TeGraph: A Novel General-Purpose Temporal Graph Computing Engine

9 authors, including:

Chengying Huan
Tsinghua University
11 PUBLICATIONS **77** CITATIONS

SEE PROFILE

Hang Liu
University of Massachusetts Lowell
56 PUBLICATIONS **988** CITATIONS

SEE PROFILE

Yongchao Liu
Ant Group
83 PUBLICATIONS **2,792** CITATIONS

SEE PROFILE

Jinlei Jiang
Tsinghua University
77 PUBLICATIONS **416** CITATIONS

SEE PROFILE

TeGraph: A Novel General-Purpose Temporal Graph Computing Engine

Chengying Huan[1], Hang Liu[2], Mengxing Liu[1], Yongchao Liu[3], Changhua He[3],
Kang Chen[1], Jinlei Jiang[1], Yongwei Wu[1], Shuaiwen Leon Song[4]

[1]*Tsinghua University*; [2]*Stevens Institute of Technology*; [3]*Ant Group*; [4]*University of Sydney*

Abstract—**Temporal graphs attach time information to edges and are commonly used for implementing time-critical applications that can not be effectively processed by traditional static and dynamic graph processing engines. State-of-the-art solutions that target temporal path problems remain ad-hoc and often suboptimal. A unified and high-performance solution that could efficiently process general temporal path problems via a universal optimization strategy and relieve practitioners from heavy optimization efforts is in urgent demand. In this paper, we make two key observations: (1) temporal path problems can be described as *topological-optimum* problems and solved by a universal single scan execution model; and (2) data redundancy commonly occurs in the native format of the transformed temporal graphs, which is unnecessary for information propagation and can be eliminated for better memory utilization and execution efficiency. Based on these core insights, we propose TEGRAPH, the first general-purpose temporal graph computing engine to provide a unified optimization strategy and execution model for general temporal path problems and their applications. TEGRAPH not only presents temporal information-aware graph representation that naturally fits temporal graphs but also offers general system-level supports such as out-of-core execution. Extensive evaluation reveals that TEGRAPH can achieve significant speedups over the state-of-the-art designs with up to *two orders of magnitude* ($241\times$) with the throughput of two hundred million edges per second.**

Index Terms—**Graph algorithm, temporal graphs.**

I. INTRODUCTION

Temporal graphs, which label the edges with time intervals, can provide additional capabilities to describe time-critical applications that can not be otherwise captured by traditional static graph computing engines [1]–[14]. In reality, many important applications are based on temporal graphs [15]–[31] such as aviation networks [32], e-commerce [33], and realtime epidemiology analysis (e.g., Influenza and COVID-19 outbreaks [34]). Social media graphs [35], [36] are also with a period of friending as the edge labels. Additionally, in the era of deep learning-based big data analytics, effectively extracting essential information from large and complex temporal graphs becomes increasingly critical for everyday life [33], [36]–[39]. Despite its importance, existing research has mainly focused

on non-temporal static graphs, while some have considered dynamic graphs as a sequence of updates to non-temporal graphs but without bounding time constraints [40]–[42]. Fig. 1(a) shows a temporal graph of an aviation network. Each edge is attached with a time interval. The time interval $(1, 2)$ attached to the edge from a to b indicates that there is a flight taking off from a at time 1 and arriving at b at time 2.

Most applications using or represented by temporal graph focus on solving the *general temporal path problems*. A *temporal path* is a legal path under time constraints. For example, to find a temporal path in an aviation network, the arrival time must be earlier than the departure time at each transit airport. There are several common temporal path problems such as *reachability*, e.g., whether there is a temporal path between two airports; *fastest path*, e.g., how to reach the destination as fast as possible; and *shortest path*, e.g., how to reach the destination with the lowest total cost. These path problems are the fundamental building blocks for many important applications.

Two main approaches have been proposed in the literature to address temporal path problems. The traditional method stores the temporal graph into an adjacency list format with each neighbor also including its temporal information. During processing, one can directly apply static graph execution algorithms, e.g., Dijkstra's algorithm [43] or Bellman-Ford [44], but with extra consideration of time constraints [45]. This is referred as *static execution* [32], [35], [46]–[51], which would experience redundant data access and computations (Sec. II-C). A more promising approach is to transform the original temporal graph to an equivalent but larger static graph by expanding each vertex to multiple ones according to the timing information. The topological structure of the transformed graph contains all the necessary timing constraints. The transformed graph is then processed using state-of-the-art static graph processing models. This method is referred as *transformation-based execution*. Fig. 1(c) illustrates a transformation example.

State-of-the-art temporal graph processing techniques face three fundamental challenges, namely *computation, space, and bandwidth*. Among them, the computation challenge is the primary obstacle to achieving the full potential of efficient processing. Static execution incurs both high computation complexity and bandwidth overhead in order to handle the additional time information. Transformation-based execution, on the other hand, drastically simplifies computation via process-
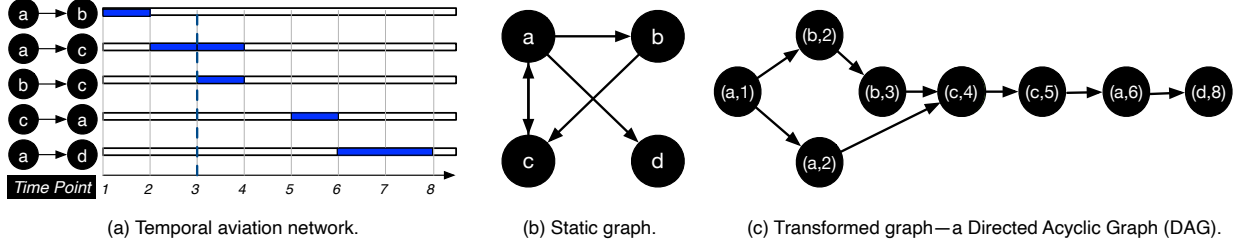
Fig. 1: An aviation map example.

ing transformed graphs (DAGs) but suffers from high latency for relying on the traditional iterative processing models. We have demonstrated that there lacks *a universal optimization strategy* for achieving the optimal performance across different path problems (Sec. II-D). For space, the graph transformation in transformation-based execution, although convenient, does suffer from the graph amplification problem which incurs large memory and disk I/O overhead. Finally, for bandwidth, state-of-the-art designs resort to temporal information associated with static graph representations, such as edge list and adjacency list, to represent temporal graphs. This results in enormous bandwidth waste for temporal graph analytics which is I/O intensive.

This paper attempts to provide a *fast, space and bandwidth efficient* solution for processing general temporal path problems and their applications. By treating this problem as a complete end-to-end optimization, we identify **two key insights** to support our overall solution. First, we find that temporal path problems essentially aim to solve so-called *topological-optimum* problems, in which each prefix (or a *subpath*) of the final result path is also the solution for the sub-problem. Therefore, we prove that once a temporal graph is transformed into a static DAG, temporal path problems become topological-optimum. By leveraging this unique feature, we propose a *universal single scan execution* (Sec. III-B) for fast processing the general path problems on temporal graphs by treating them as *topological-optimum* problems on the transformed DAGs. It replaces the generic iterative-based graph execution models which require continuously read edges and computing until convergence. Second, we investigate the graph transformation phase prior to the execution and observe that, due to the inherent nature of temporal graphs, their transformed DAGs may contain a large volume of redundancy that is unnecessary for information propagation.

Based on this insight, we propose a novel transformation model named *hyper-method* (Sec. III-D) to address the graph amplification challenge and further accelerate the execution phase via effectively condensing the transformed graph without any information loss. It dramatically reduces the execution workload and memory footprint to encourage in-memory processing with reduced disk I/O. Finally, we represent the temporal graph with a novel time-aware graph format (Sec. IV) which splits the same vertex that appears at dissimilar time stamps as different vertices to further avoid unnecessary data loads during processing.

With these three novel designs as the core, we propose TE-GRAPH, the first general-purpose temporal graph computing engine to provide a unified execution model for efficiently processing general temporal path problems and their applications. In addition to the core optimizations, TEGRAPH provides a quantitative analysis to understand the benefits of TEGRAPH. Extensive evaluation on real-world graphs has shown that TEGRAPH can achieve significant speedups over the state-of-the-art systems, up to two orders of magnitude $(241\times)$ with the throughput of two hundred million edges per second. It also performs better at larger datasets. In addition, TEGRAPH also saves up to 61% disk space and achieves up to $17\times$ graph transformation speedup. Finally, we provide the piecewise breakdown of performance speedups from each proposed technique, confirming that the overall achieved acceleration is from a combination of different optimizations.

II. BACKGROUND AND MOTIVATION

A. Notations for Temporal Graph

Different from static graphs, edges in temporal graphs have time intervals. Let $\mathbb{G} = (V, E)$ be a temporal graph, where V is the vertex set and E is the set of edges. Each edge $e \in E$ is defined as (u, v, s, t), where $u, v \in V$ and there exists an edge from u to v starting at time s and ending at time t with $s, t \in \mathbb{R}$. For a graph with costs, we attach a weight w to each edge as (u, v, s, t, w). For simplicity, we assume all the elements are meaningful if $s < t$. There may be multiple edges between the same pair of vertices. We use $d_{in}[v]$ and $d_{out}[v]$ to denote the in-degree and out-degree of v respectively. We use D to denote the maximum in-degree or out-degree in the graph. Furthermore, a path $P = \{v_1, v_2, .., v_{n+1}\}$ in a temporal graph can also be denoted as $P = e_1 \cdot e_2 \cdot ...e_n$, where $e_i = (v_i, v_{i+1}, s_i, t_i)$ and $end(e_i) \leq start(e_{i+1})$, for $1 \leq i \leq n$. For example, in Fig. 1(a), one can move from the edge $(a, b, 1, 2)$ to $(b, c, 3, 4)$, but cannot move from $(c, a, 5, 6)$ to $(a, b, 1, 2)$. A prefix of a path is called a subpath. For example, $P' = e_1 \cdot e_2 \cdot ...e_k$ is a prefix of P if $P = P' \cdot e_{k+1} \cdot ... \cdot e_n$, which can also be denoted as $P = P' \cup \{v_{k+2}, v_{k+3}, ..., v_{n+1}\}$.

In real-world functions, a temporal graph is represented as an *edge stream* [32], [33], [52]: it is simply a sequence of all the edges coming in the order of time that each edge is created or collected. Edges in temporal graphs are normally ordered based on their starting time. The edge stream data representation is a common format for temporal graphs.
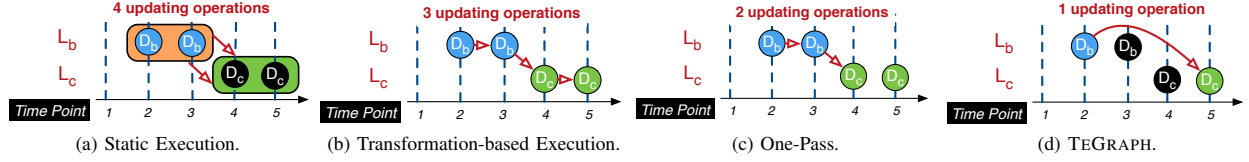
Fig. 2: Updating edge $(b, c, 3, 4)$ on Fig. 1(a) in static execution, transformation-based execution, One-Pass, and TEGRAPH.

B. General Temporal Path Problems

Most real-world applications using or represented by temporal graphs focus on solving the *temporal path* problems, which are the essential building blocks for many advanced graph analytics [25], [32], [36], [52], [53]. We list several most representative temporal path problems as follows [32], [54], where each problem makes a single-source query. Given a vertex $u \in V$ and a time interval $[start, end]$, the temporal path problem is to find the temporal path P between the time interval (i.e., $start \le start(P)$ and $end(P) \le end$) for each vertex $v \in V$, where P meets the following conditions.

- **Reachability:** Reachability from u to v means that $\mathbf{P}(u,v)$ is not empty, where $\mathbf{P}(u,v) = \{P : P \text{ is a temporal path from } u \text{ to } v\}$.
- **Fastest Path:** $P \in \mathbf{P}(u,v)$ is the fastest path between u and v if $duration(P) = min\{duration(P') : P' \in \mathbf{P}(u,v)\}$.
- **Shortest Path:** $P \in \mathbf{P}(u,v)$ is the shortest path between u and v if $dist(P) = min\{dist(P') : P' \in \mathbf{P}(u,v)\}$.
- **Top K Nearest Neighbors (Top KNN):** $\forall u \in K$ and $\forall v \in V/K$, $score(u) \le score(v)$, then K is the set of k-nearest neighbors of the vertex x. $score(u)$ is a self-defined measure function, e.g., the shortest path from x to u in this paper.

Based on these general temporal path problems, a wide range of applications can be defined, e.g., betweenness centrality [55], closeness centrality [56], [57], etc.

C. State-of-the-art Solutions

Static Execution: Typically, techniques based on static execution directly apply traditional static graph execution models such as Dijkstra's [43] or Bellman-Ford [44] based methods to process temporal graphs [32], [35], [47]–[51]. This approach incurs both high computation complexity and memory access overhead since an additional time dimension is introduced and extra procedures are needed for guaranteeing time constraints. Using shortest path processing as an example, it extends the shortest path $d[u]$ to a series of $d[u][t]$ to capture the optimal path to the vertex at time t while applying greedy algorithms. During each iteration, it needs to compare $d[u][t]$ with the other time instances of u. Fig. 2(a) demonstrates the updating function for the edge $(b, c, 3, 4)$ in Fig. 1(a) under static execution, $(b, 2)$ and $(b, 3)$ will have to individually update vertices $(c, 4)$ and $(c, 5)$ with total four update operations. This leads to redundant computations. Further, since static execution stores neighbors of different temporal information together, expired neighbors will be loaded, leading to redundant data access.

A recent proposal named **One-Pass** [32] directly modifies traditional static graph algorithms to better process temporal graphs in a single iteration. For each vertex v, One-Pass maintains a list, i.e., L_v. Each entry of L_v is a (d, t) tuple, where d is the distance of vertex v from the source vertex at time t. Here, it requires all the tuples to be sorted by t. During computation, for each edge (u, v, s, t, w), One-Pass updates the corresponding tuples in the destination endpoint v of this edge, i.e., L_v. Particularly, One-Pass needs to find all the (d, s_i) tuples in the list L_u, such that $s_i \le s$, where u is the source endpoint of this edge. Subsequently, One-Pass uses these (d, s_i) tuples to update the (d, t) tuple in L_v. If we run One-Pass on Fig. 1(a), we arrive at Fig. 2c. It sequentially scans all edges by the edge stream order. Using edge (b, c, 3, 4) as an example, when updating this edge, it first searches the list L_b whose time instance is ≤ 3, then updates the value of the destination vertex, i.e., (c, 4), in L_c to the resultant distance. Updating other edges will be the same. Since finding the tuples in L_u and L_v takes $log(D)$, the time complexity for **One-Pass** is $O(|E|log(D))$. As shown in Table I, the space complexity of One-Pass on the reachability is $O(|E|+|V|)$ which stores the list L of each vertex and graph dataset. For other applications, the space complexity is $O(|E|+2|V|)$ which needs more space for storing the range query data structure. As we will discuss shortly, although One-Pass is faster than the traditional static execution techniques, it is (i) slower than our **single scan** approach which only needs $O(|E|)$, and (ii) requires ad-hoc optimizations for different temporal path applications.

Transformation-Based Execution: The transformation-based execution [32], [52], [53], [58], [59] techniques first transform the original temporal graph to an equivalent but larger DAG (Directed Acyclic Graph) with timing information embedded (Fig. 1(c)), and then apply static graph models to the transformed DAG. Compared to the static execution, although introducing an additional transformation phase, the core graph execution is more straightforward without the need to deal with the additional time dimension. For the actual graph transformation, each edge will generate two vertices. For example, edge $u \rightarrow v : (u, v, s, t)$ will generate two vertices: $(u, s)_{out}$ (called an *out-vertex*) and $(v, t)_{in}$ (called an *in-vertex*). Each vertex in the transformed DAG has a label (u, t) where u is the *vertex instance* and t is the *time instance* (Fig. 1(c)). Formally, we define the graph transformation in the state-of-the-art transformation-based execution work **Trans** [52], [53]:

STEP 1: VERTEX TRANSFORMATION. Suppose that the original graph G is transformed to a new graph G'. Let $T_{in}[v]$

TABLE I: Time and space complexity comparison.

Algo.	One-Pass		Trans		A*		TeGraph	
	Time	Space	Time	Space	Time	Space	Time	Space
Reachability	$O(|E|)$	$O(|E|+|V|)$	$O(|E|)$	$O(|E|+|V|)$	$O(b^d)$	$O(|E|+|V|)$	$O(|E|)$	$O(|E|+|V|)$
Fastest Path	$O(|E|log(D))$	$O(|E|+2|V|)$	$O(|E|)$	$O(|E|+|V|)$	$O(b^d)$	$O(|E|+|V|)$	$O(|E|)$	$O(|E|+|V|)$
Shortest Path	$O(|E|log(D))$	$O(|E|+2|V|)$	$O(|E|log(|E|))$	$O(|E|+2|V|)$	$O(b^d)$	$O(|E|+|V|)$	$O(|E|)$	$O(|E|+|V|)$
Top KNN	$O(|E|log(D))$	$O(|E|+2|V|)$	$O(|E|log(|E|))$	$O(|E|+2|V|)$	$O(b^d)$	$O(|E|+|V|)$	$O(|E|)$	$O(|E|+|V|)$

and $T_{out}[v]$ be the set of in-vertices and out-vertices in G' with the same vertex instance v. Both sets are sorted by their time instances. All the elements in $T_{in}[v]$ are unique, and also is for $T_{out}[v]$. The vertices set in G' is $V' = \bigcup_{v \in V}\{T_{in}[v] \cup T_{out}[v]\}$.

STEP 2: EDGE TRANSFORMATION. First, all edges in the original temporal graph G are directly included in G' by connecting the corresponding transformed vertices. Specifically, edges are created between the in-vertices set $T_{in}[v]$ of each vertex v; and the same for the out-vertices set $T_{out}[v]$. Finally, edges are created from $T_{in}[v]$ to $T_{out}[v]$ for each vertex v by the increment of time. Specifically, for each $(v, t) \in T_{in}[v]$, if it can find a tuple $(v, t') \in T_{out}[v]$ satisfying that $t' = min\{t'' \mid (v, t'') \in T_{out}[v] \ , \ t < t''\}$ and there does not exist any in-vertex $(v, t'') \in T_{in}[v]$ satisfying that $t < t'' \le t'$, it will create a direct edge from (v, t) to (v, t').

With the graph transformed as G', traditional static graph execution models such as the priority-queue-based Dijkstra's algorithm [60], which uses a priority queue for holding the distances to the vertices, can be directly applied to process G'. Fig. 2(b) shows the updating process of edge $(b, c, 3, 4)$ in **Trans** [52], [53]. Unlike the static execution that requires both (b,2) and (b,3) to individually update (c,4) and (c,5), the update function here follows the path: from (b,2)->(b,3), (b,3)->(c,4), and then (c,4)->(c,5). Since the transformation is about sorting the edges by the temporal information, high-dimensional temporal information will slightly disturb the transformation. That is, we will sort the edges by the first dimension of the temporal vector. If the first dimension information is the same, we will move on to the second dimension and thereafter. As shown in Table I, for the reachability and fastest path, Trans uses Breadth-First Search (BFS) which only needs $O(|E|)$ time and $O(|E|+|V|)$ space. For the shortest path and Top KNN, the time complexity is $O(|E|log(|E|))$ and the space complexity is $O(|E|+2|V|)$. Note, shortest path and Top KNN need more space (e.g., priority queue) for the query.

A*-Based Execution: Under the transformed execution, some static graph searching algorithms such as A* [61] and A*-like algorithms [62]–[64] can be directly applied to the transformed graph for computations. A* uses an additive evaluation function $f(u) = g(u) + h(u)$ to search the transformed graph. $f(u)$ indicates the priority of each vertex u, $g(u)$ records the value of u starting from the source, and $h(u)$ represents the estimated value from u to the destination. During searching, it finds the vertex, which is yet searched and presents the highest priority, in the OPEN list, updates the functions (f, g, and h) of the vertex's neighbors, then moves the vertex into the CLOSED list. The time complexity of A*
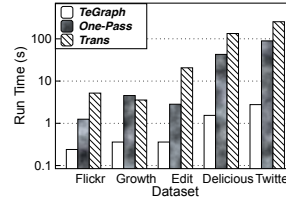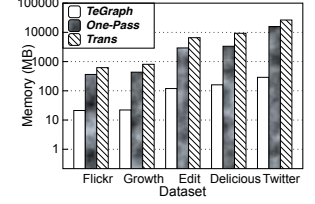

Fig. 3: Performance.   Fig. 4: Memory overhead.

is $O(b^d)$ as shown in Table I, where b denotes the branching factor and d denotes the depth of the solution. While this time complexity is close to $O(|E|)$, it contains higher constants for involving more updates per computation, i.e., updating $f(u)$, $g(u)$ and $h(u)$, when compared to our design which only needs one update. The space complexity of A* is $O(|E|+|V|)$ for storing the graph and the vertex states [65].

D. Open Problems and Challenges

Although the state-of-the-art techniques have shown some promises for providing ad-hoc solutions to certain temporal path problems, a *fast, space and bandwidth efficient* solution still does not exist. We summarize the following three major challenges facing the current approaches through both empirical results and theoretical analysis.

Computation Challenge: As discussed previously, static execution approaches (e.g., **One-Pass**) incur both computation challenge and memory access overhead for handling the additional time dimension on execution. Transformation-based execution, on the other hand, significantly reduces the computation complexity via processing transformed static graphs (DAGs) but suffers from high latency of using traditional iterative static graph execution models (Dijkstra's or Bellman-Ford) and additional graph transformation overhead. Fig. 3 shows that both **One-Pass** and **Trans** experience high latency when processing the shortest path problem on different temporal graph datasets. More importantly, these previous approaches only provide individual optimizations towards specific path problems without a generic graph format for achieving optimal performance across different general path problems. Table I demonstrates the theoretical time complexity across different designs on several path problems.

Space Challenge: Transforming temporal graphs into DAGs for processing is much more convenient without introducing time dimension constraints on execution. However, it suffers from the graph amplification challenge and additional overhead for the transformation process itself. For instance, under the traditional graph transformation strategy (e.g., **Trans**), the size of the vertices set increases from $|V|$

to $|E|$ and the edges set is doubled. Therefore, the space complexity of **Trans** in Table I is up to $O(4|E|)$ while TEGRAPH always enjoys $O(|E|+|V|)$ space complexity. The graph amplification poses significant performance and memory overhead for the execution phase afterward. First, it increases the base workload for graph processing. Second, it incurs large memory overhead which prevents large-scale temporal graphs from being efficiently executed on a single machine with out-of-core execution support. Fig. 4 demonstrates this inefficiency. In our experiments, we observe an average of $2.5\times$ and $20\times$ enlargement for edges and vertices under **Trans**, respectively.

Bandwidth Challenge: Recent efforts that simply associate the temporal information to either traditional edge list or adjacency list formats would result in redundant data access. Using time constraint associated adjacency list as an example, since we put the neighbors of various time constraints together, at a certain time step, although some neighbors are already expired, this adjacency list format will still load the entire adjacency list of an active vertex into the cores for filtering and further processing. This will exacerbate the I/O intensity of the temporal graph analytics.

Our Objective: To address these challenges, we propose TEGRAPH, the first general-purpose temporal graph computing framework for efficiently processing temporal path problems and their applications (i.e., requires a single update operation shown in Fig. 2(c)). The core of TEGRAPH is its novel execution (Sec. III-B, Sec. III-C), transformation (Sec. III-D) models, and I/O friendly graph representation (Sec. IV) which effectively address both the performance (e.g., Fig. 3) and graph amplification challenges (e.g., Fig.4).

III. TEGRAPH'S EXECUTION AND TRANSFORMATION

A. System Overview

Fig. 5 shows the overall architecture of TEGRAPH, which consists of three major components: the *transformation phase* based on *hyper-method* (Sec. III-D), the *execution phase* supported by topological single scan (Sec. III-B), and a *time-aware* graph representation (Sec. IV). Inside TEGRAPH, a temporal graph is first transformed to a static DAG via our *hyper-method*, with new identifiers assigned to vertices and edges. After the transformation, a novel time-aware graph format is proposed to efficiently store the graph. Finally, TEGRAPH's execution engine applies the proposed *topological single scan* to fast process temporal graph applications: it scans all edges only once according to the order of their starting time and updates properties of their destination vertices.

B. Execution Model

Sec. II-D has demonstrated that traditional execution models are limited to only provide ad-hoc or application-specific optimizations without providing a general and unified optimization strategy for achieving optimal performance across a variety of temporal path problems and their applications. This is because they do not leverage the inherently unique features presented
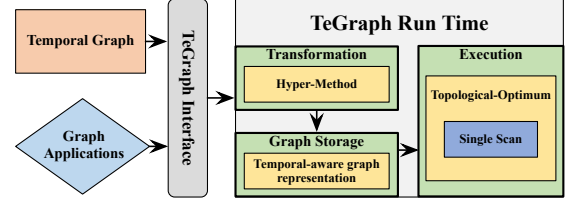

Fig. 5: TEGRAPH execution flow diagram.

in temporal graphs and their transformation, which drives the design of TEGRAPH's *single scan execution model*.

First, we define the concept of *topological-optimum*: if a path in a graph is an optimal path and each of its subpaths is also an optimal path, the graph is *topological-optimum* and has a *greedy* feature. Formally, given two vertices u, v, the path $P = \{u, v_1, v_2, ..., v_n, v\}$ is an optimal path between u, v. Under the topological-optimum rule, for any internal vertex $v_k, k \leq n$, the path $P_k = \{u, v_1, v_2, ..., v_k\}$ is also an optimal path between u and v_k. In other words, a subpath of the final result path is also the solution for the sub-problem. However, because of the time constraints in temporal graphs, they are inherently not topological-optimum. That is why the state-of-the-art strategies tend to apply ad-hoc solutions to find application-specific optimizations.

Next, we make the following **key observation**: *When a temporal graph is transformed into a static DAG, it becomes topological-optimum. More importantly, the processing of its path problems also follows this feature.*

To further prove this observation, we then define the *target function* of a temporal path problem: all the temporal path problems can be described using a target function $target(P)$ on path P. Note that according to this target function, path problems can be divided into minimum path problems and maximum path problems. Because a maximum path problem can be transformed to a minimum path problem, and vice versa, we use minimum path to simplify the discussion. Additionally, if there are more than one legal paths between two vertices in a graph, the problem is to find the path P with the minimum $target(P)$. We define $Target_{min}(\mathbf{P}(u, v)) = min\{target(P) : P \in \mathbf{P}(u, v)\}$. A path P is the *minimum path* between u, v if $target(P) = Target_{min}(\mathbf{P}(u, v))$. Thus, all the legal paths in a DAG G' after the transformation from the original temporal graph G can be described as $P = \{(u, t_1), ..., (v, t_k)\}$. We list the target functions of common temporal path problems as follows:

1) **Reachability:** $target(P) = 0$ for all legal paths.
2) **Fastest path:** $target(P) = t_k - t_1$, where t_k is the time instance of the last vertex and t_1 is the time instance of the source vertex.
3) **Shortest path:** $target(P) = \sum w_i$, where w_i is the edge weight on the path.
4) **Top KNN:** $target(P) = \sum w_i$, where w_i is the edge weight on the path.

It is straightforward to derive these temporal path problems with the given target functions are topological-optimum.
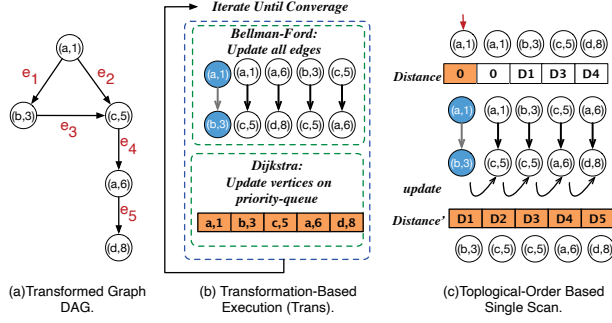
Fig. 6: Workflows of the single scan execution.

Algorithm 1 Single scan algorithm.

> **for all** $v \in V$ **do**
> $d[v] = +\infty$ except $v = u.\ d[u] = 0$.
> **for all** $e = (u', v) \in E$ in the topological order **do**
>   **if** $d[v] > update(d[u'], e)$ **then**
> $d[v] = update(d[u'], e)$
> **return** $d[v]$ for all $v \in V$

Single Scan Execution: By leveraging this unique feature, the core execution model of TEGRAPH is a *universal single scan execution* for fast processing the general path problems on temporal graphs by treating them as *topological-optimum* problems on the transformed graphs (DAGs). Because the topological order of the DAG is already contained by following the time instance order from the original temporal graph, it requires no additional cost for reordering DAG after transformation. Thus, a temporal path problem can be solved by *only a single round of scan over all the edges following the topological order of the transformed DAG* thanks to the time order of temporal graph. Fig. 6 and Algorithm 1 demonstrates how our single scan accesses the transformed graph in Fig. 6(a) merely once and finishes the entire computation. It first initializes all $d[v] = +\infty$ and initializes the source vertex to zero ($d[(a, 1)] = 0$), then scans all the edges according to their topological order in Fig. 6(a) and updates the distance value of the destination vertex of each edge. After this single scan, the distance values for all the vertices are the final results. As seen in Fig 6(c), we work on edge e_1 and update the distance of $(b, 3)$ to D_1, similarly for the rest of the edges. The correctness of the final results is expressed as Theorem 1 which can be derived by greedy choice.

Theorem 1. *Single scan algorithm can derive the target value for each vertex.*

Fig. 6(b) illustrates how to apply state-of-the-art designs, e.g., Bellman-Ford [44] or priority-queue based Dijkstra's algorithms [60] to the transformed graph. Basically, in each iteration, it updates edges until the algorithm is converged. For Bellman-Ford based execution, it introduces large computational redundancy because of slow convergence speed.

We also notice that our design is closely related to the existing static execution model, i.e., One-Pass [32]. However, for updating each edge, we only need $O(1)$ as opposed to $O(log(D))$ time complexity required by One-Pass as discussed
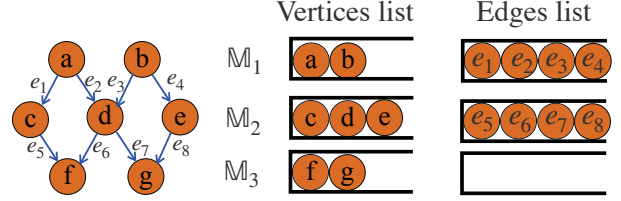
in Sec. II-C. The root cause is that One-Pass uses a list to maintain all the temporal instances for each vertex. Therefore, although One-Pass still needs a single round scan as TEGRAPH, it has larger processing overhead than TEGRAPH for ($O(|E|log(D))$ vs $O(|E|)$).

Fig. 7: A parallel single scan example.

Algorithm 2 Parallel single scan.

> **for all** $v \in V$ **do**
> $d[v] = +\infty$ except $v = u.\ d[u] = 0$.
> P= Find edges whose source vertices do not have in-edges in E
> **for all** $e = (u', v) \in P\ in\ parallel$ **do**
>   **if** $d[v] > update(d[u'], e)$ **then**
> $d[v] = update(d[u'], e)$
>   Remove e in E
>   **if** v does not have in-edges in E **then**
>   Add out-edges of v to P

C. Parallel Single Scan

Despite that our single scan sequentially scans the graph and updates each edge following the order of the transformed graph (sorted by starting time of edges), it can support multithreads. Particularly, the parallelism opportunity surfaces when the edges can be updated independently. Formally, we define an order $\prec$ between two edges e and e' as Equation 1 to show that the update of edge e will affect that of edge e'. If $e \prec e'$, e must be updated before e'.

$$e \prec e' \longrightarrow \exists\ P \mid P = e \cdot e_1 \cdot e_2 \cdot ... \cdot e'. \tag{1}$$

Therefore, we can get an edges set $\mathbb{M}$ where all the edges in $\mathbb{M}$ are independent. Since our single scan is based on the topological-optimum which follows the topology order of the transformed graph (i.e., DAG), $\mathbb{M}$ can contain edges in the subgraph with their source vertices has no in-edges (in-degree number of source vertices are zero). Using Fig. 7 as an example, since the in-degree of vertex a and b is zero, $\mathbb{M}_1$ contains edges of a and b, that is, $\{e_1, e_2, e_3, e_4\}$. We can hence use four threads to work on these four edges in parallel. During computation, we further get $\mathbb{M}_2$ with $\{e_5, e_6, e_7, e_8\}$. Here, again, we can assign various threads to work on these edges in parallel as shown in Algorithm 2. In summary, our single scan can be extended to support parallel processing.

D. Transformation Model

To tackle the graph amplification challenge, we propose a novel transformation model for further accelerating the execution phase (discussed previously) via *effectively condensing the transformed DAG size without any accuracy loss on graph information*. This can dramatically reduce the execution

workload and memory footprint to increase the chance for in-memory processing with reduced disk I/O.

From Fig. 1(a), we observe that after the transformation the same vertex has repeated appearances in a DAG at different time instances, e.g., vertex a is expanded to multiple new vertices $(a, 1)$, $(a, 2)$ and $(a, 6)$ at different timestamps 1, 2 and 6. All of them share the same parent vertex a. Due to this unique feature of the temporal graph, the transformed DAG may contain many redundant vertices that are unnecessary for information propagation in temporal graph execution (e.g., $(a,2)$). Specifically, we find that *when in-vertices are used to propagate information to out-vertices, it can lead to redundant vertices and edges*.

Fig. 8(a) demonstrates such redundancy in the state-of-the-art transformation (Sec. II-C). This example extracts all the time instances of the vertex b and their connectivity from a DAG (not shown here). For the in-vertices $(b, 1)$ and $(b, 3)$, since they only propagate e_1 and e_2 to next time instances, they are unnecessary for processing and can be merged into $(b, 2)$ and $(b, 4)$, respectively. After this, since the out-vertex $(b, 5)$ is not propagating information, it can be merged with the previous time instance $(b, 4)$. Based on this unique redundancy feature of temporal graph transformation, we propose a new transformation method applied prior to the execution, named hyper-method. It is used to reduce the size of the transformed graph by merging unnecessary vertices into the essential ones. Specifically, we define the original temporal graph as G and the transformed graph (DAG) as G'. We only pick the essential vertices for execution, called *hyper vertices*, and the related vertices that are redundant are merged with them. Our hyper-method guarantees the topological structure of DAG and the hyper vertices' target values are unchanged.

We formally define *hyper vertex*. A hyper vertex must be an out-vertex. The entire hyper vertices set is defined as $T_{hyper}[u] = \{(u, t) : (u, t) \in T_{out}[u]\}$ and satisfies at least one of the following criteria:

- **Criterion 1:** $\forall (u, t_1) \in T_{out}[u], \ t \leqslant t_1$
- **Criterion 2:** $\exists (u, t_1) \in T_{in}[u], \ t_1 \leqslant t \ \Rightarrow \forall (u, t_2) \in T_{out}[u], \ t_2 < t_1 \ or \ t_2 \geqslant t$

Using Fig. 8(a) as an example, there are two in-vertices and three out-vertices. Vertex $(b, 2)$ is a hyper vertex because it has the smallest time instance (Criterion 1). $(b, 4)$ is also a hyper vertex because there is no out-vertex between $(b, 3)$ and $(b, 4)$ (Criterion 2). However, $(b, 5)$ is not a hyper vertex. Thus, $T_{hyper}[b] = \{(b, 2), (b, 4)\}$. Furthermore, the merge operation for vertex $(u, t) \notin T_{hyper}[u]$ is performed if it satisfies at least one of the following conditions:

- **Merge Cond1:** When $(u, t) \in T_{in}[u]$, it can be merged to vertex $(u, t_1) \in T_{hyper}[u]$ if $t_1 = min \ \{t_i : \ (u, t_i) \in T_{hyper}[u] \ and \ t_i \geq t\}$
- **Merge Cond2:** When $(u, t) \in T_{out}[u]$, it can be merged to vertex $(u, t_1) \in T_{hyper}[u]$ if $t_1 = max \ \{t_i : \ (u, t_i) \in T_{hyper}[u] \ and \ t_i < t\}$

Merge Cond1 and Cond2 describe how an in-vertex and out-vertex is merged into a hyper vertex, respectively. Fig. 8(b)

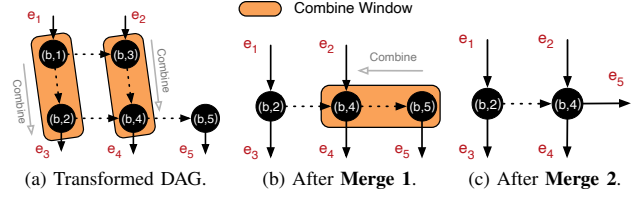(a) Transformed DAG. (b) After **Merge 1**. (c) After **Merge 2**.

Fig. 8: Hyper-method illustration. Dash arrows indicate the time order, not the actual edges in the DAG.

is reduced from Fig. 8(a) by **Merge Cond1**. Fig. 8(c) is reduced from Fig. 8(b) by **Merge Cond2**. We denote the merged graph as G'' (Fig. 8(c)). The target value of vertex v is denoted as $d[v]|G'$ for graph G' and $d[v]|G''$ for graph G''. We need to prove that for any vertex v in G' there will be $d[v]|G' = d[v]|G''$. To prove this, we separate the *real* edges from the *virtual* edges in G' and G''. The *real edges* are the edges that exist in the original graph G, while the *virtual edges* do not exist in G and they are created between the same vertex's different time instances, e.g., all the edges between (b, i) in Fig. 8(a). First, we have an observation that the virtual edges do not have weights ($w = 0$) and they have no impact on updating the target values. This is formally defined as Observation 1. Based on this, we can find that the target values are preserved with the same real edge sequence. It is defined and proved as Lemma 1. Finally, we propose Theorem 2 which shows that the hyper-method does not change the target values of vertices for the minimum (maximum) path problem.

Observation 1. *Given a virtual edge e which connects two vertices, i.e., (u, t_1) and (u, t_2), computing on e, i.e., $update((u, t_1), e)$, will not change the distance of the destination endpoint (u, t_2).*

Lemma 1. *Given two paths P_1, P_2 with the same real edge sequence, $target(P_1) = target(P_2)$.*

Proof. Given a path $P_1 = e_1 \cdot e_2 \cdot ... \cdot e_{n-1} \cdot e_n$, the target value is calculated by $target(P_1) = update(update(...update(u, e_1), ..., e_{n-1}), e_n)$. Because virtual edges do not change the target value, we can omit all virtual edges in the equation, i.e., the target value is calculated by all real edges. The new equation is as follows, in which e_{r_k} denotes the k-th real edge in the path P.

$$target(P_2) = update(update(...update(u, e_{r_1}), ..., e_{r_{k-1}}), e_{r_k})$$

According to Observation 1, $target(P_1) = target(P_2)$. $\square$

Theorem 2. $Target_{min}(u, v)$ *are the same in both G' and G''.*

Proof. Given any path P in G' from u to v, there exists a P' in G'' with the same real edge sequence. According to Lemma 1, $target(P) = target(P')$. By definition, $d[v] = min\{target(P) : P \in \mathbf{P}(u, v)\}$. Thus, $d[v]|G' \leq d[v]|G''$ because for any path to v in G'', there exists a path in G' with the same target value.

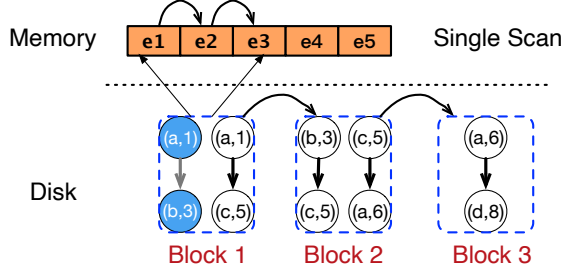Similarly, $d[v]|G'' \leq d[v]|G'$. Thus, $d[v]|G' = d[v]|G''$. $\square$

Fig. 9: TEGRAPH out-of-core data organization.

IV. TIME-AWARE GRAPH REPRESENTATION

To better support our single scan execution, TEGRAPH introduces a novel time-aware graph data organization. Particularly, our single scan execution requires processing the temporal graphs in a time elapsing order. At each time step, the graph is further traversed in a topological manner. Consequently, directly storing the vertices of the same label together would result in loading a tremendous amount of unnecessary data during processing. Using vertex a in Fig. 6(a) as an example, if we store the edges connected to vertex a, i.e., (a, 1)→(b, 3), (a, 1)→(c, 5), and (a, 6)→(d, 8) together, we will need to load them when either (a, 1) or (a, 6) is active. Clearly, when (a, 1) is active, loading the neighbors for (a, 6) is unnecessary, and vice versa.

The good news is that the graph topology of the transformed graph in Fig. 6(a) has already captured the temporal information. For example, if edge e_i has larger time instance than e_j, then only e_j can update e_i. In other words, e_i and e_j are regarded as different edges in the graph. Consequently, TEGRAPH can regard identical vertex IDs with dissimilar time instances as different vertices. *This design decouples the connection between vertices with the same vertex label but with different time instances.* With this further transformation, the transformed graph can be viewed as a new static DAG. For example, in Fig. 6(a), vertex $(a, 6)$ will be viewed as a different vertex from $(a, 1)$ on computing. Thus vertex $(c, 5)$ can directly update vertex $(a, 6)$ and will not affect vertex $(a, 1)$. Formally, during computation, for each vertex u, the value of vertex (u, t_{max}) (t_{max} is the maximum time instance of vertex u) will be the target value of u because there must be a path from each vertex of u ((u, t_i), $t_i < t_{max}$) to (u, t_{max}) and the value of (u, t_{max}) has been updated by vertex (u, t_i). In other words, this data update will not affect the correctness of the processing algorithms. For data representation, we can simply store this new static graph into the popular compressed sparse row format because the vertices with the same ID and time instance are grouped together.

Fig. 9 shows how to store Fig. 6(a) in an external-memory setting. Here we target external-memory setting because temporal graphs are often too large to fit in the main memory of the commodity computing systems. Basically, the entire graph is stored according to the increasing time instances. At each time step, identical vertices are grouped together. We further partition the graph into several blocks so that each block can

---

Algorithm 3 Out-of-core execution.

$Blocks = \{B_1, B_2, .., B_k\}$, E is partitioned into k blocks
for all $B_i \in Blocks$ **do**
    Parallel single scan in B_i

---

fit into the memory. As shown in Algorithm 3, during out-of-core execution, TEGRAPH applies the parallel single scan to the just loaded block. To further hide the data transfer cost, we overlap the computing of one block with the transfer of another due to the fact that the execution and disk I/O are independent. Furthermore, there are two optimizations proposed to enable time-aware format conversion: *vertex grouping* to optimize the vertex transformation, and *parallel relabeling* to optimize the edge transformation.

Vertex Grouping: All transformation algorithms, including our hyper-method in TEGRAPH, require to sort the vertices for vertex transformation, i.e., putting the vertices sets $T_{in}[u]$ and $T_{out}[u]$ in a logical array to relabel all the vertices into integers. The typical sorting method used in the current transformation phase (e.g., that in **Trans**) are common sorting algorithms (e.g., quick-sort) with 2-dimension comparison, resulting in the time complexity of $O(|E|log(D))$. But due to our *hyper-method*, we only need to sort the hyper vertices, e.g., (a, t). Since the second dimension (t, time instance) is already sorted due to the edge stream with increasing starting time, only the first dimension (i.e., vertex label) is needed to be sorted with the second dimension unchanged. Then the cumulative flow diagram is constructed to group all vertices (u, t) for the same vertex u with its time instances relatively positioned in the edge stream. When the vertex u is grouped, all vertices (u, t) are sorted. This brings down the sorting time complexity from $O(|E|log(D))$ to $O(|E|)$.

Parallel Relabeling: After vertex transformation, TEGRAPH needs to identify which hyper vertices can be further merged by using our *hyper-method*. For each in-vertex, we need to find a hyper vertex with the minimum time instance no earlier than it. And for each out-vertex, the corresponding hyper vertex to merge must have the maximum time instance no later than it. As there are no data dependencies during searching, the relabeling can be done in parallel with $O(\frac{|E|log(D)}{Threads\_number})$ time complexity. The time complexity of this process is near $O(|E|)$ under multiple threads and the overall graph transformation time complexity is also $O(|E|)$.

Streaming Graph Support: For each newly arrived edge, we need to update the transformed graph. Because these new edges have bigger time instances than the existing ones [36], we can simply create new hyper vertices for those new arrivals. Therefore, the average updating time complexity is $O(1)$, similarly for the update to the distance value of the new edges.

A. I/O Complexity Analysis

This section quantitatively analyzes the I/O complexity of TEGRAPH. Note that One-Pass and Trans do not provide out-of-core execution support due to the large memory overhead (Sec. II-D). For baseline, we choose **GridGraph** [3] and **Wonderland** [2] which are suitable for high diameter graphs because the transformed graph (DAG) often has a longer

diameter. Specifically, we feed transformed graphs into these out-of-core graph engines and apply static graph algorithms in the transformed graphs. Note that they operate on the same transformed DAG as we do. Also, since other out-of-core engines such as LUMOS [66] are not optimized for high-diameter graphs, we do not analyze them here.

TABLE II: Quantitative analysis of out-of-core execution.

Systems	TEGRAPH	GridGraph	Wonderland
Complexity	min(|E|/S,|E|/C)	N*β*(|E|/S+|E|/C/T)	N*γ*(|E|/S+|E|/C/T)

Table II shows the quantitative analysis on out-of-core execution performance of TEGRAPH, **GridGraph** and **Wonderland**. $|E|$ represents the number of edges; S is the sequential read bandwidth of the disk; C is the CPU processing speed (edges/sec); N is the iteration number; β is the proportion of the average active graph data of GridGraph and γ for Wonderland; T is the thread number. We can ignore the abstraction computation overhead of Wonderland because the abstraction size is usually small. We observe that the out-of-core performance for all three is bounded by graph size, I/O bandwidth, and CPU speed.

Table II suggests that TEGRAPH always outperforms **Grid-Graph** and **Wonderland** in theory. Also, the performance of **GridGraph** and **Wonderland** is heavily impacted by the nature of the input graphs and algorithms, which is reflected in the parameters N, β, or γ. However, because of our hyper-method together with only a single round of sequential readings on the transformed graph, TEGRAPH significantly reduces the random memory accesses on edges and iterations (N). Furthermore, TEGRAPH can pipeline the loading of data blocks with the execution phase to hide I/O latency.

V. TEGRAPH PROGRAMMING INTERFACE

TEGRAPH provides intuitive programming model and APIs for users to easily express general path problems and their applications on temporal graphs via simple target function update (Sec. III-B). Its key interfaces are described below.

In the framework of TEGRAPH, each edge is denoted as a 5-tuple (source, dist, start, end, weight), representing the source and destination vertices, start and end time, and the weight for the edge. Each vertex is represented by a 2-tuple (vid, d), indicating its vertex instance and property (i.e., the target value of vid), respectively. The framework inputs sid, did, V and E, and returns the query answer from sid to did, where sid is the source vertex of each query, did is the target vertex, V represents the vertices set and E is the edges set. In general, the framework works by sequentially invoking the following interface functions: **TRANSFORM()**, **VINIT()**, **EMAP()** and **VAGGRE()**. **TRANSFORM()** transforms the graph into a static DAG only once, and then the transformed graph can be reused by various computations afterward. **VINIT()** is used to initialize the property of each vertex through the ***Init*** function provided by users. **EMAP()** is used to update all edges based on our single scan algorithm. Each edge will modify the properties of the corresponding vertices. **EMAP()** calls the

user-defined function ***Update*** to calculate the new distance for each edge, and another user-defined function ***Compare*** to update the distance of the current edge's destination vertex. **VAGGRE()** is provided to aggregate the properties of vertices with the same parent vertex instance to output the final result. To output the final distance of the target vertex did, the distance of each vertex v in $T_{hyper}[did]$ is aggregated by ***Compare***. It needs to be stressed that users only need to define ***Init***, ***Update***, and ***Compare*** functions, according to certain applications.

We take the shortest path as an example to show the usage of APIs. For the ***Init*** function, we set the values for the source vertices (sid) to 0 and the others to $+\infty$. The ***Update*** function is the same as that in the traditional shortest path algorithms, i.g., edge's weight plus the distance of the current vertex to achieve a new distance. The ***Compare*** function chooses the smaller distance between the two distances. With our general programming model, users only need to provide a few lines of code for implementing different temporal graph applications.

VI. EVALUATION

A. Experiment Settings

Testbed: All the evaluations are conducted on a server using Duo Intel(R) Xeon(R) Processors E5-2640 v2 @ 2.00GHz (8-cores/processor, total 16 threads) with 20MB L3 Cache, 32GB DRAM, and a 1TB SSD drive with a throughput around 650 MB/s for sequential reads. TEGRAPH is implemented in around 3000 lines of C++ code and uses OpenMP for multithreading (16 threads). To further evaluate TEGRAPH's out-of-core execution efficiency, we apply $cgroup$ to set various memory limits based on the input datasets. The system fundamentals of TEGRAPH are briefly described as follows.

TABLE III: Real-World Temporal Graph Datasets(k=10^3).

Graph	Vertices ($|V|$)	Edges ($|E|$)	$Avg(D)$	$Max(D)$
Flickr [67]	2.3M	33.1M	28.8	3.42E4
Growth [68]	1.8M	39.9M	42.7	2.27E5
Edit [69]	21.5M	266.8M	21.1	3.27E6
Delicious [70]	33.8M	301.2M	66.7	4.36E6
Twitter [71]	41.7M	1.47B	70.5	6.42E6
UK-Union [72]	134M	5.51B	70.3	3.04E6

Input Graphs: We selected six popular benchmarks in Table III for our evaluation. Edges in the temporal graph edge stream are sorted by their starting time. Note that we select to use large static graphs *Twitter* and *UK-Union* because publicly available temporal graphs are not large enough for evaluation. For large static graphs which have not been attributed with time instances, we randomly allocate time instances to them according to the temporal graph format that is consistent with the state-of-the-art works. Main attributes of these graphs are listed in Table III. Note that $Avg(D)$ and $Max(D)$ represent the average and maximum vertex degree, respectively. They are important attributes for applications whose processing overhead can be significantly affected by the vertex degrees.

Applications: We evaluate four typical temporal path applications: reachability, fastest path, shortest path, and top-k nearest neighbors. For the top-k nearest neighbors, we use

the shortest path to calculate the top k nearest neighbors and choose k as $|V|/10$. We randomly select 100 vertices as input and report the average execution time for each application (single-source query).

Evaluation Methodology: Traditional static graph computation models such as vertex- or edge-centric approaches will lead to high overhead because they require additional efforts to process time constraints recorded on the topological structure. When applied to time-aware format in Sec. IV, we find TEGRAPH can work with both approaches. We evaluate the performance of TEGRAPH under two modes: (1) the full memory mode which puts all the graph data inside memory; and (2) the out-of-core mode uses one SSD drive to hold the graph data. The full memory mode demonstrates the efficiency of TEGRAPH compared to two state-of-the-art works discussed in Sec. II-C: **One-Pass** [32] (static execution) and **Trans** [52] (transformation-based execution). **One-Pass** and **Trans** have not been open-sourced and we implement them according to the original paper as fast as possible. Besides, we also test **A*** in Sec. II-C with the transformed graph of **Trans** as the input. We evaluate the performance improvement from each component in TEGRAPH including parallel single scan, hyper-method based transformation, and the time-aware graph transformation. For the out-of-core mode, we compare TEGRAPH with **GridGraph** [3] and **Wonderland** [2] as discussed in Sec. IV-A. We use their most current official versions. For fairness, all engines are fed with the same transformed graphs that are the outputs of our hyper-method based graph transformation. We report the execution time breakdown for both graph partition and execution.

B. Overall Performance: Full Memory Mode

In this section, we provide the overall performance analysis and comparison across different designs under the full memory mode to emphasize TEGRAPH's contribution as a whole and its impact on different temporal path applications. The reported overall performance does not include the execution time for graph transformation for both TEGRAPH and Trans because the same transformed graph can be used by many applications to average out the graph transformation time. The graph transformation will be discussed independently in Sec. VI-D. In Sec. VI-C, we will demonstrate the piecewise contribution breakdown from our proposed techniques in TEGRAPH. Table IV illustrates the overall execution time comparison on all datasets in Table III except *UK-Union* which can not fit the main memory (i.e., takes 88GB) and will be used for out-of-core testing. The evaluation uses 16 threads for multithreading and the scalability is shown in Sec. VI-C. Overall, we observe that TEGRAPH outperforms **One-Pass**, **Trans**, and **A*** for every data point, with speedups from $1.69\times$ (Top KNN on Flickr) to $241.18\times$ (Shortest Path on Twitter). TEGRAPH also performs better with larger graphs.

Reachability: Since the algorithm for reachability used in Trans and A* is the same as the fastest path, we only compare TEGRAPH and **One-Pass** here. Table IV shows that TEGRAPH can achieve a speedup up to $16.74\times$ over One-Pass. Although

both designs have the same theoretical time complexity (Table I), TEGRAPH outperforms One-Pass in three aspects. First, due to the topological-optimum nature, TEGRAPH only needs to use bitwise operations to update an edge, i.e. $d[v] \mid = d[u]$ for (u, v, s, t), while One-Pass requires more operations for updating, i.e. $if\ d[u] \leq s,\ d[v] = min\{d[v],\ t\}$. Second, in reachability, TEGRAPH only requires to process two variables for each edge (the source and destination vertex labels) while One-Pass requires four (i.e., start and end time additionally) because of the extra time dimension. Third, although TEGRAPH benefits from our multithreading design (e.g., $3\times$ to $4\times$ under 16 threads), we observe that the parallel versions for One-Pass and Trans [52] both perform worse than their single-thread versions due to the large message-passing overhead.

Fastest Path: For **One-Pass**, TEGRAPH processes $26.71 \sim 112.61\times$ faster than One-Pass on different workloads. Shown in Table I, for fastest path, the time complexity of One-Pass is $O(|E|log(D))$ while TEGRAPH is $O(|E|)$. With a larger graph, $log(D)$ tends to be larger which further benefits TEGRAPH. For **Trans**, it applies BFS-based algorithm to the fastest path [32]. However, TEGRAPH can still achieve $7.65 \sim 16.30\times$ speedup over Trans due to three reasons: (1) although both designs have the same theoretical time complexity, Trans consumes more time in each iteration to maintain each vertex's degree (up to 36% of the total execution time); (2) the graph size after transformation is $1.84 \sim 2.56\times$ larger than that from TEGRAPH, resulting in more computation and cache misses; (3) parallel single scan can also achieve a $2.69 \sim 4.11\times$ speedup than the single-threaded version which is studied in Sec. VI-C. Compared to **A***, TEGRAPH gets $3.75 \sim 13.04\times$ speedup. For each update, A* has to update the three functions of its neighbors (studied in Sec. II-C). However, TEGRAPH needs only one single operation for each edge update, while parallel single scan can speed up the execution process as well. A* has nearly $O(|E|)$ time complexity but has larger constants than TEGRAPH. Additionally, the input transformed graph of A* is the same with that of Trans, which is $2.09\times$ larger than TEGRAPH on average.

Shortest Path: TEGRAPH achieves significant performance gains over both **One-Pass** and **Trans**: up to $90.48\times$ and $241.18\times$, respectively. The performance gap comes from TEGRAPH's lower time complexity which reduces from $O(|E|log(D))$ (**One-Pass**) and $O(|E|log(|E|))$ (**Trans**) to $O(|E|)$ as shown in Table I. When compared to **A***, TEGRAPH can get $4.05 \sim 17.36\times$ speedup. The improvement comes from the smaller computation constants, transformed graph of TEGRAPH over A*, and the parallel single scan of TEGRAPH. Note, even using the same transformed graph as TEGRAPH, A* still takes $8.74\times$ longer time than TEGRAPH.

Top-K Nearest Neighbors: Its computation consists of two portions: calculating the shortest path for all vertices and using a priority queue to find k vertices with the smallest shortest path. Table IV shows that compared to **One-Pass**, **Trans**, and **A***, TEGRAPH achieves up to a $39.74\times$, $104.99\times$, and $8.08\times$ speedup, respectively. The reasons for these speedups are analogous to the reasons in the shortest path.

TABLE IV: Overall Execution Time in Seconds.

dataset	reachability		fastest path				shortest path				top k nearest neighbors			
	TeGraph	One-Pass	TeGraph	One-Pass	Trans	A*	TeGraph	One-Pass	Trans	A*	TeGraph	One-Pass	Trans	A*
Flickr	**0.0086**	0.0907	**0.0987**	2.6368	0.7546	0.3697	**0.0986**	1.2522	5.1928	0.3998	**0.4324**	1.5859	5.5265	0.7336
Growth	**0.0162**	0.1794	**0.1446**	4.5404	1.1879	1.0572	**0.1458**	2.8376	20.610	1.3602	**0.4985**	3.1903	20.962	1.7130
Edit	**0.1131**	1.6978	**0.6184**	64.574	6.1619	5.6689	**0.6213**	42.793	133.67	9.8426	**4.2825**	46.455	137.33	13.504
Delicious	**0.1942**	3.3769	**1.1095**	117.60	15.798	13.428	**1.1140**	89.362	254.06	15.243	**7.0614**	95.310	260.01	21.191
Twitter	**1.3613**	22.794	**6.1952**	697.64	100.98	80.785	**6.2266**	563.41	1501.7	108.11	**14.380**	571.56	1509.9	116.27

Fig. 10: Breakdown.

Fig. 11: Scalability.

TABLE V: Transformed Graph Size (k=10^3).

	TeGraph		Trans	
Dataset	|V|	|E|	|V|	|E|
Flickr	5556k	36732k	38718k	69555k
Growth	5789k	47980k	50351k	88433k
Edit	31319k	277170k	374866k	620193k
Delicious	42036k	310319k	529894k	797300k
Twitter	75792k	1502505k	1508208k	2934921k
UK-Union	295037k	5805037k	6339373k	11849373k

Fig. 12: Graph transformation time breakdown.
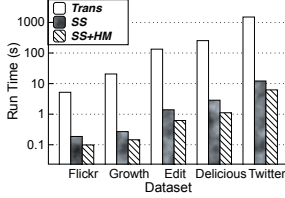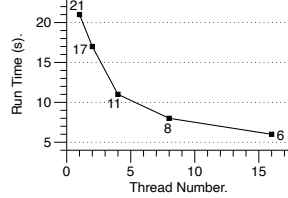
End-to-end Comparison: Although graph transformation can be used for all applications to amortize the graph transformation time, under the end-to-end comparison, TEGRAPH can still get up to $1.17 \sim 7.18\times$ speedup than One-Pass which does not need the preprocessing. This is because the end-to-end execution only needs $O(|E|)$ time complexity, whereas One-Pass needs $O(|E|log(D))$ time complexity.

C. Piecewise Breakdown

To evaluate the performance improvement gained through our optimizations, topological single scan (multithreading) and hyper-method, we compare the execution phase of TEGRAPH with **Trans** under different optimizations, i.e., only single scan (SS) and single scan together with hyper-method (SS+HM) shown in Fig. 10. Then we will show the scalability of our parallel single scan method in Fig. 11.

(I) Impact of Topological Single Scan: Fig. 10 shows that TEGRAPH's topological single scan achieves up to $124\times$ speedup over the priority-queue based Dijkstra's algorithm applied in Trans on the *Twitter* dataset. Even on the smallest dataset *Flickr*, the speedup is around $27.8\times$. This significant performance improvement comes directly from the reduction of time complexity from $O(|E|log(|E|))$ of priority-queue based Dijkstra's to $O(|E|)$ of TEGRAPH's single scan. Additionally, our single scan converts the random memory accesses to sequential ones which further reduces cache misses. Furthermore, we study the scalability impact of multithreading in our single scan. In Fig. 11, we take the *Twitter* dataset as an example to demonstrate our single scan's scalability. Compared to the single-threaded version, multithreading not only needs to update each edge but also requires maintaining the degree number of vertices as seen in Sec. III-C. This will introduce more operations, e.g., the updating operation will be atomic to ensure correctness. Although this can affect the scalability, we still observe an overall $2.69 \sim 4.11\times$ speedup.

(II) Impact of Hyper-Method: Hyper-method aims to eliminate unnecessary redundancy in order to (1) reduce the size of the transformed graph which lowers the memory consumption, and (2) improve the core execution's performance
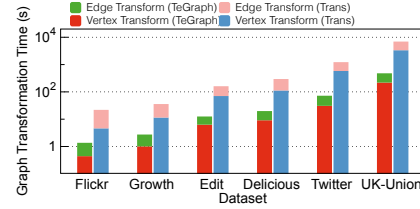
with a smaller workload. For (1), Table V demonstrates that, compared to **Trans**, TEGRAPH can reduce the total number of edges and vertices by up to 61% and 94.9%. Since TE-GRAPH's time complexity is $O(|E|)$, reducing edges' number directly affects the processing time. Meanwhile, reduction of vertices benefits the out-of-core execution because vertices are more likely to be held in the main memory. For (2), we use the shortest path problem as an example to showcase our *hyper-method*'s impact on core execution. Fig. 10 shows that hyper-method contributes up to $2.6\times$ performance improvement over the TEGRAPH version without it.

D. Time-Aware Graph Transformation

We evaluate the effectiveness of the two steps introduced to transform the temporal graph into our time-aware format. Specifically, we compare the transformation phase of TE-GRAPH with that in **Trans**. Note that *UK-Union* is transformed under the out-of-core mode for both engines and we report the overall execution time. As discussed previously, there are two steps in graph transformation: vertex and edge transformation. TEGRAPH applies vertex grouping and parallel relabeling to accelerate these two steps respectively. Fig. 12 shows the graph transformation time breakdown under two designs. The vertex grouping for optimizing vertex transformation can get up to $18\times$ speed up when compared to the sorting method in Trans. When the datasets get bigger, vertex grouping can retain better improvement because our time complexity of vertex grouping is near to $O(|E|)$ but the time complexity of the traditional sorting method in Trans is $O(|E|log(D))$. That means Trans will be greatly affected by degrees. For parallel relabeling
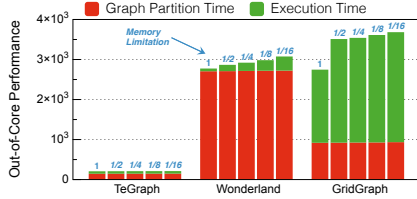
Fig. 13: Out-of-core evaluation on the UK-Union.

which is used to optimize edge transformation, TEGRAPH uses $OpenMP$ in C++ that presents good scalability. Particularly, we achieve nearly $15\times$ speedup under 16 threads concurrency. Overall, TEGRAPH obtains up to $16.97\times$ speedup on graph transformation over Trans.

E. Out-of-Core Execution

To prove our the quantitative performance modeling on different out-of-core execution designs in Table II, we experimentally evaluate TEGRAPH against two state-of-the-art out-of-core engines on the *UK-Union* dataset, **GridGraph** and **Wonderland**. For testing, we use *cgroup* to limit the memory consumption to all three engines. Specifically, we evaluate the systems using the memory sizes of S, $S/2$, $S/4$, $S/8$, and $S/16$, where S is the size of the transformed graph. For fair evaluation, we feed them with the same transformed graph from TEGRAPH. We list both the graph partition and execution time. Note that the graph partition time here is for partitioning the graph into different blocks to fit the memory size for out-of-core execution, i.e. 2D partition for GridGraph, 2D partition plus abstraction finding (for iteration reduction) for Wonderland, and 1D partition for TEGRAPH.

Fig. 13 shows the performance comparison with the two-phase breakdown. Overall, TEGRAPH outperforms the other two engines in both graph partition and execution. For graph partition, TEGRAPH partitions the graph into blocks linearly with very low overhead, while Wonderland requires much longer time on graph partition to complete both 2D partition and abstraction searching. GridGraph has better processing than Wonderland because of no abstraction optimization overhead. For execution, both GridGraph and Wonderland read edges and modify the properties of vertices until convergence. However, these iterative methods still perform poorly on long diameter graphs such as the transformed DAGs here. More iterations could lead to larger disk I/O access. On the contrary, TEGRAPH only reads the dataset once from the disk with the disk I/O linearly proportion to $|E|$. Thus, GridGraph performs much worse than TEGRAPH for graph execution even under the full memory mode. Wonderland proposes a graph abstraction mechanism to reduce the iteration number on GridGraph but is still slower than TEGRAPH especially for the lower memory modes because TEGRAPH only requires one iteration regardless of the memory size. Our TEGRAPH outperforms Wonderland and GridGraph by up to $5\times$ and $43.4\times$ respectively for the execution time. Disk I/O occupies 79% of the total execution time in TEGRAPH, 74.5% of Wonderland, and 47.9% of GridGraph, due to the fast execution algorithm

in TEGRAPH. When the memory limit is under $S/16$, the disk I/O of Wonderland is $4.71\times$ larger than TEGRAPH and GridGraph is $26.3\times$ larger than TEGRAPH. For the memory usage, under different memory limits, all three engines adjust the partitioned graph size to make the maximum usage of the main memory for disk I/O reduction.

VII. RELATED WORK

The existing temporal path problems are solved by using ad-hoc solutions or application-specific optimizations. There are some other applications that adopt temporal path problems as their core component including online reachability [53], [73], betweenness centrality [55], minimum spanning tree problem [59], widest path [74], random walk [23], and components problem [75], [76]. These applications are all based on path problems, thus they can all benefit from our TEGRAPH.

For temporal graph processing, several static-execution based engines [35], [50], [51], [77], [78] have been proposed, but they are slower than our static-execution baseline ONE-PASS due to high time complexity of $O(|E|Dlog(|V|D))$. Some distributed works have been proposed such as GRAPHITE [79] which optimizes temporal partitioning and message passing, but they mainly focus on the system-level optimizations without fundamentally reducing algorithms' time complexity. Some temporal graph database systems [46]–[48], [80] proposed to mainly focus on traversal language which provides the temporal syntax for temporal graph otherwise algorithm and system-levels optimizations of temporal graph applications. There also exist efforts that focus on graph storage strategy such as GraphOne [45] and LiveGraph [81]. These attempts use graph data stores to provide specific graph storage formats for real-time analysis on temporal graphs. Whereas, all these systems fall short at finding a temporal-aware format for temporal graphs. Together with our topological-optimum and single scan, TEGRAPH can further speed up temporal graph computing.

We also notice that several studies have proposed specific optimizations on static graph reduction [82], [83]. However, they cannot replace our hyper-method because hyper-method guarantees accuracy and correctness without losing any information, while almost all these reduction methods can only perform approximation.

VIII. CONCLUSION

In this paper, we propose TEGRAPH, the first general-purpose temporal graph computing engine for efficiently processing temporal path problems and their applications. The core of TEGRAPH is its temporal information-aware graph format, novel single-pass execution workflow, and information redundancy removing transformation models which effectively address the computation, space, and bandwidth challenges. Taken together, TEGRAPH can achieve up to 61% disk space-saving, up to $17\times$ graph transformation speedup, and up to two orders of magnitude performance speedup over the state-of-the-art designs.

References

[1] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: large-scale graph computation on just a pc," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 31–46.

[2] M. Zhang, Y. Wu, Y. Zhuo, X. Qian, C. Huan, and K. Chen, "Wonderland: A novel abstraction-based out-of-core graph processing system," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, 2018, pp. 608–621. [Online]. Available: https://doi.org/10.1145/3173162.3173208

[3] X. Zhu, W. Han, and W. Chen, "Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015, pp. 375–386.

[4] D. M. Miller, D. G. Espinosa-heidmann, J. Legra, S. R. Dubovy, I. J. Sűner, D. D. Sedmak, R. D. Dix, and S. W. Cousins, "The association of prior cytomegalovirus infection with neovascular age-related macular degeneration," *American journal of ophthalmology*, vol. 138, no. 3, pp. 323–328, 2004.

[5] A. McCrabb, E. Winsor, and V. Bertacco, "DREDGE: dynamic repartitioning during dynamic graph execution," in *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*, 2019, p. 28. [Online]. Available: https://doi.org/10.1145/3316781.3317804

[6] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, "Mosaic: Processing a trillion-edge graph on a single machine," in *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, 2017, pp. 527–543. [Online]. Available: https://doi.org/10.1145/3064176.3064191

[7] K. Vora, G. H. Xu, and R. Gupta, "Load the edges you need: A generic I/O optimization for disk-based graph processing," in *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, 2016, pp. 507–522. [Online]. Available: https://www.usenix.org/conference/atc16/technical-sessions/presentation/vora

[8] V. Balaji and B. Lucia, "When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs," in *2018 IEEE International Symposium on Workload Characterization, IISWC 2018, Raleigh, NC, USA, September 30 - October 2, 2018*, 2018, pp. 203–214. [Online]. Available: https://doi.org/10.1109/IISWC.2018.8573478

[9] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, 2016, pp. 301–316. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhu

[10] Y. Cheng, H. Jiang, F. Wang, Y. Hua, D. Feng, W. Guo, and Y. Wu, "Using high-bandwidth networks efficiently for fast graph computation," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 5, pp. 1170–1183, 2019. [Online]. Available: https://doi.org/10.1109/TPDS.2018.2875084

[11] V. Balaji and B. Lucia, "Combining data duplication and graph reordering to accelerate parallel graph processing," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2019, Phoenix, AZ, USA, June 22-29, 2019*, J. B. Weissman, A. R. Butt, and E. Smirni, Eds. ACM, 2019, pp. 133–144. [Online]. Available: https://doi.org/10.1145/3307681.3326609

[12] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel, "Chaos: scale-out graph processing from secondary storage," in *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, E. L. Miller and S. Hand, Eds. ACM, 2015, pp. 410–424. [Online]. Available: https://doi.org/10.1145/2815400.2815408

[13] S. Grossman, H. Litz, and C. Kozyrakis, "Making pull-based graph processing performant," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*, A. Krall and T. R. Gross, Eds. ACM, 2018, pp. 246–260. [Online]. Available: https://doi.org/10.1145/3178487.3178506

[14] V. Kalavri, V. Vlassov, and S. Haridi, "High-level programming abstractions for distributed graph processing," *IEEE Trans. Knowl. Data Eng.*, vol. 30, no. 2, pp. 305–324, 2018. [Online]. Available: https://doi.org/10.1109/TKDE.2017.2762294

[15] A. Gautreau, A. Barrat, and M. Barthélemy, "Microdynamics in stationary complex networks," *Proceedings of the National Academy of Sciences*, vol. 106, no. 22, pp. 8847–8852, 2009.

[16] P. Holme and J. Saramäki, "Temporal networks," *Physics reports*, vol. 519, no. 3, pp. 97–125, 2012.

[17] V. Nicosia, J. Tang, C. Mascolo, M. Musolesi, G. Russo, and V. Latora, "Graph metrics for temporal networks," in *Temporal networks*. Springer, 2013, pp. 15–40.

[18] T. M. Przytycka, M. Singh, and D. K. Slonim, "Toward the dynamic interactome: it's about time," *Briefings in Bioinformatics*, vol. 11, no. 1, pp. 15–29, 2010. [Online]. Available: https://doi.org/10.1093/bib/bbp057

[19] J. Tang, S. Scellato, M. Musolesi, C. Mascolo, and V. Latora, "Small-world behavior in time-varying graphs," *Physical Review E*, vol. 81, no. 5, p. 055101, 2010.

[20] S. Zhang, L. Yao, A. Sun, and Y. Tay, "Deep learning based recommender system: A survey and new perspectives," *ACM Computing Surveys (CSUR)*, vol. 52, no. 1, p. 5, 2019.

[21] T. Arnoux, L. Tabourier, and M. Latapy, "Combining structural and dynamic information to predict activity in link streams," in *Proceedings of the 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2017*. ACM, 2017, pp. 935–942.

[22] J. Viard and M. Latapy, "Identifying roles in an ip network with temporal and structural density," in *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2014, pp. 801–806.

[23] S. Huang, Z. Bao, G. Li, Y. Zhou, and J. S. Culpepper, "Temporal network representation learning via historical neighborhoods aggregation," in *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 2020, pp. 1117–1128. [Online]. Available: https://doi.org/10.1109/ICDE48307.2020.00101

[24] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou, "Aligraph: A comprehensive graph neural network platform," *PVLDB*, vol. 12, no. 12, pp. 2094–2105, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol12/p2094-zhu.pdf

[25] Y. Zhao, X. Wang, H. Yang, L. Song, and J. Tang, "Large scale evolving graphs with burst detection," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, 2019, pp. 4412–4418. [Online]. Available: https://doi.org/10.24963/ijcai.2019/613

[26] Z. Abbas, V. Kalavri, P. Carbone, and V. Vlassov, "Streaming graph partitioning: An experimental study," *Proc. VLDB Endow.*, vol. 11, no. 11, pp. 1590–1603, 2018. [Online]. Available: http://www.vldb.org/pvldb/vol11/p1590-abbas.pdf

[27] M. Li, F. M. Choudhury, R. Borovica-Gajic, Z. Wang, J. Xin, and J. Li, "Crashsim: An efficient algorithm for computing simrank over static and temporal graphs," in *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 2020, pp. 1141–1152. [Online]. Available: https://doi.org/10.1109/ICDE48307.2020.00103

[28] Y. Liu, K. Zhao, G. Cong, and Z. Bao, "Online anomalous trajectory detection with deep generative sequence modeling," in *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 2020, pp. 949–960. [Online]. Available: https://doi.org/10.1109/ICDE48307.2020.00087

[29] T. Chen, H. Yin, Q. V. H. Nguyen, W. Peng, X. Li, and X. Zhou, "Sequence-aware factorization machines for temporal predictive analytics," in *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 2020, pp. 1405–1416. [Online]. Available: https://doi.org/10.1109/ICDE48307.2020.00125

[30] H. Liu and H. H. Huang, "SIMD-X: programming and processing of graph algorithms on gpus," in *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, D. Malkhi and D. Tsafrir, Eds. USENIX Association, 2019, pp. 411–428. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/liu-hang

[31] S. Pandey, L. Li, A. Hoisie, X. S. Li, and H. Liu, "C-saw: A framework for graph sampling and random walk on gpus," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–15.

[32] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu, "Path problems in temporal graphs," *Proceedings of the VLDB Endowment*, vol. 7, no. 9, pp. 721–732, 2014.

[33] S. Kumar, X. Zhang, and J. Leskovec, "Predicting dynamic embedding trajectory in temporal interaction networks," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, A. Teredesai, V. Kumar, Y. Li, R. Rosales, E. Terzi, and G. Karypis, Eds. ACM, 2019, pp. 1269–1278. [Online]. Available: https://doi.org/10.1145/3292500.3330895

[34] Y. Zheng, Z. Li, J. Xin, and G. Zhou, "A spatial-temporal graph based hybrid infectious disease model with application to covid-19," in *10th International Conference on Pattern Recognition Applications and Methods*, 2021.

[35] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen, "Chronos: a graph engine for temporal graph analysis," in *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, 2014, pp. 1:1–1:14. [Online]. Available: https://doi.org/10.1145/2592798.2592799

[36] G. H. Nguyen, J. B. Lee, R. A. Rossi, N. K. Ahmed, E. Koh, and S. Kim, "Continuous-time dynamic network embeddings," in *Companion of the The Web Conference 2018 on The Web Conference 2018, WWW 2018, Lyon , France, April 23-27, 2018*, 2018, pp. 969–976. [Online]. Available: https://doi.org/10.1145/3184558.3191526

[37] R. Trivedi, M. Farajtabar, P. Biswal, and H. Zha, "Dyrep: Learning representations over dynamic graphs," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019. [Online]. Available: https://openreview.net/forum?id=HyePrhR5KX

[38] H. Cai, V. W. Zheng, and K. C. Chang, "A comprehensive survey of graph embedding: Problems, techniques, and applications," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 9, pp. 1616–1637, 2018.

[39] M. Starnini, A. Baronchelli, A. Barrat, and R. Pastor-Satorras, "Random walks on temporal networks," *Physical Review E*, vol. 85, no. 5, p. 056115, 2012.

[40] M. Mariappan and K. Vora, "Graphbolt: Dependency-driven synchronous processing of streaming graphs," in *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, G. Candea, R. van Renesse, and C. Fetzer, Eds. ACM, 2019, pp. 25:1–25:16. [Online]. Available: https://doi.org/10.1145/3302424.3303974

[41] K. Vora, R. Gupta, and G. H. Xu, "Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, Y. Chen, O. Temam, and J. Carter, Eds. ACM, 2017, pp. 237–251. [Online]. Available: https://doi.org/10.1145/3037697.3037748

[42] ——, "Synergistic analysis of evolving graphs," *TACO*, vol. 13, no. 4, pp. 32:1–32:27, 2016. [Online]. Available: https://doi.org/10.1145/2992784

[43] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[44] R. Bellman, "On a routing problem," *Quarterly of applied mathematics*, vol. 16, no. 1, pp. 87–90, 1958.

[45] P. Kumar and H. H. Huang, "Graphone: A data store for real-time analytics on evolving graphs," in *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*, A. Merchant and H. Weatherspoon, Eds. USENIX Association, 2019, pp. 249–263. [Online]. Available: https://www.usenix.org/conference/fast19/presentation/kumar

[46] X. Chen, C. Zhang, B. Ge, and W. Xiao, "Temporal social network: Storage, indexing and query processing." in *EDBT/ICDT Workshops*, 2016.

[47] C. Kankanamge, S. Sahu, A. Mhedhbi, J. Chen, and S. Salihoglu, "Graphflow: An active graph database," in *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, 2017, pp. 1695–1698. [Online]. Available: https://doi.org/10.1145/3035918.3056445

[48] W. Han, K. Li, S. Chen, and W. Chen, "Auxo: a temporal graph management system," *Big Data Mining and Analytics*, vol. 2, no. 1, pp. 58–71, 2018.

[49] B. B. Xuan, A. Ferreira, and A. Jarry, "Computing shortest, fastest, and foremost journeys in dynamic networks," *International Journal of Foundations of Computer Science*, vol. 14, no. 02, pp. 267–285, 2003.

[50] M. Steinbauer and G. Anderst-Kotsis, "Dynamograph: extending the pregel paradigm for large-scale temporal graph processing," *International Journal of Grid and Utility Computing*, vol. 7, no. 2, pp. 141–151, 2016.

[51] B. Erb, D. Meißner, F. Kargl, B. A. Steer, F. Cuadrado, D. Margan, and P. R. Pietzuch, "Graphtides: a framework for evaluating stream-based graph processing platforms," in *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Houston, TX, USA, June 10, 2018*, A. Arora, A. Bhattacharya, G. H. L. Fletcher, J. L. Larriba-Pey, S. Roy, and R. West, Eds. ACM, 2018, pp. 3:1–3:10. [Online]. Available: https://doi.org/10.1145/3210259.3210262

[52] H. Wu, J. Cheng, Y. Ke, S. Huang, Y. Huang, and H. Wu, "Efficient algorithms for temporal path computation," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 11, pp. 2927–2942, 2016.

[53] H. Wu, Y. Huang, J. Cheng, J. Li, and Y. Ke, "Reachability and time-based path queries in temporal graphs," in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 2016, pp. 145–156.

[54] D. Kempe, J. Kleinberg, and A. Kumar, "Connectivity and inference problems for temporal networks," *Journal of Computer and System Sciences*, vol. 64, no. 4, pp. 820–842, 2002.

[55] R. Pfitzner, I. Scholtes, A. Garas, C. J. Tessone, and F. Schweitzer, "Betweenness preference: Quantifying correlations in the topological dynamics of temporal networks," *Physical review letters*, vol. 110, no. 19, p. 198701, 2013.

[56] R. K. Pan and J. Saramäki, "Path lengths, correlations, and centrality in temporal networks," *CoRR*, vol. abs/1101.5913, 2011. [Online]. Available: http://arxiv.org/abs/1101.5913

[57] D. E. J. Wang, "Fast approximation of centrality," *Graph Algorithms and Applications*, vol. 5, no. 5, p. 39, 2006.

[58] S. Huang, J. Cheng, and H. Wu, "Temporal graph traversals: Definitions, algorithms, and applications," *CoRR*, vol. abs/1401.1919, 2014. [Online]. Available: http://arxiv.org/abs/1401.1919

[59] S. Huang, A. W.-C. Fu, and R. Liu, "Minimum spanning trees in temporal graphs," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 419–430.

[60] A. Andreiana, C. Badica, and E. Ganea, "An experimental comparison of implementations of dijkstra's single source shortest path algorithm using different priority queues data structures," in *24th International Conference on System Theory, Control and Computing, ICSTCC 2020, Sinaia, Romania, October 8-10, 2020*, L. Barbulescu, Ed. IEEE, 2020, pp. 124–129. [Online]. Available: https://doi.org/10.1109/ICSTCC50638.2020.9259693

[61] R. Dechter and J. Pearl, "Generalized best-first search strategies and the optimality of a*," *J. ACM*, vol. 32, no. 3, pp. 505–536, 1985. [Online]. Available: https://doi.org/10.1145/3828.3830

[62] D. Ferguson and A. Stentz, "Field d*: An interpolation-based path planner and replanner," in *Robotics Research: Results of the 12th International Symposium, ISRR 2005, October 12-15, 2005, San Francisco, CA, USA*, ser. Springer Tracts in Advanced Robotics, S. Thrun, R. A. Brooks, and H. F. Durrant-Whyte, Eds., vol. 28. Springer, 2005, pp. 239–253. [Online]. Available: https://doi.org/10.1007/978-3-540-48113-3_22

[63] G. A. Mills-Tettey, A. Stentz, and M. B. Dias, "Dd* lite: Efficient incremental search with state dominance," in *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*. AAAI Press, 2006, pp. 1032–1038. [Online]. Available: http://www.aaai.org/Library/AAAI/2006/aaai06-162.php

[64] P. Yap, N. Burch, R. C. Holte, and J. Schaeffer, "Block a*: Database-driven search with applications in any-angle path-planning," in *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*, W. Burgard and D. Roth, Eds. AAAI Press, 2011. [Online]. Available: http://www.aaai.org/ocs/index.php/AAAI/AAAI11/paper/view/3449

[65] A. Martelli, "On the complexity of admissible search algorithms," *Artif. Intell.*, vol. 8, no. 1, pp. 1–13, 1977. [Online]. Available: https://doi.org/10.1016/0004-3702(77)90002-9

[66] K. Vora, "LUMOS: dependency-driven disk-based graph processing," in *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, D. Malkhi and D. Tsafrir, Eds. USENIX Association, 2019, pp. 429–442. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/vora

[67] A. Mislove, H. S. Koppula, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Growth of the flickr social network," in *Proceedings of the first Workshop on Online Social Networks, WOSN 2008, Seattle, WA, USA, August 17-22, 2008*, C. Faloutsos, T. Karagiannis, and P. Rodriguez, Eds. ACM, 2008, pp. 25–30. [Online]. Available: https://doi.org/10.1145/1397735.1397742

[68] A. E. Mislove and K. P. Gummadi, *Online Social Networks: Measurement, Analysis, and Applications to Distributed Information Systems*. Rice University.

[69] J. Kunegis, "KONECT: the koblenz network collection," in *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013, Companion Volume*, L. Carr, A. H. F. Laender, B. F. Lóscio, I. King, M. Fontoura, D. Vrandecic, L. Aroyo, J. P. M. de Oliveira, F. Lima, and E. Wilde, Eds. International World Wide Web Conferences Steering Committee / ACM, 2013, pp. 1343–1350. [Online]. Available: https://doi.org/10.1145/2487788.2488173

[70] R. Wetzker, C. Zimmermann, and C. Bauckhage, "Analyzing social bookmarking systems: A del.icio.us cookbook," in *Mining Social Data*, 2008.

[71] H. Kwak, C. Lee, H. Park, and S. B. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, M. Rappa, P. Jones, J. Freire, and S. Chakrabarti, Eds. ACM, 2010, pp. 591–600. [Online]. Available: https://doi.org/10.1145/1772690.1772751

[72] P. Boldi, M. Santini, and S. Vigna, "A large time-aware web graph," *SIGIR Forum*, vol. 42, no. 2, pp. 33–38, 2008. [Online]. Available: https://doi.org/10.1145/1480506.1480511

[73] D. Wen, Y. Huang, Y. Zhang, L. Qin, W. Zhang, and X. Lin, "Efficiently answering span-reachability queries in large temporal graphs," in *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 2020, pp. 1153–1164. [Online]. Available: https://doi.org/10.1109/ICDE48307.2020.00104

[74] Q. Ma and P. Steenkiste, "On path selection for traffic with bandwidth guarantees," in *Proceedings 1997 International Conference on Network Protocols*. IEEE, 1997, pp. 191–202.

[75] V. Nicosia, J. Tang, M. Musolesi, G. Russo, C. Mascolo, and V. Latora, "Components in time-varying graphs," *Chaos: An interdisciplinary journal of nonlinear science*, vol. 22, no. 2, p. 023101, 2012.

[76] R. A. Rossi, D. F. Gleich, A. H. Gebremedhin, M. M. A. Patwary, and M. Ali, "A fast parallel maximum clique algorithm for large sparse graphs and temporal strong components," *CoRR, abs/1302.6256*, 2013.

[77] D. Margan and P. R. Pietzuch, "Large-scale stream graph processing: Doctoral symposium," in *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS 2017, Barcelona, Spain, June 19-23, 2017*. ACM, 2017, pp. 378–381. [Online]. Available: https://doi.org/10.1145/3093742.3093907

[78] Y. Miao, W. Han, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, E. Chen, and W. Chen, "Immortalgraph: A system for storage and analysis of temporal graphs," *ACM Trans. Storage*, vol. 11, no. 3, pp. 14:1–14:34, 2015. [Online]. Available: https://doi.org/10.1145/2700302

[79] S. Gandhi and Y. Simmhan, "An interval-centric model for distributed computing over temporal graphs," in *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 2020, pp. 1129–1140. [Online]. Available: https://doi.org/10.1109/ICDE48307.2020.00102

[80] J. Byun, S. Woo, and D. Kim, "Chronograph: Enabling temporal graph traversals for efficient information diffusion analysis over time," in *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 2020, pp. 2026–2027. [Online]. Available: https://doi.org/10.1109/ICDE48307.2020.00232

[81] X. Zhu, M. Serafini, X. Ma, A. Aboulnaga, W. Chen, and G. Feng, "Livegraph: A transactional graph storage system with purely sequential adjacency list scans," *Proc. VLDB Endow.*, vol. 13, no. 7, pp. 1020–1034, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p1020-zhu.pdf

[82] R. Jin, N. Ruan, B. You, and H. Wang, "Hub-accelerator: Fast and exact shortest path computation in large social networks," *CoRR*, vol. abs/1305.0507, 2013. [Online]. Available: http://arxiv.org/abs/1305.0507

[83] A. Kusum, K. Vora, R. Gupta, and I. Neamtiu, "Efficient processing of large graphs via input reduction," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2016, Kyoto, Japan, May 31 - June 04, 2016*, H. Nakashima, K. Taura, and J. Lange, Eds. ACM, 2016, pp. 245–257. [Online]. Available: https://doi.org/10.1145/2907294.2907312