-decreasing weight using a standard MST algorithm like Kruskal's in $O(|S| \log(|S|))$ time, *breaking ties by giving preference to non-permanent edges*. All non-permanent edges that are not picked are deleted and every picked permanent edge is contracted. This results in a sparsifier that has size $O(|S|)$ that is passed to children. Hence, Eppstein (1994) gives a c-divide-and-conquer algorithm where $c = 1$. ■

Lemmas 24 to 26 combined with Theorem 2.1 directly give the following theorem.

Theorem 27 (Predicted-Updates Algorithms) *Using our framework given in Algorithm C.3, we obtain the following fully dynamic algorithms in the predicted-updates model, assuming $||\mathbf{p}-\mathbf{r}||_1 = O(T)$, where T is the total number of updates, $\mathbf{p}$ is a vector of predicted event times, and $\mathbf{r}$ is a vector of real event times:*

1. *An algorithm for triconnectivity in $\widetilde{O}(1)$ amortized update and query times (Peng et al. (2017));*
2. *An algorithm for k-edge connectivity in $\widetilde{O}(1)$ amortized update time and query time for any constant $k \geq 1$ (Chalermsook et al. (2021)); and*
3. *A dynamic minimum spanning forest maintained in $\widetilde{O}(1)$ amortized update time (Eppstein (1994)).*

Incremental to Fully Dynamic To obtain our fully dynamic algorithms, we use the worst-case incremental algorithms given in the following works Baswana et al. (2019); Chen et al. (2018, 2020); Das et al. (2022); Feldman et al. (2022); Holm and Rotenberg (2020); Goranci (2019); Goranci et al. (2019) combined with Corollary 10 to obtain Theorem 28.

Theorem 28 (Predicted-Deletion Algorithms) *Using our framework, we obtain the following fully dynamic algorithms in the predicted-deletion model, assuming $||\mathbf{p}-\mathbf{d}||_1 = O(T)$, where T is the total number of updates, $\mathbf{p}$ is a vector of predicted deletion times, and $\mathbf{d}$ is a vector of real deletion times:*

1. *An exact all-pairs shortest path algorithm for planar directed graphs with $\widetilde{O}(\sqrt{n})$ amortized update time and $\widetilde{O}(\sqrt{n})$ worst-case query time (Chen et al. (2020));*
2. *An algorithm for triconnectivity in $\widetilde{O}(1)$ amortized update time and $O(\log^3(n))$ worst-case query time (Holm and Rotenberg (2020));*
3. *A dynamic DFS tree reported in $\widetilde{O}(n)$ amortized update time (Baswana et al. (2019); Chen et al. (2018));*
4. *A $O\left(\log^{8k}(n)\right)$-approximate maxflow/min-cut algorithm with $\widetilde{O}\left(n^{2/(k+1)}\right)$ amortized update time and reports the maxflow between any pair of vertices s and t in $\widetilde{O}\left(n^{2/(k+1)}\right)$ worst-case query time (Goranci (2019); Goranci et al. (2019));*
5. *A $O\left(\log^{8k}(n)\right)$-approximate multi-commodity concurrent flow algorithm with $\widetilde{O}\left(n^{2/(k+1)}\right)$ amortized update time and $\widetilde{O}(P^2)$ worst-case query time where P is the number of queried pairs (Goranci (2019); Goranci et al. (2019));*
6. *A $O\left(\log^{8k}(n)\right)$-approximate uniform sparsest cut algorithm with $\widetilde{O}\left(n^{2/(k+1)}\right)$ amortized update time (Goranci (2019); Goranci et al. (2019)).*
7. *A 0.3178-approximate monotone submodular maximization algorithm under a matroid constaint of rank k and makes $\widetilde{O}(\text{poly}(k, \log n))$ function evaluations per update for any $n, k > 0$ (Feldman et al. (2022)).*

Decremental to Fully Dynamic To obtain our fully dynamic algorithms, we use the worst-case decremental algorithms for strongly connected components of Roditty (2013) and topological sort (trivial) Corollary 10 to obtain Theorem 22 when $||\mathbf{p}-\mathbf{i}||_1 = O(T)$ where $\mathbf{p}$ are the predicted times for insertions, $\mathbf{i}$ is the vector of real insertion times, and T is the total number of updates.

Theorem 29 (Predicted-Insertion Algorithms) *Using our framework, we obtain the following fully dynamic algorithms in the predicted-deletion model, assuming $||\mathbf{p} - \mathbf{i}||_1 = O(T)$, where T is the total number of updates, $\mathbf{p}$ is a vector of predicted insertion times, and $\mathbf{i}$ is a vector of real insertion times:*

1. *An algorithm for maintaining strongly connected components with $\widetilde{O}(m)$ amortized update time time (matching fine-grained lower bounds of Abboud and Williams (2014)) using Roditty (2013); and*

2. *An algorithm for maintaining a topological sort in $\widetilde{O}(1)$ amortized update time (using the trivial decremental algorithm).*

Appendix G. Additional Discussion of Related Work and Connections

Sliding window and look-ahead models. In the *sliding window model*, introduced by Datar et al. (2002), the algorithm views an infinite stream of data, and must maintain a statistic over the last N data points seen (where N is the width of the window). The sliding window model has a large body of work in the streaming literature Papapetrou et al. (2015); Wei et al. (2016); Epasto et al. (2017); Braverman et al. (2020); Epasto et al. (2022); Jayaram et al. (2022); Woodruff and Zhou (2022), including Crouch et al. (2013); Reitblat (2019); Biabani et al. (2021); Alexandru et al. (2023) who give *semi-streaming* algorithms for graph problems in this model. Interestingly, despite being an inherently dynamic model, (consisting of insertions, and deletions exactly N days after insertion), it has not been studied much in the dynamic algorithms literature. A related family of models are *look-ahead models*, where a dynamic algorithm is given information about future events. Examples include graph algorithms with access to the set of vertices involved in the next few updates (but not the full sequence of edge operations) Khanna et al. (1996), and algorithms that have full access to the next few operations Sankowski and Mucha (2010).

The model that is most relevant to our work is the *deletion look-ahead model*. In this model, the algorithm maintains a statistic over a subset of some ground set of elements (e.g. a graph is a subset of possible edges), and has access to the future deletion times of all existing elements. This is a strict generalization of the sliding window model. It is known that designing an algorithm with an amortized guarantee in this model can be reduced to designing an algorithm with a worst-case guarantee in the *incremental* model, where elements are only inserted and never deleted Chan (2011); van den Brand et al. (2019). This reduction and a stronger reduction that achieves a worst-case guarantee in the known-deletion model, are formalized in Peng and Rubinstein (2023). Peng and Rubinstein (2023) actually considers a slightly more general setting in which the algorithm has access to the order in which the elements will be deleted, and not necessarily the exact deletion times. Our main contribution considers a more general model, in which information about the future is subject to error. We note that the *predicted-deletion dynamic model* that we introduce is a strict generalization of the known-deletion model, which is itself a strict generalization of the sliding window model.

Algorithms with predictions. *Algorithms with predictions*, also often called *learning augmented algorithms*, is a paradigm that has been gaining much attention in recent years. An algorithm solicits *predictions* from an untrusted source (e.g. a machine learning model) to help make decisions. The goal is to design algorithms that achieve the following three desiderata Lykouris and Vassilvitskii (2021):

(1) **(Consistency)** If the predictions are of high quality, the algorithm performs much better than a worst-case algorithm.

(2) **(Competitiveness)** If the predictions are of low quality, the algorithm does not perform any worse than a worst-case algorithm.

(3) **(Robustness)** The performance of the algorithm degrades gracefully as a function of the prediction error.

Additionally, we want to solicit predictions that can be reasonably obtained in practice. A detailed overview of the field is given in Mitzenmacher and Vassilvitskii (2021).

These desiderata present a challenge. While we would like to achieve all three for some reasonable notion of prediction, it is not a priori clear that such a guarantee is even possible. For some problems and settings, it is not indeed possible to achieve all three simultaneously, and algorithms are designed that allow the user to trade off these objectives. One example is the classic online problem of rent-or-buy, which exhibits an inherent trade-off between consistency and robustness Kumar et al. (2018); Gollapudi and Panigrahi (2019); Wei and Zhang (2020). Thus it is particularly interesting that it is indeed possible to achieve all three points for dynamic algorithms in our setting.

The setting closest to the dynamic model is the *warm start* setting. In this setting, a static algorithm is given an instance, along with some additional predicted information. Examples include graph problems Dinitz et al. (2021); Chen et al. (2022a); Davies et al. (2023), in which the prediction is a candidate solution, and the quality of the prediction is measured as the distance from the prediction to the true optimal solution. In the warm start setting, it is often the case that algorithms are able to achieve consistency and robustness simultaneously. A strong motivation for this model is time-series data, in which we want to solve a series of instances with the context that "yesterday's instance is likely not too different from today's instance." In this setting, we can think of using each day's solution as the prediction for the next day's instance as a kind of dynamic procedure. The guarantees for such a process differ from the standard dynamic model. In the dynamic model, the work done by the algorithm on a given day scales with the magnitude of the change in the instance, whereas in the warm start setting, the work scales with the magnitude of the change in the solution. In general, results of these types are incomparable. One benefit of the warm start setting is that it models a larger range of possible updates between subsequent instances. On the other hand, we cannot expect the same kinds of update times that we get for fully dynamic algorithms, since for many of these problems, even checking if a predicted solution is optimal for a fresh instance can take time linear in the size of the instance. Thus our work differs significantly from work on warm starts, both in terms of the kinds of predictions we expect, and in the kinds of guarantees we achieve.

Online algorithms are the area in which algorithms with predictions were first studied, starting with the motivating work of Kraska et al. (2018), which demonstrated that machine learned predictions could dramatically improve the efficiency of index structures, both in theory and in practice. Many online problems have since been studied in this model. Some, including rent-or-buy Kumar et al. (2018); Gollapudi and Panigrahi (2019); Wei and Zhang (2020), and problems related to caching Lykouris and Vassilvitskii (2021); Bamas et al. (2020a); Jiang et al. (2020b); Rohatgi (2020); Wei (2020); Angelopoulos et al. (2020); Indyk et al. (2022), utilize predictions of when future events occur. Others such as the secretary problem Dütting et al. (2021) and combinatorial optimization problems Bamas et al. (2020b); Lavastida et al. (2021), use predictions of what the optimal solution is. An important line of work includes scheduling and queuing problems Antoniadis et al. (2020); Lattanzi et al. (2020); Mitzenmacher (2021), in which access to predictions about parameters such as job length can allow an algorithm to circumvent strong lower bounds. This is related to our work, as our algorithm essentially schedules partial computations with access to predictions of the longevity of the edges involved. In the online setting, it is often the case that problems exhibit an inherent tradeoff between consistency and robustness, and algorithms often include a tunable parameter that allows the user to trade off these objectives.

Some sequential settings require predictions of the frequencies of certain objects in a sequence, such as Cohen et al. (2020); Jiang et al. (2020a); Chen et al. (2022b) which design streaming algorithms, and Eden et al. (2021) which minimizes sample complexity. Other settings include learning good hashes and sketches of data Mitzenmacher (2018); Vaidya et al. (2021); Indyk et al. (2019); Hsu et al. (2018), and using predicted information to maximize revenue in mechanism design settings Medina and Vassilvitskii (2017).

Appendix H. Future Directions

Extension to other problems. One follow-up direction to our work is to find more problems to which this framework can be applied. This could help design algorithms that are more efficient and/or simpler than the state-of-the-art fully-dynamic solution. Alternatively, it would be interesting to see algorithms designed for this model that look very different from this framework. A particularly interesting follow-up question is: are there problems for which the a predicted-deletion dynamic algorithm can circumvent a fully-dynamic lower bound when the prediction error is sufficiently low?

Derandomization and/or robustness to adaptive adversary. The framework presented in this work has a key reliance on randomization and is not robust to an adaptive adversary. It is an interesting question whether the framework can be derandomized, or otherwise made robust to adaptive adversaries. Potential challenges and avenues, along with vulnerabilities of the framework in this work to an adaptive adversary are discussed in depth in Appendix C.6.

Removing amortization. The framework presented in this work relies heavily on amortization to achieve the desired runtime bounds. It is an interesting question of whether problems in this model can meet a per-update runtime bound. It is also not immediately clear what role prediction error should play in such a bound. For example, one could hope to design an algorithm that, on each day t, has the update time of an incremental algorithm, with an additive factor that scales with the "error at day t." This would require an interesting definition of error in a local sense, in contrast to the global measure of error that we consider in this work (ℓ_1 error).

Lower bounds. It would be interesting to get fine-grained lower bounds for problems in this model. In particular, this framework accrues logarithmic factors in various places, including a competitive ratio for online metric matching on a line, and the expected size of the smallest node of the partition tree fully containing a subsequence $[a, b]$. For the approach in this work, some of the logarithmic factors are necessary side effects of lower bounds for these quantities. It would be interesting to see if there are indeed necessary for any approach to solve dynamic problems in this model, or if they can be circumvented by a different approach.

Implementation. A main motivation of this model is that it could potentially allow practitioners to take advantage of known structure in real-world data that classical fully-dynamic algorithms are oblivious to. It would be interesting to see how algorithms designed in this framework compare with the best algorithms and heuristics in practice.