Incremental Strongly Connected Components with Predictions

Ronald Deng
Williams College, Williamstown, MA, USA
Samuel McCauley
Williams College, Williamstown, MA, USA
Aidin Niaparast
Carnegie Mellon University, Pittsburgh, PA, USA
Helia Niaparast
Carnegie Mellon University, Pittsburgh, PA, USA
Bennett Ptak
Williams College, Williamstown, MA, USA
Shirel Quintanilla
Williams College, Williamstown, MA, USA
Shikha Singh
Williams College, Williamstown, MA, USA
Nathan Vosburg
Williams College, Williamstown, MA, USA

Abstract

Algorithms with predictions is a growing area that aims to leverage machine-learned predictions to design faster beyond-worst-case algorithms. In this paper, we use this framework to design a learned data structure for the incremental strongly connected components (SCC) problem. In this problem, the n vertices of a graph are known a priori and the m directed edges arrive over time. The goal is to efficiently maintain the strongly connected components of the graph after each insert. Our algorithm receives a possibly erroneous prediction of the edge sequence and uses it to precompute partial solutions to support fast inserts. We show that our algorithm achieves nearly optimal bounds with good predictions and its performance smoothly degrades with the prediction error. We also implement our data structure and perform experiments on real datasets. Our empirical results show that the theory is predictive of practical runtime improvements.

2012 ACM Subject Classification Theory of computation → Dynamic graph algorithms

Keywords and phrases algorithms with predictions, learning augmented algorithms, incremental graph algorithms, strongly connected components, data structures

Digital Object Identifier 10.4230/LIPIcs.SWAT.2026.17

Related Version *Full Version*: https://arxiv.org/abs/2604.26062

Funding *Aidin Niaparast*: This material is based upon work supported in part by the Air Force Office of Scientific Research under award number FA9550-23-1-0031.

1 Introduction

Incremental computation on graphs is a fundamental primitive in many database and optimization problems. In an incremental graph problem, the vertices of the graph G are known ahead of time and the edges are revealed over time. The goal is to design an efficient data structure for answering queries at time t about some property (such as connectivity or shortest path) of the current graph G_t. Designing fast *worst-case* algorithms for incremental graph problems is an area of extensive research [7, 3, 5, 12, 13].

Algorithms designed to be fast in the worst case provide strong guarantees against adversarial inputs. However, these algorithms fail to exploit the correlations in the structure of typical instances. As most computations are performed repeatedly on similar datasets, heuristics optimized for domain specific inputs tend to be faster in practice.

A growing body of work, known as algorithms with predictions [25], seeks to bridge the gap between simple heuristics that lack strong theoretical guarantees and more complex worst-case algorithms that are provably efficient but often impractical. The main idea is to use machine-learned predictions to capture correlations in the input instances and leverage them to guide algorithmic decisions.


© Ronald Deng, Samuel McCauley, Aidin Niaparast, Helia Niaparast, Bennett Ptak, Shirel Quintanilla, Shikha Singh, and Nathan Vosburg;
licensed under Creative Commons License CC-BY 4.0

Predictions are particularly effective for online or dynamic problems, where the input is not known ahead of time. Predictions trained on historical data can then serve as a proxy for future inputs – turning a hard *online* problem into an easier-to-solve *offline* one. Using such predictions, an algorithm can then precompute partial solutions ahead of time and consequently perform faster updates and queries at runtime. The algorithms designed in this framework are referred to as *learning-augmented* or *learned* algorithms.

The running time of a learned algorithm is measured in terms of the quality of the prediction – the bound is stated in terms of both the size of the input and the prediction error. Ideally, a learned algorithm for an online problem (1) is nearly as fast as the best-offline solution when the prediction is perfect, (2) has bounded worst-case performance cost when the prediction is arbitrarily wrong, and (3) has performance that degrades gracefully with error. These properties in order are referred to as *consistency, robustness*, and *smoothness* respectively in the literature. Thus, learned algorithms leverage good predictions to be fast and have provable safeguards against bad predictions.

Incremental SCC. In this paper, we study the *incremental strongly connected components (SCC)* problem in the algorithms with predictions model. In this problem, the n vertices V of a directed graph G are known a priori and the m directed edges E are inserted one by one. Let $e_1, \ldots, e_m$ denote the sequence of edges and let $G_t = (V, \{e_1, \ldots, e_t\})$ denote the graph at time t. The goal is to maintain the **strongly connected components** of G_t at all times t. A strongly connected component $S \subseteq V$ is a maximal set of vertices of G such that every pair of vertices u, v are *mutually reachable*, that is, there is a directed path from v to u and vice versa.

Maintaining SCCs is a fundamental building block in many applications, e.g. social networks [9], graph processing [34], program analysis [14] and fault-tolerant networks [2].

DFS-based approaches such as Tarjan's algorithm [31] can compute the SCCs of a static offline graph G in $O(n + m)$ time. As a step towards solving the incremental SCC problem, consider the *offline incremental problem*. In this variant, the edge insert sequence $\sigma = e_1, e_2, \ldots, e_m$ is known ahead of time and the goal is to design a data structure to answer queries about the SCCs of each intermediate graph G_t. Naively running Tarjan's repeatedly takes $O(m^2)$ time.[1] A folklore divide-and-conquer approach [28] improves upon this naive approach and solves the offline incremental SCC problem in $O(m \log m)$ total time.

Offline to Learned Incremental SCC. To design a learned algorithm for the *online incremental problem*, we assume that a prediction $\hat{\sigma}$ of the edge sequence σ is given to the algorithm. We define the **prediction error** η as the maximum error between when an edge was predicted to arrive in $\hat{\sigma}$ and its actual arrival time in σ. To ensure this metric is well-defined, we assume that $\hat{\sigma}$ is a permutation of σ (we discuss this assumption further in Section 2.1). If the prediction is perfect ($\eta = 0$), the problem reduces to the offline incremental variant. If the prediction is arbitrarily erroneous ($\eta = m$), the algorithm effectively gets no information about the input and the problem reduces to the worst-case model. The main challenge is to design an algorithm that smoothly interpolates between the two regimes as a function of the prediction error η.

1.1 Our Contributions

One of our main contributions is designing the first learned data structure for the incremental SCC problem which is consistent, robust and smooth with respect to the prediction error.

[1] We assume $m \geq n$ to simplify the runtimes.

▶ **Theorem 1.** *We design a learned data structure that, given prediction $\hat{\sigma}$ with error η, maintains the strongly connected components of the graph in total time $O(m \log m \cdot \max\{\eta, 1\})$ for m inserts. Queries about the SCC of a given vertex take $O(1)$ time.*

To put this result in context, notice that if the edge arrivals are reasonably predictable (e.g. $\eta = O(\text{polylog}(m))$), our algorithm for the online incremental problem is nearly as fast as the offline variant.

The algorithmic idea is to use a divide-and-conquer approach on the predicted edge sequence $\hat{\sigma}$ to recursively precompute solutions to partial subproblems using Tarjan's algorithm. As edges arrive that are different from $\hat{\sigma}$, our algorithm updates the prediction and lazily recomputes the affected subproblems. At a high level, we show that we can bound the number of times each subproblem is recomputed by the prediction error η and thus lose an η factor over the offline incremental algorithm.

Robustness to Error. The running time of our algorithm grows with η – in fact, for sufficiently high η, our algorithm may be slower than known worst-case techniques. However, the algorithm can be made robust to arbitrarily erroneous predictions with respect to any worst-case algorithm by switching to the worst-case algorithm if the learned algorithm's runtime grows larger than the worst-case guarantee.

Similarly, our algorithm's running time is given in terms of the maximum error – as stated, even a single outlier can significantly impact performance guarantees. One potential strategy to handle a small number of outliers is to reset the algorithm (restarting from time zero with the outlier predicted correctly) each time an outlier arrives. Each reset is as expensive as an entire algorithm run, but between resets, performance is proportional to the maximum error among non-outliers. While this strategy is expensive on a per-outlier basis, it can be used to handle a small number of outliers without degrading the performance too much. In our experiments, our algorithm (without this intervention) seems to be somewhat resistant to outliers – its performance on our datasets is better than the maximum error would indicate.

Comparison with Worst-Case Approaches. Most prior algorithms for maintaining SCCs in incremental graphs do so by maintaining *topological ordering* of the *condensed graph* (a directed acyclic graph in which all vertices within an SCC are contracted into a new single node). A topological ordering is a labeling $L : V \to \mathbb{Z}$ of the vertices V such that $L(v) < L(u)$ if there is a directed path from v to u. A topological ordering exists if and only if the directed graph is acyclic. An algorithm that maintains the topological ordering can detect new cycles whenever a new edge points from a node with a higher label to a lower label. Such a cycle leads to the merging of previous (condensed) SCCs.

The best-known worst case total running times of algorithms using the topological-ordering-based combinatorial approach are $\tilde{O}(n^2)$ [3] for dense graphs and $\tilde{O}(m^{4/3})$ [5] for sparse graphs. Note that the $\tilde{O}$-notation hides logarithmic factors. Thus, even with reasonably large error as long as $\eta < \min\{n^2/m, m^{1/3}\}$, our algorithm improves upon these approaches.

In a recent theoretical advancement, Chen et al. [7] use the interior-point method to maintain SCCs in incremental graphs (without a topological ordering) in total time $m \cdot e^{O(\log^{167/168} m \log\log m)}$. While the improvement is theoretically appealing, the algorithm and techniques employed are not practical and the running time itself is galactic.

Faster algorithms are known for maintaining SCCs in the *decremental* setting (where all edges are available at the beginning and deleted one by one). This can be done in total expected time $O(m \log^4 m)$ [6]. Predictions of the edge sequence essentially help turn an incremental SCC problem (which seems to be harder) into a decremental one.

In contrast to all prior work, our algorithm maintains SCCs without maintaining a topological ordering or using complex techniques. Our algorithm exploits predictions to precompute partial solutions to recursive subproblems. Our recursive technique that uses predictions to lift the offline problem to an online one follows the approach used by McCauley et al. [21] for maintaining approximate shortest paths in incremental graphs using predictions.

One way our paper differs from the previous works on dynamic graph problems with predictions [32, 20, 21, 15] is that we co-designed the algorithm with its implementation. Concretely, our analysis explicitly tracks logarithmic factors in running time, and avoids constant-factor overheads when possible (e.g. rebuilding unnecessary subproblems as in [21]).

Experimental Results. We compare the runtime of our algorithm to the state-of-the-art heuristic IncSCC$^+$ [12] on real datasets[2]. IncSCC$^+$ maintains a topological ordering of the condensed graph and optimizes over running Tarjan's from scratch repeatedly. Despite the optimizations, their worst-case total cost is $\Theta(m^2 \log m)$ for m inserts.

Our experiments show that (1) our algorithm is scalable and can handle large datasets, (2) it outperforms the known recursive algorithm for the offline problem under perfect predictions, (3) it significantly outperforms IncSCC$^+$ on reasonably accurate predictions, and finally (4) its runtime degrades gradually with error. These results complement our theoretical analysis: they give evidence that the algorithm is reasonably simple, and its running time does not involve large constants.

Summary. In summary, our theoretical and experimental results demonstrate that predictions can help bridge the gap between theoretically-good algorithms and practical heuristics. Our results provide a proof-of-concept for designing algorithms with strong provable guarantees that can match or even outperform state-of-the-art heuristics on predictable inputs.

1.2 Additional Related Work

Algorithms with Predictions For Runtime. Kraska et al. [16] jump-started the area by *empirically* demonstrating how machine-learned predictions can speed up data structures. Dinitz et al. [11] present a *theoretical* framework for analyzing predictions for speeding up *offline* algorithms. This has since been used for problems such as maximum flow [8, 27], shortest path [17], minimum cut [26], stable matching [24], and convex optimization [30].

Predictions have been used to design provably faster data structures for problems such as online list labeling [22], incremental topological ordering [23], binary search [10], sorting [1], dictionaries [33, 19], and priority queries [4]. Similar to this paper, predictions have been used to "lift" offline solutions to dynamic graph problems to improve the runtime of incremental graph problems, such as incremental shortest paths [32, 15, 21] and divide-and-conquer algorithms [20]. We note in particular that our divide-and-conquer framework is similar to that of McCauley et al. [21], however, we keep only a single path rather than maintaining the entire tree. This improves runtime performance by several log factors and also makes the data structure more practical. Liu et al. [20] also use a divide-and-conquer framework, but their analysis requires balanced subproblem sizes. In contrast, we can handle unbalanced subproblems with the trade-off that our error bounds are on max error (rather than average). Overall, these results demonstrate significant potential of leveraging machine-learned predictions to speed up algorithms and data structures.

[2] We ended up optimizing IncSCC$^+$ beyond what was intended in [12]. We include runtime comparison against both variants.

2 Preliminaries

This section presents the formal setup and a warm-up algorithm for the offline incremental SCC problem.

2.1 Model and Notation

Let V be the set of vertices of the directed graph G. Let $\sigma = e_1, \ldots, e_m$ denote the sequence of online edge inserts. We assume that one edge arrives per time step, so we often say that e_t arrives at "time t." We use "graph at time t", denoted by G_t, to refer to the graph with vertex set V and the first t edges $e_1, \ldots, e_t$. We define G_0 as the graph on vertex set V with no edges. We assume that G_m has no isolated vertices; if it is not then all isolated vertices in G_m can be removed before running the algorithm, increasing the cost of the algorithm by $|V|$. Before any edges arrive, the learned algorithm receives a prediction $\hat{\sigma}$ of σ. We define $m = |\sigma| = |\hat{\sigma}|$. Let $i(e)$ be the index of e in σ and $\widehat{i}(e)$ be the index of e in $\hat{\sigma}$. Let $\eta_e = |i(e) - \widehat{i}(e)|$ for each edge e in σ. The error $\eta = \max_{e \in \sigma} \eta_e$ is the maximum error of any edge; for simplicity of exposition, we assume $\eta \geq 1$ (the analysis is asymptotically the same if $\eta = 0$ or $\eta = 1$). As we go through the edges, we update the predictions, and we denote the predicted arrival sequence at time t by $\hat{\sigma}_t$. This is the same prediction model as past work on incremental graph algorithms [15, 32, 21].

We assume that the set of edges in $\hat{\sigma}$ and σ are the same – that is, $\hat{\sigma}$ is a permutation of σ. It is possible to generalize this assumption, albeit with some extra cost. If an edge e arrives that is not in $\hat{\sigma}$, the algorithm can simply start over from the beginning, at a cost of $O(m \log m)$ – this does not increase the overall cost if the number of such edges is at most η. Edges in $\hat{\sigma}$ that never arrive are more expensive, since effectively $\eta_e = m$. One strategy to handle such edges is to use a large threshold τ: at each time step t, the algorithm can remove any edge that was predicted to arrive before $t - \tau$; if the edge later arrives it can be treated as an edge that is not in $\hat{\sigma}$.

Our goal is to maintain a data structure that, after edge e_t arrives, can answer queries to determine if u and v are in the same SCC of G_t, for any $u, v \in V$.

2.2 Warm Up: Offline Incremental SCC

Our algorithm is based closely on the offline recursive algorithm for the problem. For the offline incremental problem, the entire sequence of edge insertions σ is known ahead of time. The goal is to quickly process σ to allow us to answer SCC queries for G_t for any t. We describe a folklore recursive method to perform this preprocessing efficiently. This algorithm has been described in programming contest literature, see e.g., [28], and a similar structure has been used for designing a worst-case algorithm for decremental SCC [29].

The algorithm recursively divides the interval $[0, m]$ into halves. Each interval generated in this process corresponds to a **subproblem**. In particular, the resulting intervals form a hierarchical set of dyadic subintervals (for ease of exposition, we assume that m is a power of two, although this assumption is not necessary for the algorithm or its correctness). Each subproblem has two endpoints ℓ and r. We refer to a subproblem either by its endpoints, $[\ell, r]$, or by its midpoint, $x := (r + \ell)/2$. The subproblem $[\ell, r]$ has a **left child** $[\ell, x]$ and a **right child** $[x, r]$, and is referred to as their **parent** subproblem. See Figure 1 for a schematic view of the recursion tree.

For each interval $[\ell, r]$, the algorithm maintains a graph $\tilde{G}_x = (V_x, E_x)$. $\tilde{G}_x$ is defined recursively below, but to help motivate the definition let us briefly summarize its contents. Consider contracting all SCCs of G_ℓ into single vertices, retaining only edges between two

distinct SCCs. These vertices and edges are enough to determine what SCCs are created during $[\ell, r]$. Now, consider trimming the graph further: we remove all edges that do not contribute to forming a new SCC during $[\ell, r]$, and remove all isolated vertices from the graph; trimming the graph in this way still allows us to determine the correct SCCs during $[\ell, r]$. This is the idea behind $\tilde{G}_x$.

At the root of the recursion tree, $\ell = 0$, $r = m$ and $\tilde{G}_x = G_{m/2}$. The algorithm runs Tarjan's algorithm [31] on G_x to compute its SCCs. It uses the SCCs to compute the graphs for the left and right subproblems $\tilde{G}_x^{\text{left}}$ and $\tilde{G}_x^{\text{right}}$ respectively. We define $\tilde{G}_x^{\text{left}}$ using its vertex set V_x^{left} and edge set E_x^{left}; $\tilde{G}_x^{\text{right}}$ is defined likewise.

Right Subproblem. Consider a pair of vertices u and v in the same strongly connected component S of $\tilde{G}_x$. Since edges are only ever inserted, u and v will remain in the same SCC at all time steps after x. Thus, to construct the right subproblem, the algorithm contracts all nodes within the same SCC in $\tilde{G}_x$ into a single node and deletes all edges (u, v) such that u and v are within the same SCC. In particular, for the right subproblem corresponding to the interval $[x, r]$, it sets V_x^{right} to contain a contracted node for each strongly connected component in $\tilde{G}_x$. The edge set E_x^{right} includes all edges $(u, v) \in E_x$ such that (u, v) is an *inter SCC* edge with endpoints u and v in different SCCs in $\tilde{G}_x$.

Left Subproblem. Now consider an edge (u, v) in $\tilde{G}_x$ with end points in different SCCs of $\tilde{G}_x$. Then, the vertices u and v will be in different SCCs at all time steps before x. The algorithm can safely delete such edges when recursing on the left subproblems. That is, V_x^{left} contains all vertices in V_x with at least one incident edge, and E_x^{left} only contains the *intra-SCC* edges in E_x with both end points in the same SCC.

Recursion Tree. The algorithm recurses on the left and right subproblems until the interval $[\ell, r]$ has length two. This way, each time $t = 1, \ldots, m-1$ is the midpoint of exactly one subproblem in the recursion tree. Notice that the nodes of this recursion tree in an in-order traversal correspond to the sequence of graphs $G_0, \ldots, G_m$; see Figure 1.

Queries. The folklore solution [28] builds a graph to implicitly track SCC merges. Here we describe one simple way to handle the queries recursively instead.

To see if u and v are in the same SCC at time t, we begin at the root and find the descendant of the root containing t. If it is a right child, u and v may each be merged into a new vertex. If so, we replace u (resp. v) with the merged vertex, and recurse. When we reach a subproblem with midpoint t, we check if u and v are in the same SCC in $\tilde{G}_t$, and return the result. This takes $O(\log m)$ time.

In Section 3.2, we present a data structure for the online problem that explicitly maintains node labels over time and answers queries in $O(1)$ time.

Analysis. Each time the algorithm processes a subproblem, it recurses on two subproblems obtained by splitting the time interval into two halves, implying a recursion depth of $O(\log m)$. Running Tarjan's algorithm on a subproblem with s edges takes $O(s)$ time. At each recursive call, an edge is either an inter- or intra-SCC edge, and thus is only passed down to one of the left or right subproblems. Thus, each edge contributes to one subproblem per level and the total runtime is $O(m)$ per level of the recursion tree. The total insertion cost is thus $O(m \log m)$.

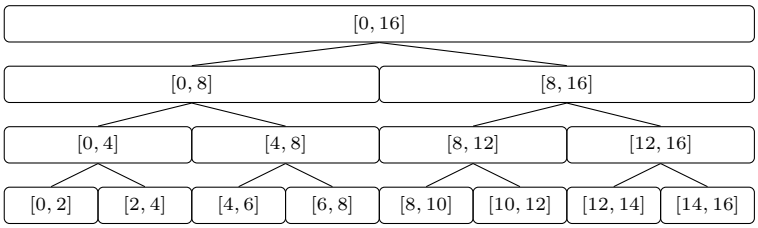

Figure 1 Recursive tree of subproblems for $m = 16$.

3 Incremental SCC with Predictions

In this section, we describe how we adapt the offline recursive algorithm for the incremental SCC problem with predictions. That is, given a prediction $\hat{\sigma}$ of σ at the beginning, we describe how to create a data structure that at each time step t, efficiently: (1) inserts e_t (the t-th edge in σ), and (2) allows answers to SCC queries on G_t.

First, notice that if the prediction sequence $\hat{\sigma}$ is *perfect* ($\hat{\sigma} = \sigma$), then the online problem reduces to the offline incremental problem. However, in general, the prediction $\hat{\sigma}$ could differ from σ significantly, and naively redoing the offline recursive approach each time is too expensive. Instead, our algorithm *only maintains a single path of the recursion tree* from Section 2.2; when each edge arrives, the algorithm updates the path, and updates the SCC of each vertex. In particular, initially, the algorithm only computes the root-to-left path from subproblem $[0, m]$ to subproblem $[0, 0]$ in the offline recursion tree built using the prediction $\hat{\sigma}$. As edges arrive, it updates the predicted sequence $\hat{\sigma}$ and recomputes the current root-to-leaf path for time step t. In our analysis, we show that we only incur recomputations that we can directly charge to the error in the prediction, that is, the algorithm is only $O(\eta)$-factor worse than the offline incremental algorithm.

3.1 Learned Incremental SCC Algorithm

First, we describe how our algorithm, **Learned IncSCC**, maintains and uses the prediction.

Our algorithm continuously updates $\hat{\sigma}$ as edges arrive. Let $\hat{\sigma}_t$ denote the updated prediction after t edges have arrived and $\hat{\sigma}_0 = \hat{\sigma}$. Thus $\hat{\sigma}_t$ agrees with σ on the first t edges. Let t and $\hat{t}$ denote the arrival time step of an edge e_t in σ and $\hat{\sigma}_{t-1}$ respectively. If $\hat{\sigma}_{t-1}$ was correct, then $\hat{t} = t$. Otherwise, we update $\hat{\sigma}_{t-1}$ to obtain $\hat{\sigma}_t$: we delete e from time $\hat{t}$, and insert it to time t. All edges in $\hat{\sigma}_{t-1}$ between t and $\hat{t}$ are shifted one time step later (note that $t' > t$); this gives $\hat{\sigma}_t$.

Now, we discuss how subproblems are maintained. At a high level, instead of building the entire recursive tree of the offline algorithm from Section 2.2 with respect to $\hat{\sigma}_0$, Learned IncSCC is more conservative. It lazily maintains the only thing needed to answer queries about the SCCs at the current timestep t: the path from the root to t in the recursive tree that corresponds to the updated prediction $\hat{\sigma}_t$.

Before any edges arrive, the algorithm uses the prediction $\hat{\sigma}_0$ and the offline recursive approach to **build** all subproblems along the path from the root $[0, m]$ to the leaf subproblem $[0, 0]$. We describe the implementation of this build in more detail momentarily.

Now consider step t when edge e_t is about to arrive. Let $\hat{t}$ be the time e_t was predicted to arrive in $\hat{\sigma}_{t-1}$. Before e_t arrives, the algorithm maintains the path from the root to time step $t-1$ of the recursion tree corresponding to $\hat{\sigma}_{t-1}$. After e_t arrives, the algorithm updates $\hat{\sigma}_t$ and the root-to-leaf path it maintains as follows. The algorithm begins with the last

subproblem in the current path, and walks backwards through the path to find a subproblem with interval $[\ell, r]$ such that $\hat{t} \in [\ell, r]$. Thus, $[\ell, r]$ is the lowest common ancestor (LCA) of the nodes $t-1$ and $\hat{t}$ in the recursion tree with respect to $\hat{\sigma}_{t-1}$. The algorithm calls the build procedure on this subproblem (and thus recursively on its children) to compute the path from the root to t with respect to $\hat{\sigma}_t$.

What We Store. Our algorithm maintains a path from the root to the current time t, implemented as an array of at most $\lceil \log_2 m \rceil$ pointers to subproblems.

Our algorithm maintains the following for each subproblem $[\ell, r]$ with midpoint x: (1) a graph $\tilde{G}_x = (V_x, E_x)$ whose nodes are contracted nodes of V, (2) the SCCs of $\tilde{G}_x$, (3) a mapping M_x from each vertex in $\tilde{G}_x$ to its **representative vertex** in V, and (4) a set of **left edges** and **right edges** that correspond to the edges of the left and right subproblems E_x^{left} and E_x^{right} as defined in Section 2.2. Each subproblem stores the edges of its children so that the edges do not need to be recomputed when the children are rebuilt.

Finally, our algorithm maintains the Node Label Data Structure which stores at all times t, a **label** $L(v)$ for each vertex $v \in V$, such that $L(u) = L(v)$ at time t if and only if u and v are in the same SCC in G_t. We use this data structure as a blackbox below and defer its implementation to Section 3.2.

Building a Subproblem. Below we describe how the algorithm performs a **build** operation at time t on an interval $[\ell, r]$ with midpoint x with respect to the predicted sequence $\hat{\sigma}_t$.

The algorithm begins by calculating the graph $\tilde{G}_x$ and the mapping M_x. We split into two cases depending on if $[\ell, r]$ is the root; in the case it is not, we split into further cases depending on if it is a left or right child.

If $[\ell, r]$ is the root subproblem, then $\tilde{G}_x = (V_x, E_x)$ where $V_x = V$ and E_x contains all edges in $\hat{\sigma}_t$. The mapping M_x has $M_x(v_i) = v_i$ for all vertices v_i.

If $[\ell, r]$ is not the root, let $[\ell^p, r^p]$ denote its current parent node. Let $\tilde{G}_x^p = (V_x^p, E_x^p)$ and M_x^p denote the graph and mapping stored by the parent subproblem. There are two cases depending on whether the interval $[\ell, r]$ is the left or right child of $[\ell^p, r^p]$.

If $[\ell, r]$ is the left child of $[\ell^p, r^p]$, then V_x is the set of vertices in V_x^p with at least one incident edge, i.e., incoming or outgoing, and the edge set E_x is the left edges stored in the parent subproblem $\tilde{G}_x^p$. The mapping is inherited from the parent, that is, $M_x = M_x^p$.

If $[\ell, r]$ is the right child of $[\ell^p, r^p]$, the algorithm contracts the vertices in the same SCC of $\tilde{G}_x^p$ to a single node, and these contracted nodes form V_x. The edge set E_x is the right edges stored in the parent subproblem $\tilde{G}_x^p$. For each node $c \in V_x$, which corresponds to an SCC in $\tilde{G}_x^p$, we compute $M_x(c)$ as follows. Let v be an arbitrary node in the SCC corresponding to c. We set $M_x(c) := M_x^p(v)$.

Now that $\tilde{G}_x$ and M_x are computed, the algorithm uses them to finish processing the subproblem. Similar to the offline case, the algorithm computes the strongly connected components of $\tilde{G}_x$ using Tarjan's algorithm, considering only the edges in E_x that appear at timesteps at most x according to $\hat{\sigma}_t$.

First, consider the case when the midpoint $x \neq t$. The algorithm computes the left edges of $\tilde{G}_x$ as those whose endpoints lie in the same SCC in $\tilde{G}_x$, and the right edges as those whose endpoints lie in different SCCs. These left and right edges are stored in the parent subproblem and inherited by child subproblems (rather than being recomputed when a child is rebuilt). After computing the left and right edges, the algorithm recursively builds the child containing t.

The base case occurs when the midpoint $x = t$. In the base case, the algorithm performs a final merge of the labels of nodes in the same SCC of $\tilde{G}_x$ in the Node Label Data Structure; we define this merge below.

3.2 Node Label Data Structure

The *Node Label Data Structure* stores, at all times t, the SCC of each vertex in G_t. It consists of: (1) an array L of size n, storing a *label* for each vertex such that $L[i] = L[j]$ if and only if vertices v_i and v_j are in the same SCC in G_t; and (2) for each label ℓ, a doubly linked list of all vertices with label ℓ (in other words, a list of all vertices v_i such that $L[i] = \ell$).

At initialization, $L[i] = i$ for all $i \in [n]$.

On a query to see if vertices v_i and v_j are in the same SCC, we simply check if $L[i] = L[j]$. This leads to $O(1)$ query time.

Given a collection of vertices V_c, the **merge** operation merges the SCCs of all vertices in V_c into a single SCC. Let V_c consist of k vertices $v_{c_1}, v_{c_2}, \ldots, v_{c_k}$. For $i = 1, \ldots, k-1$, we find labels $L[c_i]$ and $L[c_{i+1}]$, and determine which has a smaller list. Let ℓ_1 be the label of the smaller list (breaking ties arbitrarily) and ℓ_2 be the other. We iterate through the list of all vertices with label ℓ_1; for each vertex v_j, we set $L[j] = \ell_2$. Then, we add all vertices with label ℓ_1 to the list of vertices with label ℓ_2. Since the lists are implemented as linked lists, each merge operation takes time proportional to the number of vertices with label ℓ_1.

4 Running Time Analysis

In this section, we analyze the running time of our learned algorithm. We defer proofs related to correctness to the full version of the paper. To start, we bound what edges in the predicted edge sequence $\hat{\sigma}$ can cause a given subproblem to be rebuilt.

▶ **Lemma 2.** *If a subproblem $[\ell, r]$ with midpoint x using prediction $\hat{\sigma}$ is rebuilt at time t, then the edge e that arrives at time t was predicted to arrive in $\hat{\sigma}$ at a time either in $[x, x+\eta]$ or in $[r, r+\eta]$.*

Proof. The subproblem $[\ell, r]$ is rebuilt only when it lies on the path from $\mathrm{LCA}(t, \hat{t})$ to t for some edge. There are two ways this can happen:

1. $[\ell, r]$ is the $\mathrm{LCA}(t, \hat{t})$ for some pair $(t, \hat{t})$. This occurs only if $t \le x$ and $\hat{t} \ge x$. Since $|t - \hat{t}| \le \eta$, this requires that $\hat{t} \in [x, x+\eta]$.
2. $[\ell, r]$ lies strictly below the LCA. This only happens if t lies inside $[\ell, r]$, but $\hat{t}$ lies outside the subtree rooted at $[\ell, r]$; therefore, $\hat{t} \ge r$. Since $[\ell, r]$ is on the path to t, we must have $t \le r$; since $\hat{t} - t \le \eta$, we have that $\hat{t} \in [r, r+\eta]$. ◀

Before the next lemma, we motivate the structure of our analysis. Lemma 2 states that each subproblem is rebuilt only $O(\eta)$ times; this means that if we can bound the size of all subproblems (in other words, the total number of edges in all subproblems) we are done. Indeed, the classic analysis of the offline problem with $\eta = 0$ states that each edge is in at most one subproblem at each level. This would mean that the total size of all subproblems is $O(m \log m)$; combining with Lemma 2 would give the desired $O(\eta m \log m)$ bound.

Unfortunately, this analysis is incorrect. The reason is that as $\hat{\sigma}$ changes, what edge belongs to each subproblem changes as well. For example, according to $\hat{\sigma}_0$, the endpoints of an edge $e = (u, v)$ might not be in the same SCC in $G_{m/2}$, which puts e in the right child of the root. But at some later time $t < m/2$, an edge originally predicted to arrive at $t' > m/2$ might arrive that puts u and v in the same SCC in $G_{m/2}$. This would place e in the left child of the root. Lemmas 3 through 5 bound how often this can happen – i.e., an edge may be a part of $O(\eta \log m)$ builds in total.

For any edge (u, v) and prediction $\hat{\sigma}$, let the **combining time** $C_{u,v}(\hat{\sigma})$ be the first time in $\hat{\sigma}$ when u and v are in the same SCC. If u and v are never in the same SCC in $\hat{\sigma}$, then $C_{u,v}(\hat{\sigma})$ is undefined. Lemma 3 shows that at any time t, an edge (u, v) is only in a subproblem that contains its combining time. This lemma is useful both in analyzing the running time and proving correctness.

▶ **Lemma 3.** *Let $[\ell, r]$ be a subproblem in the offline recursive tree built on $\hat{\sigma}_t$ that contains edge $e = (u, v)$. If $C_{u,v}(\hat{\sigma}_t)$ is defined, then $C_{u,v}(\hat{\sigma}_t) \in [\ell, r]$; if $C_{u,v}(\hat{\sigma}_t)$ is undefined, then $r = m$.*

Proof. We argue by contradiction.

First, consider the case where $C_{u,v}(\hat{\sigma}_t)$ is undefined; assume by contradiction that e is in a subproblem $[\ell, r]$ with $r \neq m$. Let $[\ell, r]$ be the subproblem closest to the root containing e where $r \neq m$. Then $[\ell, r]$ was a left child – but by definition, this means that u and v were in the same component in the graph of the parent of $[\ell, r]$; a contradiction.

If $C_{u,v}(\hat{\sigma}_t)$ is defined, let $[\ell, r]$ be the subproblem in the current path closest to the root such that e is an edge in $[\ell, r]$, but $C_{u,v}(\hat{\sigma}_t) \notin [\ell, r]$.

If $C_{u,v}(\hat{\sigma}_t) < \ell$, let $[\ell_a, r_a]$ be the ancestor of $[\ell, r]$ with midpoint ℓ. $[\ell_a, r_a]$ must exist since if $C_{u,v}(\hat{\sigma}_t)$ exists, $\ell > 0$. Furthermore, $[\ell, r]$ must be a descendant of the right child of $[\ell_a, r_a]$. But since $C_{u,v}(\hat{\sigma}_t) < \ell$, u and v are in the same SCC in the midpoint of $[\ell_a, r_a]$, so e cannot be in the right edges of $[\ell_a, r_a]$, contradicting that it is an edge in $[\ell, r]$.

If $C_{u,v}(\hat{\sigma}_t) > r$, let $[\ell_a, r_a]$ be the ancestor of $[\ell, r]$ with midpoint r. $[\ell_a, r_a]$ must exist since if $C_{u,v}(\hat{\sigma}_t)$ exists, $r < m$. Furthermore, $[\ell, r]$ is a descendant of the left child of $[\ell_a, r_a]$. But since $C_{u,v}(\hat{\sigma}_t) > r$, u and v are in different SCCs in the midpoint of $[\ell_a, r_a]$, so e cannot be in the left edges of $[\ell_a, r_a]$, contradicting that it is an edge in $[\ell, r]$. ◀

Now, we show that the error bound η implies that the combining time of any pair of vertices cannot differ significantly as $\hat{\sigma}$ is updated. First, we need a lemma showing that the way we update predictions does not increase the error of any edge beyond η.

▶ **Lemma 4.** *For any edge e and any $\hat{\sigma}_t$, let $i_t(e)$ be the position of e in $\hat{\sigma}_t$, and $i(e)$ be the position of e in σ. Then if $\hat{\sigma}_0$ has maximum error η, we have $|i(e) - i_t(e)| \leq \eta$ – thus, $\hat{\sigma}_t$ also has maximum error η. This implies that for each t_1 and t_2 we have $|i_{t_1}(e) - i_{t_2}(e)| \leq 2\eta$.*

Proof. We prove this by induction on t; it is trivially true for $t = 0$.

Note that when we move from $\hat{\sigma}_{t-1}$ to $\hat{\sigma}_t$, we move an edge from $\hat{t} \geq t$ to its final location t; all edges in positions from t to $\hat{t} - 1$ move forward one, and all others remain unchanged. By definition, $\hat{t} - t \leq \eta$. The edge e being moved to t has $i_t(e) = i(e)$.

Any edge e being moved forward one (i.e. an edge between t and $\hat{t} - 1$ in $\hat{\sigma}_{t-1}$) that has $i(e) \geq \hat{t}$ is being moved closer to $i(e)$; by the inductive hypothesis the lemma holds. Now suppose an edge e is being moved forward one, and $i(e) \leq \hat{t}$. For all the edges that move forward we have $i_t(e) \leq \hat{t}$. Furthermore, since e has not arrived, $i(e) > t$. Combining with $\hat{t} - t \leq \eta$ implies $|i_t(e) - i(e)| \leq \eta$. ◀

Finally, Lemma 5 states that the combining time does not change significantly as we update the predictions.

▶ **Lemma 5.** *For any pair of vertices u, v, and any $\hat{\sigma}_t$, $C_{u,v}(\hat{\sigma}_t) \in [C_{u,v}(\hat{\sigma}_0) - 2\eta, C_{u,v}(\hat{\sigma}_0) + 2\eta]$.*

Proof. Consider the graph G' constructed using all vertices in V, and the first $C_{u,v}(\hat{\sigma}_0)$ edges in $\hat{\sigma}_0$; u and v are in the same strongly connected component in G'. Let G'' be the graph constructed using vertices in V, and the first $C_{u,v}(\hat{\sigma}_0) + 2\eta$ edges in $\hat{\sigma}_t$. By Lemma 4,

all edges in G' must have already arrived by time $C_{u,v}(\hat{\sigma}_0) + 2\eta$ in $\hat{\sigma}_t$, so the edges of G'' are a superset of the edges of G'; that is, u and v must be in the same SCC in G''. Therefore, $C_{u,v}(\hat{\sigma}_t) \leq C_{u,v}(\hat{\sigma}_0) + 2\eta$.

Let G' be the graph at time $C_{u,v}(\hat{\sigma}_0) - 1$ for $\hat{\sigma}_0$; u and v are not in the same strongly connected component in G'. Let G'' be the graph at time $C_{u,v}(\hat{\sigma}_0) - 2\eta - 1$ for $\hat{\sigma}_t$. By Lemma 4, all edges in G'' must have already arrived by $C_{u,v}(\hat{\sigma}_0) - 1$ in $\hat{\sigma}_0$, so G' has every edge G'' has. This means that u and v are not in the same strongly connected component in G'', which implies that $C_{u,v}(\hat{\sigma}_t) \geq C_{u,v}(\hat{\sigma}_0) - 2\eta$. ◀

We are ready to prove Theorem 1.

Proof of Theorem 1. Each time a node is relabeled in the Node Label Data Structure, the size of its SCC doubles; thus the total cost of all merges is $O(n \log n)$.

The cost to build a subproblem is linear in the number of vertices of $\tilde{G}_x$ and the number of edges $\tilde{E}_x$. Let $\mathcal{S}(t)$ be the set of subproblems built at time t. The total cost to build all subproblems is

$$\sum_t \sum_{t' \in \mathcal{S}(t)} (\#\text{ edges in } t') = \sum_{e \in \sigma} (\#\text{ builds of subproblems containing } e).$$

We bound this final sum. First, any $e = (u, v)$ where $C_{u,v}(\hat{\sigma}_t)$ is undefined can only be in subproblems with $r = m$ by Lemma 3; there are at most $\log m$ such subproblems, and each is rebuilt $O(\eta)$ times by Lemma 2, so the total cost of these edges is $O(m\eta \log m)$.

Now suppose $C_{u,v}(\hat{\sigma}_t)$ is defined and $e = (u, v)$ is in $[\ell, r]$ built at time t. We show that in each level, over time, there are $O(\eta)$ such subproblems. Summing over all $\log m$ levels gives a cost of $O(\eta m \log m)$.

If e is in $[\ell, r]$ at time t, by Lemma 3 we have $C_{u,v}(\hat{\sigma}_t) \in [\ell, r]$, and Lemma 5 implies that $C_{u,v}(\hat{\sigma}_0) \in [\ell - 2\eta, r + 2\eta]$.

Consider a depth d, such that $r - \ell \geq \eta$ for subproblems at depth d. There are at most five subproblems at depth d that have $C_{u,v}(\hat{\sigma}_0) \in [\ell - 2\eta, r + 2\eta]$. Each is rebuilt $O(\eta)$ times by Lemma 2; therefore, there are $O(\eta)$ subproblem builds at depth d that contain e.

Now we consider a depth d such that all subproblems have $r - \ell < \eta$. If a subproblem $[\ell, r]$ in this level contains e, then $C_{u,v}(\hat{\sigma}_0) \leq r + 2\eta < \ell + 3\eta$ and $C_{u,v}(\hat{\sigma}_0) \geq \ell - 2\eta > r - 3\eta$, which imply $\ell > C_{u,v}(\hat{\sigma}_0) - 3\eta$ and $r < C_{u,v}(\hat{\sigma}_0) + 3\eta$, respectively. Then if e' causes the subproblem $[\ell, r]$ to be rebuilt, by Lemma 2, e' was predicted to arrive at time steps $[\ell, r + \eta]$ in $\hat{\sigma}$. The length of this interval is $O(\eta)$, which means that e has been involved in $O(\eta)$ rebuilds at depth d over time. ◀

5 Experimental Evaluation

In this section, we describe our experimental results.[3]

Incremental SCC$^+$. We compare against the state-of-the-art heuristic for the problem: IncSCC$^+$ [12]. In IncSCC$^+$, the algorithm maintains a topological ordering of the nodes at all times, that is, each node v in the current (possibly contracted) graph is assigned a *topological rank* $r(v)$ such that for every arc (u, u') between distinct SCCs, we have $r(u) > r(u')$. When

[3] The Python implementations of our experiments are available at `https://github.com/helia-niaparast/incremental-strongly-connected-components-with-predictions`.

a new arc (v, w) is added in the "wrong" direction, i.e., $r(v) < r(w)$, the algorithm identifies potentially affected nodes by running forward and backward depth-first searches from w and v, respectively. It then applies Tarjan's algorithm on the affected nodes to determine whether a new cycle has been formed. We were not able to obtain the authors' original code, so we implemented our own version of their algorithm. In this implementation, when a cycle is formed, we merge the SCCs in the cycle into a new SCC, and run DFS on the whole graph to find the new topological ordering.

We also made an optimized version (IncSCC+ optimized) with two modifications. First, we skip the final run of Tarjan's algorithm, as cycle formation can be detected immediately from the affected nodes. Second, after a node is contracted, in order to find a new valid topological ordering, we only rearrange the affected nodes. These factors do not change the main ideas of the algorithm, but lead to a noticeable performance improvement in practice. In our tests, we compare to both IncSCC$^+$ implementations.

We also include the performance of the **offline incremental algorithm** from Section 2.2, which has access to the entire edge arrival sequence ahead of time.

Datasets. Our datasets are from SNAP[4] [18]. The `sx-superuser-c2a` and `sx-askubuntu` datasets are graphs based on two stackexchange forums; a link represents one user posting an answer or comment on another user's question or answer. These two are temporal, i.e., links are ordered based on the time the comments occurred. The other dataset used, `Slashdot0811`, is not temporal; edges arrive in random order. Each edge represents a user who tagged another as "friend" or "foe." The number of vertices and number of unique edges for each of these datasets is given in Table 1.

Experimental Setup. Our implementations are in Python. Our experiments were run on an Ubuntu Linux machine with a 2.8GHz i7-1165G7 Intel CPU with 32GB RAM.

Predictions. We parameterize how to generate predictions with a variable S, which is roughly the standard deviation of the error. For each value of S, we construct a perturbed edge sequence from the original ordering as follows. We traverse the edges in their original order, and for each edge, we sample an offset from a normal distribution with mean 0 and standard deviation S. The target index is set to the current index plus this offset. If both the current and target indices have not yet been modified, we swap the edges at these positions; otherwise, the edge remains fixed.

After processing all edges, we measure the prediction quality using two metrics relative to the original sequence: the average prediction error (η_{avg}) and the maximum prediction error ($\eta_{\max}$). For each $S \in \{0, 10, 20, \ldots\}$, we generate 10 independent perturbed sequences using the procedure described above. Each sequence yields its own average and maximum error values. We then run the algorithm on each sequence and report the average runtime and error across the 10 runs. In the plots, η_{avg} is the mean of the 10 average-error values and $\eta_{\max}$ is the mean of the 10 maximum-error values.

Discussion. We include the plot for `sx-askubuntu` in Figure 2, and include the results for the remaining datasets in the full version of the paper. With sufficiently accurate predictions, our algorithm gives significantly improved performance over both versions of IncSCC$^+$. The

[4] `https://snap.stanford.edu/data/`

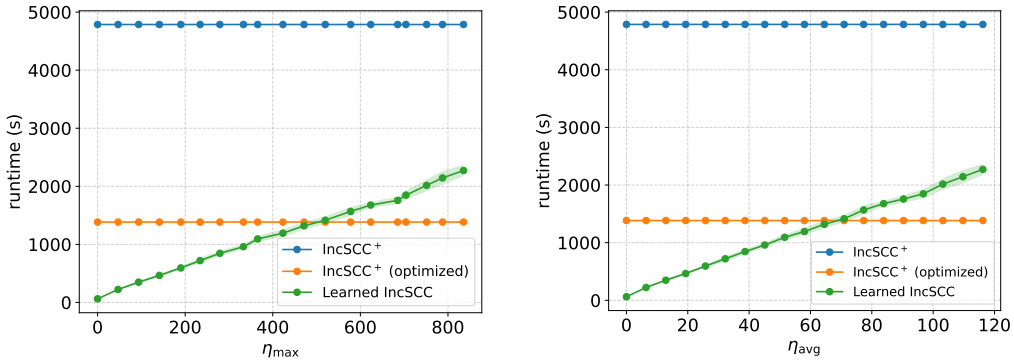

Figure 2 Runtime comparison on the `sx-askubuntu` dataset. In the left and right plots, the x-axis denotes the maximum and average prediction error with respect to the actual edge sequence, respectively. The green line shows the mean runtime, and the surrounding shaded region indicates the standard deviation.

Table 1 Comparison of the performance of Learned IncSCC against the baselines. The first two columns show the number of unique edges and nodes. The next four columns report the runtimes of the purely offline algorithm, Learned IncSCC with perfect predictions, and the two versions of IncSCC$^+$. Crossover η_{avg} and crossover $\eta_{\max}$ are the largest average and max errors for which Learned IncSCC outperforms IncSCC$^+$ (optimized).

Dataset	m	n	Offline	Learned IncSCC $\eta = 0$	IncSCC$^+$	IncSCC$^+$ (optimized)	Crossover η_{avg}	Crossover $\eta_{\max}$
`sx-superuser-c2a`	289487	101052	60.70	34.21	932.10	331.33	≈ 32	≈ 219
`sx-askubuntu`	596933	159316	85.37	62.24	4786.33	1384.01	≈ 70	≈ 517
`Slashdot0811`	905468	77360	71.33	47.92	2221.04	362.68	≈ 20	≈ 150

experiments show our performance grows roughly linearly with the error; in fact it grows linearly with both the η_{avg} and $\eta_{\max}$ (our theoretical analysis bounds performance only in terms of $\eta_{\max}$).

Table 1 summarizes further results, including the largest average and maximum η for which our algorithm improves on optimized IncSCC$^+$.

In Table 1, we also compare our algorithm with perfect predictions to the offline incremental algorithm from Section 2.2. While both algorithms are given the entire edge sequence, our algorithm can adapt to errors on-the-fly, while the offline algorithm is static. Despite this overhead, our running time is not only comparable but in fact slightly better than the offline algorithm. This demonstrates that our method is lightweight, while still allowing for potential error.

6 Conclusion

In this work, we design a new learned algorithm to maintain the strongly connected components in an incremental graph. Our theoretical bounds show that our algorithm is nearly optimal under reasonably good predictions and its performance degrades smoothly with respect to the prediction error.

We test our algorithm against the best heuristic on real-world datasets. Our experiments show that (1) theory is predictive of practice, and (2) it is possible to achieve strong provable guarantees against bad predictions, while also being able to match (and even beat) the practical performance of state-of-the-art heuristics when predictions are good.

References

1. Xingjian Bai and Christian Coester. Sorting with predictions. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL: https://openreview.net/forum?id=Qv7rWR9JWa.
2. Surender Baswana, Keerti Choudhary, and Liam Roditty. An efficient strongly connected components algorithm in the fault tolerant model. *Algorithmica*, 81(3):967–985, 2019. doi:10.1007/S00453-018-0452-3.
3. Michael A Bender, Jeremy T Fineman, Seth Gilbert, and Robert E Tarjan. A new approach to incremental cycle detection and related problems. *ACM Transactions on Algorithms (TALG)*, 12(2):1–22, 2015. doi:10.1145/2756553.
4. Ziyad Benomar and Christian Coester. Learning-augmented priority queues. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL: https://openreview.net/forum?id=1ATLLgvURu.
5. Aaron Bernstein, Aditi Dudeja, and Seth Pettie. Incremental scc maintenance in sparse graphs. In *29th Annual European Symposium on Algorithms (ESA 2021)*, 2021.
6. Aaron Bernstein, Maximilian Probst, and Christian Wulff-Nilsen. Decremental strongly-connected components and single-source reachability in near-linear time. In *Proceedings of the 51st Annual ACM SIGACT Symposium on theory of computing*, pages 365–376, 2019. doi:10.1145/3313276.3316335.
7. Li Chen, Rasmus Kyng, Yang P Liu, Simon Meierhans, and Maximilian Probst Gutenberg. Almost-linear time algorithms for incremental graphs: Cycle detection, sccs, st shortest path, and minimum-cost flow. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*, pages 1165–1173, 2024. doi:10.1145/3618260.3649745.
8. Sami Davies, Benjamin Moseley, Sergei Vassilvitskii, and Yuyan Wang. Predictive flows for faster ford-fulkerson. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proc. of the 40th International Conference on Machine Learning (ICML)*, volume 202 of *Proceedings of Machine Learning Research*, pages 7231–7248. PMLR, 23–29 July 2023. URL: https://proceedings.mlr.press/v202/davies23b.html.
9. Swati Dhingra, Poorvi S Dodwad, and Meghna Madan. Finding strongly connected components in a social network graph. *International Journal of Computer Applications*, 136(7):1–5, 2016.
10. Michael Dinitz, Sungjin Im, Thomas Lavastida, Ben Moseley, Aidin Niaparast, and Sergei Vassilvitskii. Binary search with distributional predictions. *Advances in Neural Information Processing Systems*, 37:90456–90472, 2024.
11. Michael Dinitz, Sungjin Im, Thomas Lavastida, Benjamin Moseley, and Sergei Vassilvitskii. Faster matchings via learned duals. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Proc. 34th Conference on Advances in Neural Information Processing Systems (NeurIPS)*, pages 10393–10406, 2021. URL: https://proceedings.neurips.cc/paper/2021/hash/5616060fb8ae85d93f334e7267307664-Abstract.html.
12. Wenfei Fan, Chunming Hu, and Chao Tian. Incremental graph computations: Doable and undoable. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 155–169, 2017. doi:10.1145/3035918.3035944.
13. Bernhard Haeupler, Telikepalli Kavitha, Rogers Mathew, Siddhartha Sen, and Robert E Tarjan. Incremental cycle detection, topological ordering, and strong component maintenance. *ACM Transactions on Algorithms (TALG)*, 8(1):1–33, 2012. doi:10.1145/2071379.2071382.
14. Benjamin Charles Hardekopf. *Pointer analysis: building a foundation for effective program analysis*. The University of Texas at Austin, 2009.
15. Monika Henzinger, Barna Saha, Martin P. Seybold, and Christopher Ye. On the complexity of algorithms with predictions for dynamic graph problems. In Venkatesan Guruswami, editor, *15th Innovations in Theoretical Computer Science Conference, ITCS 2024, January 30 to February 2, 2024, Berkeley, CA, USA*, volume 287 of *LIPIcs*, pages 62:1–62:25. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.ITCS.2024.62.

16 Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proc. 44th Annual International Conference on Management of Data, (SIGMOD)*, pages 489–504. ACM, 2018. doi:10.1145/3183713.3196909.

17 Silvio Lattanzi, Ola Svensson, and Sergei Vassilvitskii. Speeding up bellman ford via minimum violation permutations. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 18584–18598. PMLR, 2023. URL: https://proceedings.mlr.press/v202/lattanzi23a.html.

18 Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

19 Honghao Lin, Tian Luo, and David Woodruff. Learning augmented binary search trees. In *International Conference on Machine Learning*, pages 13431–13440. PMLR, 2022. URL: https://proceedings.mlr.press/v162/lin22f.html.

20 Quanquan C. Liu and Vaidehi Srinivas. The predicted-updates dynamic model: Offline, incremental, and decremental to fully dynamic transformations. In Shipra Agrawal and Aaron Roth, editors, *Proceedings of Thirty Seventh Conference on Learning Theory*, volume 247 of *Proceedings of Machine Learning Research*, pages 3582–3641. PMLR, 30 June–03 July 2024. URL: https://proceedings.mlr.press/v247/liu24c.html.

21 Samuel McCauley, Benjamin Moseley, Aidin Niaparast, Helia Niaparast, and Shikha Singh. Incremental approximate single-source shortest paths with predictions. In Keren Censor-Hillel, Fabrizio Grandoni, Joël Ouaknine, and Gabriele Puppis, editors, *52nd International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 334 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 117:1–117:20, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ICALP.2025.117.

22 Samuel McCauley, Benjamin Moseley, Aidin Niaparast, and Shikha Singh. Online list labeling with predictions. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Proc. 36th Conference on Neural Information Processing Systems (NeurIPS)*, volume 36, pages 60278–60290. Curran Associates, Inc., 2023.

23 Samuel McCauley, Benjamin Moseley, Aidin Niaparast, and Shikha Singh. Incremental topological ordering and cycle detection with predictions. In *Forty-first International Conference on Machine Learning*, 2024. URL: https://openreview.net/forum?id=wea7nsJdMc.

24 Samuel McCauley, Benjamin Moseley, Helia Niaparast, and Shikha Singh. Stable matching with predictions: Robustness and efficiency under pruned preferences. *arXiv preprint*, 2026. arXiv:2602.02254.

25 Michael Mitzenmacher and Sergei Vassilvitskii. Algorithms with predictions. *Communications of the ACM (CACM)*, 65(7):33–35, 2022. doi:10.1145/3528087.

26 Helia Niaparast, Benjamin Moseley, and Karan Singh. Faster global minimum cut with predictions. In *Forty-second International Conference on Machine Learning*, 2025. URL: https://openreview.net/forum?id=FeZoO7mfG7.

27 Adam Polak and Maksym Zub. Learning-augmented maximum flow. *Information Processing Letters*, 186:106487, 2024. doi:10.1016/j.ipl.2024.106487.

28 Mateusz Radecki. My own algorithm — offline incremental strongly connected components in o(m log(m)). https://codeforces.com/blog/entry/91608, 2021. Accessed: 2025-09-30.

29 Liam Roditty. Decremental maintenance of strongly connected components. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1143–1150. SIAM, 2013. doi:10.1137/1.9781611973105.82.

30 Shinsaku Sakaue and Taihei Oki. Discrete-convex-analysis-based framework for warm-starting algorithms with predictions. In *35th Conference on Neural Information Processing Systems (NeurIPS)*, 2022.

31 Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. doi:10.1137/0201010.

32 Jan van den Brand, Sebastian Forster, Yasamin Nazari, and Adam Polak. On dynamic graph algorithms with predictions. In David P. Woodruff, editor, *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024, Alexandria, VA, USA, January 7-10, 2024*, pages 3534–3557. SIAM, 2024. doi:10.1137/1.9781611977912.126.

33 Ali Zeynali, Shahin Kamali, and Mohammad Hajiesmaili. Robust learning-augmented dictionaries. In *Forty-first International Conference on Machine Learning*, 2024. URL: https://openreview.net/forum?id=XyhgssAo5b.

34 Yu Zhang, Xiaofei Liao, Xiang Shi, Hai Jin, and Bingsheng He. Efficient disk-based directed graph processing: A strongly connected component approach. *IEEE Transactions on Parallel and Distributed Systems*, 29(4):830–842, 2017. doi:10.1109/TPDS.2017.2776115.