Tripoline: Generalized Incremental Graph Processing via Graph Triangle Inequality

Xiaolin Jiang*
University of California, Riverside
xjian049@ucr.edu

Chengshuo Xu*
University of California, Riverside
cxu009@ucr.edu

Xizhe Yin*
University of California, Riverside
xyin014@ucr.edu

Zhijia Zhao
University of California, Riverside
zhijia@cs.ucr.edu

Rajiv Gupta
University of California, Riverside
gupta@cs.ucr.edu

Abstract

For compute-intensive iterative queries over a streaming graph, it is critical to evaluate the queries continuously and incrementally for best efficiency. However, the existing incremental graph processing requires a priori knowledge of the query (e.g., the source vertex of a vertex-specific query); otherwise, it has to fall back to the expensive full evaluation that starts from scratch.

To alleviate this restriction, this work presents a principled solution to generalizing the incremental graph processing, such that queries, without their a priori knowledge, can also be evaluated incrementally. The solution centers around the concept of *graph triangle inequalities*, an idea inspired by the classical triangle inequality principle in the Euclidean space. Interestingly, similar principles can also be derived for many vertex-specific graph problems. These principles can help establish rigorous constraints between the evaluation of one graph query and the results of another, thus enabling reusing the latter to accelerate the former. Based on this finding, a novel streaming graph system, called *Tripoline*, is built which enables incremental evaluation of queries without their a priori knowledge. Built on top of a state-of-the-art shared-memory streaming graph engine (Aspen), Tripoline natively supports high-throughput low-cost graph updates. A systematic evaluation with a set of eight vertex-specific graph problems and four real-world large graphs confirms both the effectiveness of the proposed techniques and the efficiency of Tripoline.

*First three authors contributed equally to this research.

EuroSys '21, April 26–28, 2021, Online, United Kingdom

ACM ISBN 978-1-4503-8334-9/21/04.
https://doi.org/10.1145/3447786.3456226

ACM Reference Format:
Xiaolin Jiang, Chengshuo Xu, Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. 2021. Tripoline: Generalized Incremental Graph Processing via Graph Triangle Inequality. In *Sixteenth European Conference on Computer Systems (EuroSys '21), April 26–28, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3447786.3456226

1 Introduction

Graphs are widely used in many application domains thanks to their capability of modeling the complex relationships among entities. In many real-world application scenarios, a stream of updates are continuously applied to the graph, often in batches for better efficiency, known as the *streaming graph* scenario. Taking social networks [4] as an example, new data that carry rich connection information, such as tweets, are continuously generated, causing updates to the existing graph. Similar scenarios also occur in the mining of online shopping activities, where new purchases may generate new connections between customers (e.g., those who bought the same product) and between products (e.g., those that are bought together). In such common scenarios, new edges and vertices are continuously added to the graph.

In the streaming graph scenario, to reduce the latency of query evaluation, it is critical to evaluate the expensive iterative graph queries incrementally upon graph updates.

State of the Art. Several streaming graph systems have been proposed recently with support for incremental evaluation of iterative graph queries. Examples include Kineograph [3], Tornado [35], Naiad [26], KickStarter [43], Graphbolt [23], and so on. The basic idea of these systems is to reevaluate the query each time the graph gets updated, as illustrated in Figure 1. Instead of reevaluating the query from scratch (i.e., a full reevaluation), they start the reevaluation directly on the results of the previous evaluation, performing just enough calculations based on the newly inserted edges and vertices to get the results stabilized again. As the new edges and vertices in each update batch usually represent just a tiny fraction of the existing graph, the incremental evaluation usually converges much faster than a full reevaluation.

However, the above approach requires a priori knowledge of the query to be incrementally evaluated, referred to as

Corrected Version of Record. V.1.1. Published April 30, 2021.

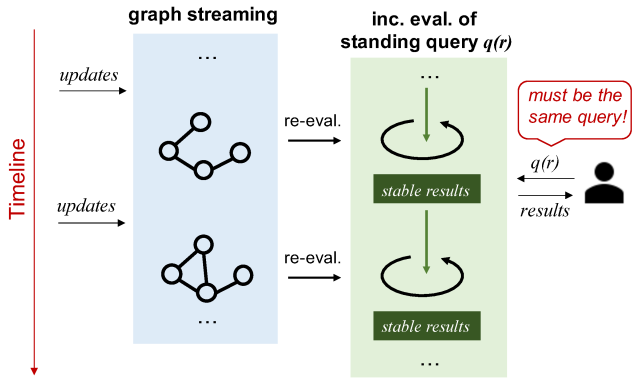


Figure 1. Streaming Graph Processing with Incremental Query Evaluation and the Limitation of Existing Solutions.

the *standing query*. This may not be an issue for queries without source vertex specification, such as PageRank; but creates a fundamental challenge for vertex-specific queries, like breath-first search (BFS), which target specific vertices of interests (e.g., $BFS(v_5)$). The source vertex of interest may be unknown until the query arrives. Thus, only the pre-selected standing queries (e.g., $BFS(v_5)$) can be incrementally evaluated; queries with other source vertices have to undergo an expensive full evaluation once they are received. This limitation significantly compromises the generality of the existing incremental streaming graph systems.

In this work, we propose a principled way to generalizing the incremental graph processing such that vertex-specific queries without their a priori knowledge may also benefit from incremental processing. The key to our solution is a concept called *graph triangle inequality*. Similar to the classic triangle inequality in the Euclidean space, triangle inequalities with generalized distance and comparison operators may also be derived for vertex-specific queries in the graph space. Based on them, we can establish rigorous constraints between a user query (whose source vertex can be any vertex in the graph) and the pre-selected standing query, thus enabling reusing the results of the latter to accelerate the evaluation of the former. We refer to this technique as *graph triangle inequality-based incremental processing*. For correctness, the graph query implementation is assumed to be monotonic and safe under asynchrony (more details are given in Section 4.3).

To demonstrate the effectiveness of the above generalized incremental graph processing, we developed a streaming graph system on top of a state-of-the-art streaming graph engine called Aspen [7], which offers a compact yet efficient data structure for high-throughput graph updates. We name the new system *Tripoline* to encapsulate its essence: use the graph triangle inequality as a "trampoline" to fast-forward the evaluation of queries different from the standing one. By continuously and incrementally evaluating a small set of pre-selected standing queries upon graph updates, Tripoline can incrementally evaluate previously unseen queries based on the results of the standing ones and the triangle inequalities.

We evaluated Tripoline using eight types of vertex-specific graph queries and four real-world large graphs (more details in Section 6). The results show that the performance benefits of Tripoline varies depending on the vertex-specific problems (and their graph triangle inequalities). Overall, we observed 8.83-30.52× speedups on four types of the evaluated graph queries, 1.18-1.89× speedups on three types of graph queries, and limited speedup (1.08×) on one type of graph queries.

In summary, this work makes the following contributions:

- It proposes to leverage *graph triangle inequality* in the scheme of incremental graph processing, which, to our knowledge, for the first time enables generalized incremental evaluation of vertex-specific queries;
- It introduces the triangle abstraction based on a pair of generalized distance and comparison operators, and establishes the specific graph triangle inequalities for a spectrum of vertex-specific graph queries.
- Finally, it develops *Tripoline*, a streaming graph system that supports *generalized* incremental evaluation for vertex-specific graph queries. The system has shown substantial performance improvements on multiple vertex-specific iterative graph queries.

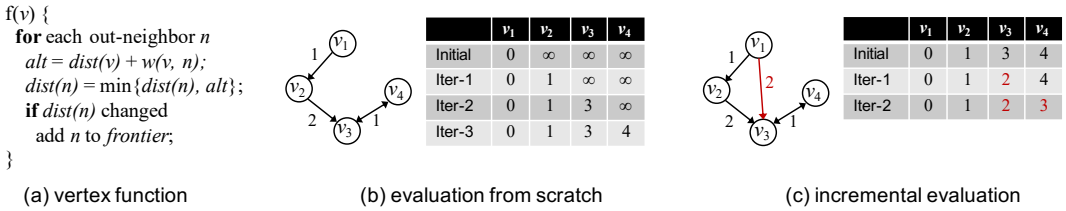
2 Background

This section introduces the basic graph programming model and the state of the art of incremental graph processing.

Vertex-Centric Programming. A commonly used model for programming graph applications is the *vertex-centric programming* model. It was first introduced by Pregel [22] based on the bulk synchronous parallel (BSP) model [41]. The model requires defining a *vertex function* that computes some properties of the vertices (a.k.a. vertex values). The graph computations start from some default initial vertex values, then apply the vertex function across all (or a subset of) vertices of the graph, iteration by iteration until the vertex values become stable (or some threshold is reached).

Figure 2-(a) illustrates a vertex-centric implementation of the single-source shortest path (SSSP) query, which finds the shortest distances from a source vertex to all other vertices in the graph. The vertex function $f(v)$ computes an alternative distance based on the current value of the vertex, then compares it with the value of each of its outgoing neighbors. If the new value is less than the existing one, the neighbor's value will be updated. This is known as as the *push* model [1]. As shown in Figure 2-(b), initially, all the vertex values are set to ∞, except the source vertex whose value is set to zero. Then the vertex function $f(v)$ is evaluated across all the vertices over iterations until all the vertex values stop changing.

[1]The vertex function can also be implemented using a *pull* model which updates the value of the vertex based on its in-neighbors' values.



	v_1	v_2	v_3	v_4
Initial	0	∞	∞	∞
Iter-1	0	1	∞	∞
Iter-2	0	1	3	∞
Iter-3	0	1	3	4

	v_1	v_2	v_3	v_4
Initial	0	1	3	4
Iter-1	0	1	2	4
Iter-2	0	1	2	3

Figure 2. Vertex-Centric Programming and Incremental Query Evaluation using SSSP(v_1) as an Example.

To improve the efficiency, an active vertex list (a.k.a. *frontier*) can be maintained which only consists of vertices whose values were changed in the last iteration, so only vertices in the frontier need to be evaluated in each iteration. In the case of SSSP, the frontier is initialized with only the source vertex and will become empty once all the vertex values are converged. Hereinafter, this work assumes a frontier-based implementation of the push model.

Incremental Graph Processing. As mentioned earlier, in the common streaming graph processing scenario [3, 23, 26, 35, 43], the graph is continuously updated with new edges and vertices, usually in batches for better efficiency. A recent work [43] also discussed the scenario with edge deletions, which is orthogonal to the focus of this work. Like many prior works [3, 35], we assume the growing graph scenarios in this work. After each batch of insertions, the standing graph query needs to be reevaluated to reflect the latest results. Instead of reevaluating the query on the updated graph from scratch (i.e., view it as a completely new graph), existing streaming graph systems adopt an incremental graph query evaluation strategy to improve the efficiency.

The design of the incremental query evaluation naturally matches the BSP model in the aforementioned vertex-centric programming. Consider the example in Figure 2-(c). After a new edge (v_1, v_3) is inserted, the reevaluation directly starts from the converged vertex values of the prior evaluation, rather than initializing the vertex values with ∞. To ensure the correctness, the source vertices of the newly inserted edges (i.e., v_1) need to be inserted into the frontier, which resumes the iterations until a new stabilization is reached. As each insertion batch is typically a tiny fraction of the existing graph, the reevaluation tends to terminate much faster than a full reevaluation [3, 43].

Despite the promise of incremental graph processing, there exists a fundamental limitation in the existing design – it assumes a priori knowledge of the query. The assumption holds for queries that do not depend on a specific vertex, such as PageRank, but imposes a major obstacle for vertex-specific queries, like BFS and SSSP. For the latter, the incremental evaluation would work only for the pre-selected standing query, like SSSP(v_1); for queries originating at other vertices in the graph, an expensive full evaluation is required.

In fact, vertex-specific graph queries appear more common than "whole-graph queries" in real-world applications. First, vertex-specific queries are concerned with the interests or capture the perspective of a specific vertex, which are common in online shopping and social networks, such as generating recommendations for individual customer [47] and finding the overlap of friends of two specific users [4]. Second, as subproblems, vertex-specific graph queries often require less time and space than their counterpart whole-graph queries (e.g., SSSP vs all-pair shortest path). This is especially critical in the streaming graph scenario, where the query evaluation needs to keep up with the graph updates.

In summary, incremental graph processing is essential to the streaming graph systems. However, its existing design suffers from a fundamental applicability challenge for an important group of graph queries – vertex-specific queries. Before presenting our solution, we first introduce the key principle behind it – graph triangle inequalities.

3 Graph Triangle Inequality

In this section, we first provide an intuition of *graph triangle inequality*, then formally define the principle and present several graph triangle inequalities examples.

3.1 Intuition

Triangle inequality [17], as illustrated in Figure 3, is a basic principle in Euclidean geometry. It states the fact that, for any given triangle Δxyz, the sum of the lengths of any two sides must be greater than or equal to the length of the third side. Prior research [11] has shown the possibility to leverage triangle inequality to accelerate K-means clustering in the Euclidean space. Inspired by this, we wondered if similar principles exist in graph problems, and hence maybe used to optimize streaming graph processing. In fact, for a spectrum of vertex-specific graph problems, similar inequalities can be naturally derived. Next, we first use SSSP as an example to introduce the graph triangle inequality, because it calculates distances, which are similar the lengths in the classical triangle inequality, except that the "domain" is a graph, rather than the Euclidean space.

SSSP Triangle. It is not hard to find that the vertices in a graph are analogous to the points in the Euclidean space. The *distance* between two points in the Euclidean space is the length of line segment connecting them. Similarly, the *distance* between two vertices v_1 and v_2 in a weighted graph

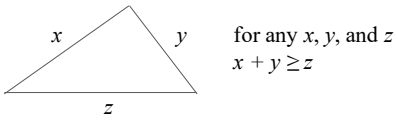


Figure 3. Triangle Inequality in Euclidean Geometry.

is the minimum weight of all paths connecting them:

$$dist(v_1, v_2) = min\{w(p) \mid p \text{ is a path from } v_1 \text{ to } v_2\} \quad (1)$$

where $w(p)$ is the sum of weights on all the edges in path p. Note that, for undirected graphs, as paths are symmetric, we have $dist(v_1, v_2) = dist(v_2, v_1)$. Based on this analogy, it is not hard to find that a triangle inequality also holds for graph distances, as illustrated in Figure 4.

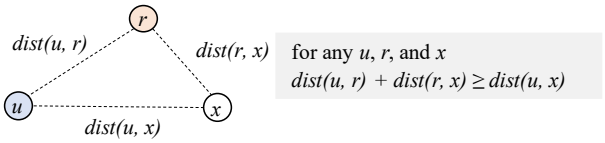


Figure 4. Triangle Inequality in SSSP (dashed lines represent the shortest paths between two vertices).

In fact, the above triangle inequality in Figure 4 becomes obvious once one realizes that the shortest path from u to r can be concatenated with the shortest path from r to x, and the resulted path is just one of the many paths from u to x, hence must be no shorter than the shortest path from u to x.

Actually, the above graph triangle inequality based on the distances between vertices is well-known in the theoretical graph community [1]. Some prior work [5] has exploited this principle to approximate distances in web-scale large graphs, which shares some of the spirit of this work. However, as we will demonstrate shortly, our work discusses a much broader definition of "distance" that goes beyond the conventional one shown in Equation 1. Furthermore, our work exploits the graph triangle inequality in a different context – streaming graph processing, where the accuracy of each query result is always guaranteed – no approximation is allowed.

For brevity, we refer to the above distance-based triangle inequality as *SSSP triangle*. Next, we generalize it by defining a more general definition of "distance" and two abstract operators for addition and comparison, respectively.

3.2 Triangle Abstraction

Rather than referring to the distance, we define the graph triangle inequality for a property between two vertices – $property(v_1, v_2)$, where (v_1, v_2) is an ordered pair for directed graphs and an unordered pair for undirected graphs.

Definition 3.1. Given the property definition between two vertices $property(v_1, v_2)$, the *graph triangle inequality* can be formally defined by the following equation:

$$property(v_1, v_2) \oplus property(v_2, v_3) \succeq property(v_1, v_3) \quad (2)$$

where $\oplus$ depicts an abstract addition and $\succeq$ represents an abstract greater than or equal operator.

To demonstrate the generality of the triangle abstraction, we next present several concrete graph triangle inequalities that are not based on the distance property.

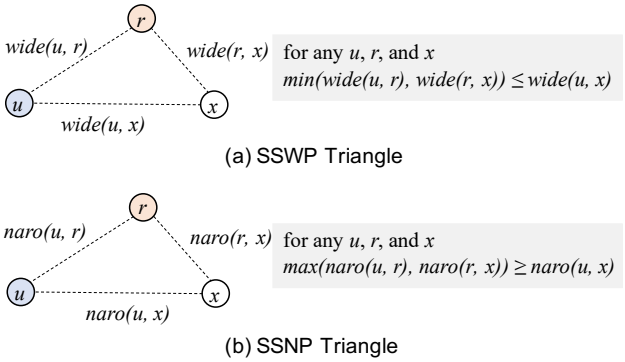


Figure 5. Triangle Inequalities in SSWP and SSNP (dashed lines depict the widest/narrowest paths between vertices).

SSWP/SSNP Triangle. SSWP and SSNP are abbreviations for single-source widest path and single-source narrowest path, respectively. Both of them play important roles in network routing [46] and transportation planning [6].

Given a source vertex v, SSWP and SSNP compute the widest and narrowest path from v to every other vertex in the graph, respectively. The widest path between two vertices is the path whose minimum edge weight is the largest, while the narrowest path between two vertices is the path whose maximum edge weight is the smallest, as defined below:

$$wide(v_1, v_2) = max\{minw(p) \mid \text{ path } p \text{ from } v_1 \text{ to } v_2\} \quad (3)$$

$$naro(v_1, v_2) = min\{maxw(p) \mid \text{ path } p \text{ from } v_1 \text{ to } v_2\} \quad (4)$$

where $minw(p)$ and $maxw(p)$ represent the minimum and maximum edge weight along path p, respectively.

Based on their definitions, it is not difficult to derive the triangle inequalities for SSWP and SSNP, shown in Figure 5. The reasonings behind these inequalities are similar to that of SSSP, except that they are based on different addition $\oplus$ and comparison $\succeq$ operators. For example, the inequality holds for SSWP because the widest paths from u to r and from r to x can be concatenated, and the width of the concatenated path must be no larger than the width of the widest path from u to x as it is just one of the paths from u to x. Similarly, we refer to the triangle inequalities for SSWP and SSNP as *SSWP triangle* and *SSNP triangle*, respectively, for brevity.

Other Triangles. Due to space limitations, we next briefly present the triangle inequalities for the other graph problems that we have considered. These include:

- Single-source reachability (SSR) [18] which finds all the vertices connected to the source vertex. Figure 6-(a)

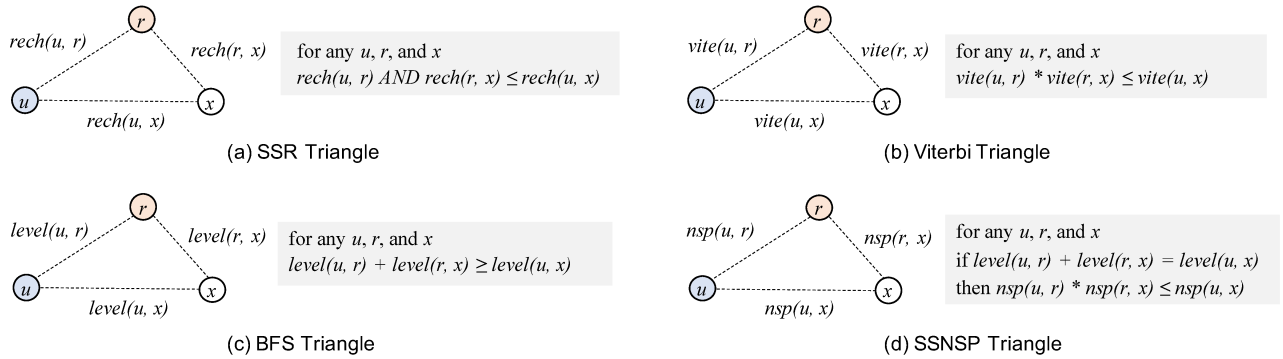


Figure 6. Triangle Inequalities in SSR, Viterbi, BFS/Radii, and SSNSP (where a dashed line depicts the connectivity, maximum probability path, BFS-level, and the shortest paths between two vertices, respectively).

shows its triangle inequality based on the reachability property defined in Equation 5.

- Viterbi algorithm (Viterbi) [20] which computes the probability along the Viterbi path (a state path that maximizes the conditional probability) from the source vertex. Figure 6-(b) shows its triangle inequality based on the *vite* property defined in Equation 6, where $w(p)$ depicts the total weights of edges in path p.
- Breath-first search (BFS) [25] which computes the level of each vertex in the BFS tree rooted at the source vertex. Figure 6-(c) shows its triangle inequality based on the BFS level property defined in Equation 7, where $nEdges(p)$ depicts the number edges in path p.
- Radii estimation (Radii) [37] which estimates the graph radius by running multiple SSSP and selecting the largest distance among their results. As it is based on SSSP, its triangle inequality is just that of SSSP.
- Single-source number of shortest path (SSNSP) [33] which computes not only the BFS levels [2], but also the number of shortest paths from the source vertex to all the other vertices. Figure 6-(d) shows its triangle inequality based on both the BFS level property and the number of shortest paths property. The latter is defined in Equation 8, where $|\cdot|$ depicts the set size.

$$rech(v_1, v_2) = \begin{cases} 1 & \text{if a path from } v_1 \text{ to } v_2 \text{ exists} \\ 0 & \text{otherwise} \end{cases} \quad (5)$$

$$vite(v_1, v_2) = max\{1/w(p) \mid \text{path } p \text{ from } v_1 \text{ to } v_2\} \quad (6)$$

$$level(v_1, v_2) = min\{nEdges(p) \mid \text{path } p \text{ from } v_1 \text{ to } v_2\} \quad (7)$$

$$nsp(v_1, v_2) = |\{\text{the shortest paths from } v_1 \text{ to } v_2\}| \quad (8)$$

For brevity, we refer to the above triangle inequalities as *SSR triangle*, *Viterbi triangle*, *BFS triangle*, and *SSNSP triangle*, respectively. Among these triangles, the Viterbi triangle and BFS triangle can be intuitively derived just based on their definitions and the fact that the paths from u to r and from r to x can be concatenated to form one path from u to x. For SSR triangle, the situation is different in that the property of interest (i.e., reachability) is about the existence of any path between two vertices. In this case, a logical *AND* perfectly fits in the role of the $\oplus$ operator. The last one, SSNSP triangle is also special in that it requires a predicate (condition) for the triangle inequality to hold. As we will show later, the predicate actually affects the effectiveness of the triangle inequality in the use of incremental query evaluation.

In summary, the graph triangle inequality, as abstracted in Equation 2, is generally enough to capture a spectrum of vertex-specific graph problems.

4 Generalized Incremental Evaluation

In this section, we show that, based on the graph triangle inequality abstraction, incremental evaluation of queries without a priori knowledge can be achieved in general.

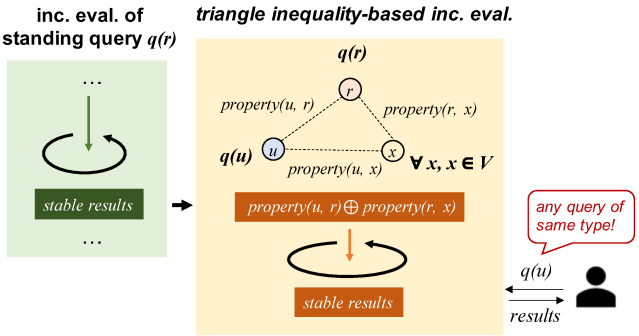


Figure 7. Triangle Inequality (Δ)-based Incremental Query Evaluation for an Arbitrary Query of the Same Type (i.e., a query starting from a different source vertex but asking for the same graph property, like SSWP(v_1) and SSWP(v_2)).

[2] In this case, SSNSP is for unweighted graphs.

4.1 Δ-based Incremental Evaluation

The key to our solution is a principled way of "connecting" the evaluation of a vertex-specific query to the results of another query evaluation of the same type (e.g., SSWP) based on their graph triangle inequality.

Execution Model. Figure 7 illustrates the basic idea of our solution. Assume $q(r)$ is the pre-selected standing query (the selection will be discussed later), where r is the source vertex. In the programming system, query $q()$ is a user-specified function that implements the (vertex-specific) querying logic while r is the parameter to the function.

First, the standing query $q(r)$ is evaluated continuously and incrementally upon graph updates, like those in the existing incremental graph processing systems [3, 26, 43]. Meanwhile, the system accepts user queries like $q(u)$ which is of the same type as $q(r)$, but its source vertex u could be any vertex in the graph. From the evaluation of $q(r)$, we can obtain the values of $property(r, u)$ and $property(r, x)$. For easier explanation, here we assume the graph is undirected (directed ones will be discussed later), which means that we can also obtain $property(u, r)$ – the same as $property(r, u)$. In addition to vertices r and u, consider an arbitrary vertex x in the graph different from r and u. Together, r, u, and x form a triangle, just like one of those in Section 3. Then, based on the addition operator $\oplus$ in the triangle abstraction (see Equation 2), we can compute the following value set:

$$\Delta(u, r) = \{property(u, r) \oplus property(r, x) \mid x \in V\} \quad (9)$$

Next, instead of evaluating $q(u)$ from scratch (i.e., using the default initial values) on the current version of the graph, the system starts its evaluation directly from $\Delta(u, r)$, and runs until all the vertex values are converged. Note that, just like full evaluation, the above incremental evaluation also starts from the source vertex u (i.e., frontier is initialized with u).

In the above process, the system maintains an in-memory state consisting of three parts: (i) the streaming graph, (ii) the evaluation of standing query, and (iii) the evaluation of user query. As detailed later, in our prototype, the streaming graph can be incrementally maintained with a compression tree-based data structure (Aspen [7]), while the results of query evaluation are kept in a *property array* of size $|V|$.

We refer to the above streaming graph execution model as *triangle inequality-based incremental evaluation*, or *Δ-based incremental evaluation* [3] for short.

In the following sections, we will first extend the proposed Δ-based incremental evaluation to directed graphs, then analyze its correctness, benefits and costs, and finally discuss how the standing query can be selected.

4.2 Dual-Model Evaluation for Directed Graphs

In the case of directed graphs, $property(u, r)$ may not be the same as $property(r, u)$, thus not available in the evaluation

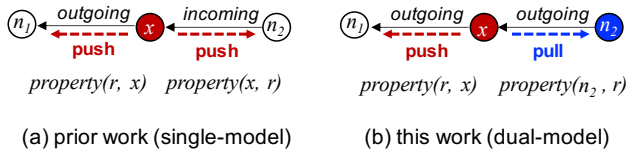


Figure 8. Computing $property(r, x)$ and $property(x, r)$ on Directed Graphs: Prior Work [3, 5] vs. Our Solution.

results of the standing query $q(r)$. In this case, we turn to the reversed graph problem, denoted as $q^{-1}(r)$, which computes the properties from all vertices to r.

$$\text{Results of } q^{-1}(r) = \{property(x, r) \mid x \in V\} \quad (10)$$

Taking SSSP as an example, $\text{SSSP}^{-1}(v)$ is to find the shortest distance from every vertex in the graph to v.

A straightforward way to evaluate $q^{-1}(r)$, as elaborated in prior work [3, 5], is to update values of the in-neighbors, rather than the out-neighbors as in the evaluation of $q(r)$. Figure 8-(a) illustrates this idea. However, with this solution, the evaluations of queries $q(r)$ and $q^{-1}(r)$ need to access both outgoing and incoming edges efficiently (i.e. indices for both outgoing and incoming neighbors). This not only doubles the memory consumption of the edge data (two-way indices rather than one-way), but also increases the cost of streaming graph maintenance – need to keep both incoming and outgoing edge representations to date.

To address the above issue, we propose a novel *dual-model query evaluation* solution for directed graphs. The solution enables us to evaluate both queries $q(r)$ and $q^{-1}(r)$ on a graph with only one-way edge representation (outgoing or incoming edge-based). The key to this solution is the fact that both push/pull model and incoming/outgoing edges are relative. From the global view, a push model along the incoming edges from the perspective of x is equivalent to a pull model along the outgoing edges from the perspective of one of x's neighbors, say n_2 [4], as illustrated in Figure 8-(b). By adopting both models for the two queries respectively, a one-way edge representation is sufficient for calculating both $property(r, x)$ and $property(x, r)$ for any x in V.

We have presented the Δ-based incremental evaluation on both indirected and directed graphs. Next, we discuss its applicability and correctness.

4.3 Applicability and Correctness

First, Δ-based incremental evaluation targets vertex-specific queries. For non-vertex-specific queries, such as PageRank and connected components (CC), because they are already well-suited to the existing incremental graph computation models [3, 43], they can be incrementally evaluated without triangle inequalities (also supported by Tripoline).

Second, to apply Δ-based incremental evaluation to a type of vertex-specific query, a graph triangle inequality needs to

[3] Here, Δ reads as triangle inequality, not delta (difference).

[4] Or vice versa if pull model is assumed to be the default model.

be established. Note that the triangle inequality is derived based on the property of interest rather than the specific implementation of its queries. Given $property(u, *)$, where $*$ refers to any vertex in the graph, a triangle inequality among $property(u, r)$, $property(r, x)$, and $property(u, x)$ often can be intuitively derived based on the fact that a path from u to r, then to x is just one of the possible paths from u to x. Following this intuition, in Section 3, we have demonstrated the possibility of establishing triangle inequalities for several commonly seen graph problems.

Though triangle inequality is independent of the query implementation, some properties of the vertex function $f(v)$ are still closely relevant to the correctness of the triangle inequality-based incremental evaluation. Next, we mainly discuss two such properties: *monotonicity* and *safety under asynchrony*, which are formally defined as below.

Definition 4.1. In vertex-centric programming framework, vertex function $f(v)$ is *monotonic* if all vertex values only change monotonically across iterations.

Definition 4.2. In vertex-centric programming framework, vertex function $f(v)$ is *safe under asynchrony*, or *async-safe* for short, if the vertex values still converge correctly even when $f(v)$ is executed asynchronously based on the new values of its neighbors calculated in the current iteration.

Note that, for vertex-centric graph algorithms, the above two properties are not rare. In fact, they are in the abstraction of many existing graph programming frameworks [13, 21, 33, 43]. For example, GraphLab [21] asynchronously executes graph algorithms for better efficiency, GRAPE [13] relies on the monotonicity of iterative graph algorithms for automatic parallelization, Subway [33] leverages the asynchrony and monotonicity to reduce the data transfer in out-of-memory graph processing, and most relevantly, KickStarter [43] requires monotonicity to support edge deletions in streaming graph processing. In the following discussion, we assume that the vertex function $f(v)$ is monotonic and async-safe.

In the following, for conciseness, we use $t_{init}(x)$ to denote the initial value of vertex x under the Δ-based incremental evaluation of query $q(u)$ (i.e., $t_{init}(x) = property(u, r) \oplus property(r, x)$), and $t_{conv}(x)$ to denote the correct converged value of x (i.e., $t_{conv}(x) = property(u, x)$).

Lemma 4.3. *In Δ-based incremental evaluation, if vertex x's initial value $t_{init}(x) \succ t_{conv}(x)$, then at least one of its in-neighbors, say vertex z, must be initialized with value $t_{init}(z)$, such that $t_{init}(z) \succ t_{conv}(z)$, and z is on the path from source vertex u to x that yields $t_{conv}(x)$.*

Proof. By contradiction, assume that all in-neighbors of x, denoted as z_i, $1 \le i \le k$ (where k is the number of neighbors of x), are initialized with their correct converged values $t_{conv}(z_i)$, $1 \le i \le k$, then the vertex r in the standing query $q(r)$ is on the paths from u to z_i that yield $t_{conv}(z_i)$. Also, one in-neighbor of x must be on the path from source vertex u to x that yields $t_{conv}(x)$. Together, we have that r is on one path from u to x that yields $t_{conv}(x)$. Thus, $t_{init}(x) = t_{conv}(x)$, which contradicts the assumption in the lemma. □

Based on Lemma 4.3, we have the following conclusion.

Theorem 4.4. *Given a vertex function $f(\cdot)$ that is monotonic and async-safe, if triangle inequality holds on the property that $f(\cdot)$ computes, the Δ-based incremental evaluation yields the same results as the non-incremental evaluation.*

Proof. Consider an arbitrary vertex x, which is initialized with $t_{init}(x)$ by Δ-based incremental evaluation. First, based on triangle inequality, $t_{init}(x) \succeq t_{conv}(x)$, where $t_{conv}(x)$ is the correct converged value of vertex x. If $t_{init}(x) = t_{conv}(x)$, then based on monotonicity, the evaluation will not change its value, so it will remain correct in the end. Otherwise, if $t_{init}(x) \succ t_{conv}(x)$, by applying Lemma 4.3 on vertex x, we know there exists one in-neighbor of x, say z, which is on the path that yields $t_{conv}(x)$ and its initial value $t_{init}(z) \succ t_{conv}(z)$. Similarly, we reapply Lemma 4.3 on vertex z. By repeating these, we can find a reversed path starting from vertex x, along which all the vertices have initial values that are greater than their correct converged values, and they are on the path from u to x that yields $t_{conv}(x)$. If the reversed path can reach the source vertex u, an activation of u will gradually stabilize all the vertex values along the path with their correct converged values, including x's value. Here, monotonicity ensures that the initial values of these vertices will be updated with their corresponding correct converged values (as $t_{init}(x) \succ t_{conv}(x)$), while async-safety ensures that these updates will not alter the converged values even when they are performed asynchronously. On the other hand, if the reversed path cannot reach the source vertex, then it would stay unchanged (the default initial value). □

Besides the theoretical correctness discussion of Δ-based incremental evaluation, our experimental evaluation also confirmed the correctness of results under many different testing cases (Section 6). Next, we discuss the benefits and costs of Δ-based incremental evaluation.

4.4 Cost-Benefit Analysis

To examine the benefits of Δ-based incremental evaluation, we discuss how the two basic cases in its initialization: $t_{init}(x) = t_{conv}(x)$ and $t_{init}(x) \succ t_{conv}(x)$, affect the computations.

Figure 9-(a) illustrates the first case $t_{init}(x) = t_{conv}(x)$, where the vertex value will not be changed during iterations according to monotonicity, thus the vertex will never activate its out-neighbors (bottom two vertices) [5]. This means that all value propagations reaching x stop. In this way, it reduces the amount of computations. In the second case, as shown in Figure 9-(b), the initial value of x is not stable, but better than the default initial value (i.e., $t(x) \prec init$). In this case,

[5]They may still be activated due to changes of their other in-neighbors.

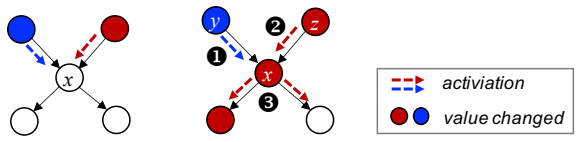


Figure 9. Benefits of Δ-based Incremental Evaluation.

it may "block" some value propagations (e.g., the blue one which yields a worse value than $t_{init}(x)$), but allow others (e.g., the red one which yields a better value than $t_{init}(x)$).

In both cases, the benefits come from the reduction of the vertex function evaluations. Thus, the benefits can be roughly captured by the *activation ratio*, denoted as R_{act}:

$$R_{act} = \frac{N_{act} \text{ with } \Delta\text{-based inc. eval.}}{N_{act} \text{ without } \Delta\text{-based inc. eval.}} \tag{11}$$

where N_{act} denotes the total number of times that the vertex function is evaluated. Note that $R_{act} \leq 1$ because the new initial values $\Delta(u, r)$ are no worse than the default ones. In general, the closer $\Delta(u, r)$ are to the stable values, the lower the R_{act} is. In addition, as a side-effect, the reduction of vertex activations often leads to fewer number of iterations – faster convergence.

On the other hand, Δ-based incremental evaluation also brings overhead. A direct cost comes from calculating the initial values – $\{property(u, r) \oplus property(r, x) \mid x \in V\}$. As $property(u, r)$ is fixed for the given standing query $q(r)$ and user query $q(u)$, the calculation simply traverses results of $q(r)$ to read $property(r, x)$, $x \in V$, from an array. Due to spatial locality, this overhead is often negligible (e.g., about 0.3% for SSSP). Besides that, there are indirect costs regarding the incremental evaluation of standing queries. For example, for direct graphs, we incrementally evaluate not only $q(r)$, but also $q^{-1}(r)$. Depending on applications, the evaluation of $q^{-1}(r)$ may be counted as an overhead if its results are not needed. As we will discuss shortly, we may also want to incrementally evaluate multiple standing queries, whose costs may also be counted as the overheads depending on the application (more details in Section 4.5).

In summary, the performance improvements of Δ-based incremental evaluation mainly depend on the activation ratio R_{act}. Even with low R_{act}, the incremental evaluation only introduces limited direct overhead. Next, we discuss a key factor for R_{act} – the selection of standing query.

4.5 Standing Query Selection and Cost Management

The effectiveness of Δ-based incremental evaluation, roughly measured by R_{act}, depends on which query is selected as the standing query $q(r)$. A better selection may yield a lower R_{act}, thus a higher speedup. Moreover, is it worthwhile to select multiple standing queries? We address these questions next. First, we present two basic selection strategies.

Triangle-based Selection. As discussed in Section 4.4, the effectiveness of Δ-based incremental evaluation depends on how close the initial values $\Delta(u, r)$ are to the stable values (in terms of $\prec$), hence it is the better to select the standing query that yields lower values of $\Delta(u, r)$. Based on this, given user query $q(u)$, we select $q(r^*)$, such that,

$$r^* = \arg\min_{r \in V} \sum_{\forall x \in V} property(u, r) \oplus property(r, x) \tag{12}$$

$$= \arg\min_{r \in V} property(u, r) \cdot |V| \oplus \sum_{\forall x \in V} property(r, x) \tag{13}$$

However, in practice, we do not know the user query $q(u)$ – u can be any vertex in V. To find the best $q(r)$ overall, we need to compute the summation in Equation 12 for every u in V and select the one that minimizes the summation of those summations. Essentially, this requires collecting the $property(v_i, v_j)$ between every pair of vertices in the graph. Apparently, this is impractical for large graphs [6] even in non-streaming scenarios due to the high time and space complexities, not to mention the streaming scenarios where the property values change as the graph is updated.

Topology-based Selection. From the perspective of graph topology, it may be attempted to select the standing query $q(r)$ whose source vertex r is closer to the vertex u in the user query $q(u)$ in terms of the number of hops, because in this way, u and r share more paths or path segments to other vertices. Interestingly, we find that this heuristic only works for some graph problems, such as SSSP and BFS, but not the others, like SSWP and Viterbi. The reason is that the heuristic may contradict the triangle inequalities. Take SSWP as an example, in fact, the more hops that u and r are away from each other, the larger value $wide(u, r)$ might be, thus the better value $wide(u, r) \oplus wide(r, x)$ may possess.

Instead, for graph topology, we focus on the reachability of r in the standing query $q(r)$ to the other vertices in the graph. In fact, to effectively leverage the triangle inequality, there should be at least one path from r to vertex u in the user query $q(u)$, and to every other vertex x, $x \in V$; otherwise, $property(u, r) \oplus property(r, x)$ would be as "bad" as the default initial value (e.g., ∞ in SSSP). One simple yet reliable way to approximate the reachability is to select a query with a high-degree source vertex [7], which is more likely to reach a larger amount of vertices. Thus, we have the following heuristic for selecting standing query $q(r)$.

$$r^* = \arg\max_{r \in V} degree(r) \tag{14}$$

As shown next, in practice, we adopt a solution combining the *triangle-based* and *topology-based* selections to achieve a balance between complexity and effectiveness. The key to exploiting this tradeoff is adopting multiple standing queries.

[6] For small graphs that are affordable for collecting these properties, the results can be directly cached – no need for incremental evaluation.
[7] Following the push model, here it refers to the out-degrees.

Selecting Multiple Standing Queries. First, we pre-select a set of K standing queries offline using the topology-base selection, that is, queries with the top-K high-degree vertices:

$$Standing_K = \{q(r_1), q(r_2), \cdots, q(r_K)\}$$

Then, at runtime, we pick the best one among the K standing queries based on the specific user query $q(u)$, according to a simplified version of Equation 13:

$$r^* = \underset{r \in Standing_K}{\arg\min}\ property(u, r) \tag{15}$$

Equation 15 is based on our experimental finding that, for the standing queries with top-K high-degree vertices, there is limited variation for the summation in Equation 13.

In this way, the standing query selection not only becomes query-specific, but also incurs negligible runtime overhead.

Managing the Costs. However, incrementally evaluating multiple standing queries may take longer – each time the graph is updated, it has to ensure that the evaluation of every standing query reaches stabilization. Here, we present two ways to alleviate these costs.

First, we evaluate the K standing queries in *batch* mode. That is, we maintain a combined frontier for all the active vertices among the K queries, and for each active vertex v, we apply the vertex function for the K standing queries together (those are inactive on v are masked). In this way, both the graph and vertex value arrays of standing queries can be accessed in a coalesced manner, thus incurring much less cost comparing to evaluating each standing query separately.

Second, we can adjust K to exploit the tradeoff between the maintaining cost of standing queries and the effectiveness of Δ-based incremental evaluation. When the user queries are made relatively more frequently than the graph updates (in batches), we may afford a larger K, as the overhead can be amortized by more user queries. In the opposite scenarios, we may reduce K such that the (incremental) standing query evaluation can finish quickly, and the following user query evaluation can start earlier.

So far, we have discussed the major aspects of the proposed Δ-based incremental evaluation. Next, we present a new streaming graph processing system that supports Δ-based incremental evaluation for vertex-specific queries.

5 Implementation of Tripoline

Based on the proposed generalized incremental evaluation, we developed *Tripoline*, a shared-memory streaming graph processing system. To our best knowledge, Tripoline is the first system of this kind that supports incremental processing of vertex-specific queries requiring no a priori knowledge of source vertices. Figure 10 illustrates its high-level structure, which consists of four major components:

- A *streaming graph engine* that accepts graph updates and maintains the data structures of the current graph. As the focus of this work is not to build such an engine, Tripoline adopts a state-of-the-art streaming graph engine called *Aspen* [7]. Internally, Aspen leverages a compressed tree-based graph representation to achieve both high-space efficiency and high-throughput graph updates, and uses a work-stealing scheduler similarly to Cilk for parallelism. Also, we extended the current version of Aspen to support edge weights.
- A *standing query evaluation module* that continuously and incrementally evaluates a set of standing queries upon graph updates; For better efficiency, we implemented the *batch* mode mentioned in Section 4.5.
- A *user query evaluation module* that employs Δ-based incremental evaluation to fulfill the user requests.
- Finally, a *programming interface* that not only provides the conventional vertex-centric programming, but also offers a triangle abstraction for specifying the triangle inequality of the specific graph problem. Basically, the developers need to overwrite the generic addition and comparison operators $\oplus$ and $\succeq$, respectively.

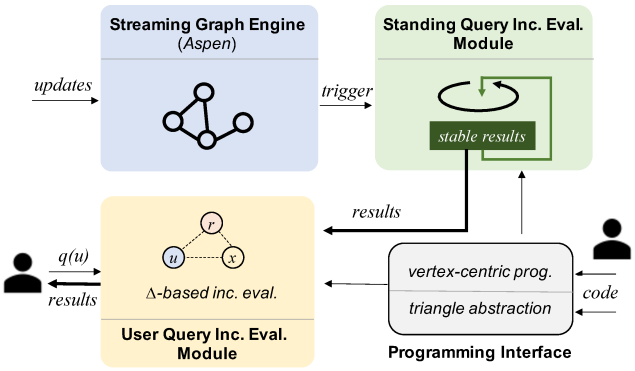


Figure 10. System Architecture of Tripoline.

Configuration and Parameters. The above three runtime modules (colored boxes in Figure 10) are configured to be executed exclusively (i.e., in serial), though each of them runs in parallel individually. This configuration maximizes the resource availability for each task: graph updates, standing query evaluation, and user query evaluation, respectively.

In our current setup, K standing queries are first selected based on their reachability to all possible source vertices in the user queries, for which we choose the top-K high-degree vertices as an approximation (i.e., the "topology-based selection" in Section 4.5). With that, the only parameter to be tuned is K, which depends on the query type and the memory capacity of the machine (a larger K means results of more queries need to be kept in memory). When the system is set up initially, K can be tuned and selected using a few sample values (as shown later in the evaluation – Table 5). To ease its deployment, a basic auto-tuner can be added to make the K selection transparent to the users. Furthermore, the standing query selection might be further improved based on the distribution of user queries when it is available.

Table 1. Benchmarks in Tripoline

Bench.	Pseudo-code of Vertex function
BFS	**for** each out-neighbor n of s level(n) = min { level(n), level(s) +1 }; **if** level(n) changed **then** add n to $frontier$;
SSSP	**for** each out-neighbor n of s dist(n) = min { dist(n), dist(s) + w(s, n) }; **if** dist(n) changed **then** add n to $frontier$;
SSWP	**for** each out-neighbor n of s wide(n) = max { wide(n), min { wide(s), w(s, n) } }; **if** wide(n) changed **then** add n to $frontier$;
SSNP	**for** each out-neighbor n of s naro(n) = min { naro(n), max { naro(s), w(s, n) } }; **if** naro(n) changed **then** add n to $frontier$;
Viterbi	**for** each out-neighbor n of s vite(n) = max { vite(n), vite(s) / w(s, n) }; **if** vite(n) changed **then** add n to $frontier$;
SSR	**for** each out-neighbor n of s rech(n) = true; **if** rech(n) changed **then** add n to $frontier$;
Radii	**for** each out-neighbor n of s dist1(n) = min { dist1(n), dist1(s) + w(s, n) }; · · · dist16(n) = min { dist16(n), dist16(s) + w(s, n) }; **if** any dist∗(n) changed **then** add n to $frontier$;

SSNSP	**for** each out-neighbor n of s
	if level(n) == level(s) + 1 **then**
	delta(n) += delta(s); add n to $frontier$;
	ssnsp(s) += delta(s);

Note that non-vertex-specific queries (e.g., PageRank and CC) can also be implemented on Tripoline, in which case, the system simply maintains them incrementally as the standing queries, like the existing incremental query evaluation [3, 26]. Similarly, vertex-specific queries with a priori knowledge can be treated as the standing queries, so that they can be maintained incrementally and answered directly.

In addition, Tripoline includes a set of built-in benchmarks for which the vertex functions are designed to satisfy the desired properties for correctness (see Section 4.3). Table 1 summarizes their vertex functions.

6 Evaluation

This section evaluates Tripoline and the effectiveness of Δ-based incremental graph processing.

6.1 Methodology

We compiled the built-in benchmarks of Tripoline using g++ 8.3, and ran the experiments on a 32-core Linux server. The server is equipped with Intel Xeon CPU E5-2683 v4 CPU and 512GB memory, running on CentOS 7.9.

The experiments used a set of four real-world large graphs whose statistics are listed in Table 2. Like many existing graph systems, such as PowerGraph [14], PowerLyra [2], and Tigr [27], Tripoline mainly targets power-law graphs, which are more common in real-world applications. Thus,

Table 2. Statistics of Input Graphs

Graph	Type	|V|	|E|	Avg. Out-Degree
Orkut	undirected	3.1M	234M	76.3
Friendster	undirected	68M	2.9B	28.9
LiveJournal	directed	4.8M	69M	14.2
Twitter	directed	41M	1.5B	35.3

this evaluation focuses on such kind of graphs. Similar to prior work [23, 35, 43], we assume that a substantial portion of edges – 50%, 60%, and 70%, has been streamed in, then the remaining edges of the graph are streamed in batches of randomly selected edges. By default, we set the update batch size to 10K. Note that, under the design of Tripoline, the impact of update batch size is limited to the standing query evaluation, which has been intensively studied in the evaluation of Aspen [7]. But, for completeness, we have included results for different batch sizes (from 1K to 500K).

As to the number of standing queries K, by default, we set it to 16. To demonstrate the tradeoff between benefits and costs in adopting multiple standing queries (see Section 4.5), we also vary the value of K from 1 to 64 and report their impacts to the standing and user query evaluations.

For each benchmark, we randomly selected 256 non-trivial user queries (whose source vertices are of degree more than two). After a batch of graph updates have been applied and the evaluation of standing queries have been re-stabilized, we evaluated each of the 256 user queries three times repetitively and reported the averaged performance. To obtain sufficient samples, we collected the performance results from the first five consecutive batches of updates at each preset starting point (50%, 60%, and 70% portions of edges).

6.2 Speedups

Table 3 lists the speedups of Δ-based incremental evaluation of user queries over the non-incremental query evaluation and the average time of the former. Overall, we observe a wide range of speedups across benchmarks. The highest come from the case of Viterbi (17.6-41.5×), while the lowest are observed on SSNSP (1.0-1.2×). In between, the results show significant performance improvements in the cases of SSWP (9.3-36.1×), SSNP (10.2-30.5×), and SSR (5.0-11.7×), and modest speedups in the remaining cases: SSSP (1.3-2.5×), BFS (1.0-1.6×), and Radii (1.1-1.2×). This large variation of speedups clearly indicates that the effectiveness of Δ-based incremental evaluation depends on the graph problems, in particular, their graph triangle inequalities. As mentioned in Section 4.4, the effectiveness can be measured more directly by the activation ratio R_{act}. Table 4 reports this ratio for cases where the graph is 60% loaded.

Overall, we find that the results are consistent with the speedups – lower activation ratios usually correspond higher speedups. More specifically, we find R_{act} is extremely low (less than 1%) in the cases of SSWP, SSNP, and Viterbi, which means more than 99% of the vertex activations are avoided

Table 3. Speedups of Δ-based Incremental Evaluation over Non-Incremental Evaluation.

Each entry is in the format of *average speedup* [*speedup standard deviation, average time (seconds) with incremental eval.*] of 256 user queries. The highest and lowest speedups for each benchmark are bold.

Graph	SSSP	SSWP	Viterbi	BFS	SSNP	SSR	Radii	SSNSP
OR-50	**2.52** [1.88, 0.15]	31.70 [6.76, 0.01]	40.16 [5.17, 0.01]	1.23 [1.04, 0.12]	26.30 [5.42, 0.01]	10.40 [0.31, 0.01]	1.21 [0.05, 2.22]	1.09 [0.18, 0.25]
OR-60	2.42 [1.69, 0.17]	33.91 [6.45, 0.01]	37.94 [3.95, 0.01]	1.25 [1.20, 0.13]	29.06 [5.30, 0.01]	10.86 [0.27, 0.01]	1.22 [0.06, 2.43]	1.09 [0.18, 0.27]
OR-70	2.45 [1.82, 0.18]	34.88 [6.14, 0.01]	39.90 [4.29, 0.01]	1.30 [1.38, 0.15]	**30.47** [5.13, 0.01]	**11.70** [0.61, 0.01]	**1.23** [0.05, 2.75]	1.10 [0.19, 0.29]
FR-50	**1.34** [0.13, 10.90]	29.69 [5.32, 0.40]	35.49 [5.38, 0.46]	1.02 [0.20, 6.62]	17.30 [3.06, 0.45]	9.28 [0.27, 0.47]	1.16 [0.05, 50.09]	**1.00** [0.03, 8.59]
FR-60	**1.34** [0.11, 12.26]	35.23 [5.86, 0.38]	**41.48** [5.31, 0.38]	1.02 [0.24, 7.16]	18.77 [2.96, 0.45]	10.44 [0.23, 0.45]	1.18 [0.04, 56.43]	**1.00** [0.03, 9.58]
FR-70	**1.34** [0.18, 13.79]	**36.09** [5.76, 0.42]	39.95 [3.87, 0.45]	**1.01** [0.11, 8.63]	20.56 [3.04, 0.45]	11.43 [0.30, 0.45]	1.16 [0.05, 61.52]	**1.00** [0.03, 9.99]
LJ-50	1.68 [0.62, 0.14]	**9.27** [1.88, 0.02]	22.91 [4.60, 0.02]	1.10 [0.30, 0.08]	**10.23** [2.08, 0.02]	**4.94** [0.39, 0.02]	**1.14** [0.03, 1.28]	1.03 [0.11, 0.18]
LJ-60	1.81 [0.87, 0.13]	11.56 [2.30, 0.01]	26.88 [5.47, 0.02]	1.12 [0.36, 0.07]	11.53 [2.35, 0.02]	5.50 [0.43, 0.02]	1.16 [0.05, 1.31]	1.03 [0.11, 0.20]
LJ-70	1.74 [0.73, 0.15]	10.60 [2.00, 0.02]	23.4 [4.60, 0.02]	1.12 [0.33, 0.08]	12.56 [2.49, 0.02]	6.01 [0.46, 0.02]	1.17 [0.04, 1.49]	1.03 [0.11, 0.20]
TW-50	1.97 [1.25, 1.24]	13.17 [2.28, 0.14]	**17.61** [2.41, 0.13]	1.49 [0.80, 0.85]	12.76 [1.77, 0.13]	7.87 [0.13, 0.15]	1.16 [0.07, 11.42]	1.16 [0.32, 2.18]
TW-60	1.95 [1.18, 1.45]	15.97 [2.55, 0.13]	19.14 [2.33, 0.13]	1.56 [0.98, 0.94]	13.42 [1.81, 0.14]	8.32 [0.25, 0.15]	1.15 [0.07, 11.85]	1.18 [0.34, 2.27]
TW-70	2.11 [1.84, 1.56]	17.23 [2.61, 0.13]	21.41 [2.74, 0.13]	**1.61** [1.12, 0.98]	15.94 [2.07, 0.13]	9.21 [0.20, 0.15]	1.19 [0.06, 13.51]	**1.20** [0.41, 2.51]
avg.	1.89	23.28	30.52	1.24	18.24	8.83	1.18	1.08

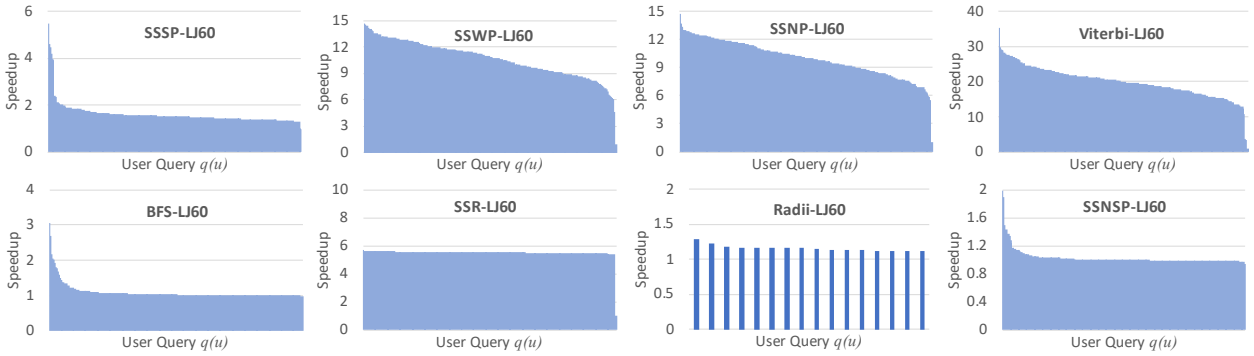


Figure 11. Speedup Distributions of 256 User Queries (16 queries in the case of Radii; x-axis is for the user queries while y-axis is for the speedups of Δ-based incremental evaluation; the user queries are sorted by the corresponding speedups.)

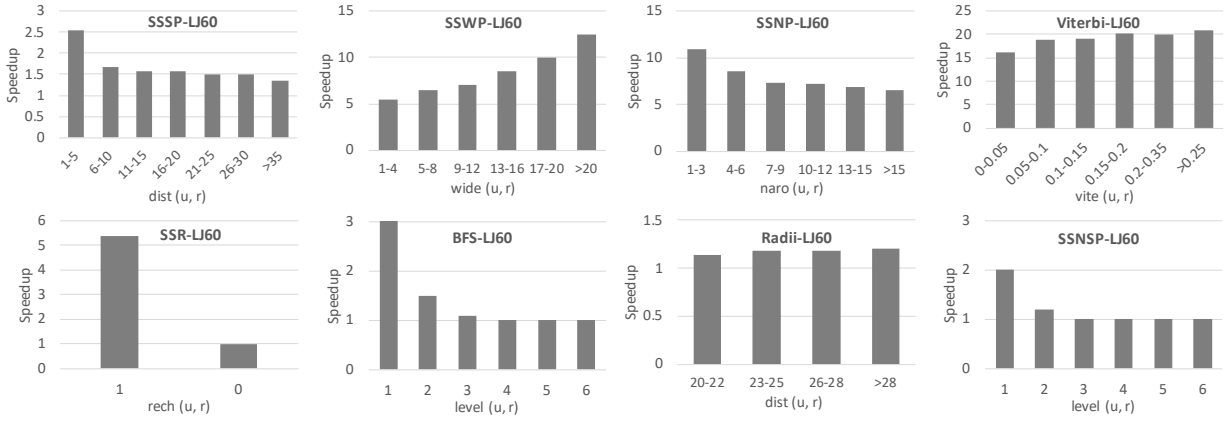


Figure 12. Correlations between Speedups and $property(u, r)$ for Verifying the Standing Query Selection Heuristic.

by incremental evaluation. Our further investigation reveals an interesting fact: the initial values $\Delta(u, r)$ of incremental evaluation are nearly all stable values, that is, the "=" part of the inequality holds – the first case of benefits we discussed in Section 4.4. There could be multiple reasons causing this phenomenon. One of them is the min-max nature of the graph problems. In the cases of SSWP and SSNP, the whole vertex function is based on the calculation of `min` and `max`. In these cases, it is not hard to prove that, for undirected graphs or strongly connected components (SCC) of a directed graph, if $property(u, r) \neq property(r, x)$, then the inequality turns to be the equality, thus, the initial values are already stable. For Viterbi, one key reason for the high stable ratio could be related to the max-division operation in the vertex function. The function tries to choose the edge with the lowest weight to propagate the probability – dividing value

Table 4. Vertex Activation Ratio of Δ-based Incremental Evaluation over Non-Incremental Evaluation.

Each entry is: average [standard derivation] of 256 user queries

	OR-60	FR-60	LJ-60	TW-60
SSSP	44.4% [13.1%]	61.7% [4.8%]	56% [12.2%]	52.8% [11.1%]
SSWP	1.9E-7 [9.0E-8]	1.3E-8 [3.6E-9]	0.79% (8.8%)	4.0E-8 [3.0E-8]
Viterbi	3.5E-7 [8.9E-7]	6.7E-8 [3.2E-7]	0.95% [9.1%]	1.7E-7 [2.8E-7]
BFS	82.2% [18.2%]	98% [6.9%]	89.4% [16.5%]	65.8% [22.6%]
SSNP	1.9E-7 [1.4E-7]	1.4E-8 [9.4E-9]	0.78% [8.8%]	3.6E-8 [2.3E-8]
SSR	3.3E-7 [0]	1.7E-8 [0]	0.78% [8.8%]	3.2E-8 [2.8E-9]
Radii	98.9% [3.7%]	91.9% [4.5%]	92.21% [4.06%]	93.9% [6.8%]
SSNSP	98.9% [4.3%]	99.97% [0.2%]	98.58% [4.88%]	94.9% [10.1%]

by the edge weight, and as we know, the lowest edge weight is one, thus the probability is likely to stay the same simply because "$vite(v_1, v_2)$ divided by 1 equals $vite(v_1, v_2)$". Back to the inequality, as long as there exists a path from u to r or a path from r to x where the edge weights are all ones, then the inequality becomes an equality. This effect can be significantly amplified by the power-law nature of the graphs, where u, r, and x are only a few hops apart, thus the conditions are very likely to become true.

At the other end of the stable ratio spectrum, Radii and SSNSP show the highest stable ratios (over 90%), meaning that less than 10% of vertex function activations are actually saved by incremental evaluation. Note that, even though Radii mainly involves a group of SSSP evaluations, its stable ratios are much higher than those of SSSP. This is because the number of vertex activations in Radii is bottlenecked by the slowest SSSP query – the one with the most number of vertex activations. For SSNSP, our evaluation involves two rounds: (i) a BFS round which computes the level of each vertex and (ii) a counting round which counts the number of paths corresponding to the lowest levels. The activation ratios reported in Table 4 are for the second round. In this case, the main problem comes from the *conditional* inequality as shown in Figure 6-(d). Our profiling shows that the condition is false for 90% of the cases during the initialization, which substantially limits the effectiveness of Δ-based incremental evaluation, resulting in a high activation ratio.

Besides the aggregated speedups shown in Table 3, we also report the speedups of the 256 individual queries on graph LiveJournal with 60% edges loaded in Figure 11. From the results, we can see three patterns roughly. For SSSP, BFS, and SSNSP, the speedup distributions are mostly biased, followed by SSWP, SSNP, Viterbi, and Radii, and finally, the distribution of SSR is almost uniform. The variations, to a large extent, depend on the $property(u, r)$, which connects the user query $q(u)$ and standing query $q(r)$. We thus discuss it together with the standing query selection next.

6.3 Standing Query Selection

Standing query selection is critical to the effectiveness of Δ-based incremental evaluation. To examine its impact, we grouped the speedups by $property(u, r)$ – the heuristic that we use for selecting the standing query (see Section 4.5). The results are reported in Figure 12. In the cases of SSSP, SSWP, SSNP, BFS, and SSNSP, there are clear correlations between $property(u, r)$ and the speedup; for Viterbi, the trend is less obvious, but still observable; for Radii, no significant enough correlation is observed; finally, for SSR, as the property is binary, we will discuss it separately. Note that whether the trend is increasing or decreasing depends on the comparison operator $\geq$. For SSSP, SSNP, BFS, Radii, and SSNSP, $\geq$ is $\geq$, while for SSWP and Viterbi, $\geq$ is $\leq$. From this perspective, the trends align well with our standing query selection heuristic – the "lower" the $property(u, r)$ is, the higher the speedup is achieved. Note that, in the case of SSR, the heuristic always chooses the standing query with $property(u, r) = 1$, which is also the one with a higher speedup. For Radii, we used the maximum distance among 16 SSSP queries, in which case the "averaging effect" blurs the correlation.

Also, note that, in the cases of SSSP, BFS, and SSNSP, the speedups are more sensitive to $property(u, r)$ when its value is low, which explains the biased speedup distributions in the corresponding graph problems shown in Figure 11.

Besides the selection heuristic, another important factor to the performance is the number of standing queries – K (see Section 4.4). A larger K offers more options for selecting the standing query, thus potentially making the incremental evaluation more effective; on the other hand, a larger K can also increase the costs: (i) the time for incremental standing query evaluation; and (ii) the time for selecting one from the K standing queries for applying the triangle inequality. For the latter, as the selection simply accesses K vertex values in the property arrays of K standing queries and chooses one based on Equation 12, the runtime cost is negligible. For the former, we report the standing query evaluation time for K from 1 to 64 in Table 5 (numbers in brackets).

Table 5 also reports how K affects the speedups. For SSSP, BFS, and SSNSP, larger K tends to yield higher speedups; For the others, there are no similar trends, which means that adding more standing queries does not necessarily increase the effectiveness of Δ-based incremental evaluation; for such cases, a single standing query is sufficient. Note that, when K increases, the evaluation time of standing queries only increases sub-linearly, thanks to the *batch* mode execution. As to the space cost, the value can be computed by (8 + 2) bytes × $Bsize$ × $|V|$, where 8 is the size of vertex value (`double`/`long`) and 2 is the two masks (`boolean`) of vertex activeness in the prior and current iterations. In general, users can tune K based on the sensitivity of their graph problem and the resource constraints.

6.4 Graph Streaming

Next, we briefly report the impact of the graph update batch size on the standing query evaluation. A detailed evaluation can be found in Aspen [7]. Table 6 shows the standing query evaluation time with update batch size varying from 1K to

Table 5. Benefits and Costs of Incrementally Evaluating K Standing Queries (on graph TW-60).

Each entry is: avg. user query speedup [standing queries eval. time(s)]

#LQ	1	2	4	16	64
SSSP	1.43 [0.30]	1.43 [0.45]	1.70 [0.71]	1.95 [1.42]	2.36 [4.73]
SSWP	16.28 [0.30]	15.98 [0.47]	15.53 [0.71]	15.97 [1.23]	14.97 [3.51]
Viterbi	18.63 [0.30]	17.87 [0.45]	19.09 [0.69]	19.14 [1.20]	17.84 [3.44]
BFS	1.03 [0.32]	1.04 [0.43]	1.23 [0.67]	1.56 [1.29]	1.86 [4.45]
SSNP	13.44 [0.37]	13.09 [0.45]	13.24 [0.75]	13.42 [1.37]	13.84 [3.92]
SSR	8.39 [0.35]	8.60 [0.45]	8.22 [0.66]	8.32 [1.36]	8.12 [4.05]
Radii	1.11 [0.36]	1.13 [0.53]	1.16 [0.97]	1.15 [1.59]	0.84 [4.68]
SSNSP	1.01 [1.74]	1.00 [0.58]	1.09 [0.91]	1.18 [2.10]	1.28 [6.46]

Table 6. Standing Query Evaluation Time under Different Update Batch Sizes on LJ-60 and FR-60.

Graph	Bsize	SSSP	SSWP	Viterbi	BFS	SSNP	SSR	Radii	SSNSP
LJ-60	1K	0.09	0.08	0.09	0.09	0.09	0.07	0.13	0.17
	10K	0.13	0.10	0.10	0.09	0.11	0.08	0.17	0.19
	50K	0.15	0.12	0.12	0.11	0.13	0.09	0.20	0.22
	100K	0.16	0.14	0.14	0.11	0.16	0.10	0.21	0.22
	500K	0.23	0.18	0.19	0.17	0.20	0.14	0.29	0.27
FR-60	1K	2.09	1.66	1.78	1.86	1.72	1.73	2.29	3.77
	10K	2.50	1.95	2.00	2.09	1.87	1.78	2.73	4.04
	50K	2.69	2.16	2.36	2.30	2.22	2.03	3.11	3.52
	100K	3.08	2.52	2.60	2.67	2.44	2.38	3.30	3.88
	500K	4.14	3.52	3.55	3.70	3.59	3.33	4.30	4.83

500K. The results show that the evaluation time increases sub-linearly as the batch size increases. The main reason for the sub-linear increase is that computations for handling different new edges are largely shared. For example, many new edges may appear on the same paths, thus sharing the activations of vertices along the paths. Moreover, the efficient data structure (a purely functional tree) ensures fast graph data accesses for incremental query evaluation.

6.5 Integration into Differential Dataflow

Though Tripoline is implemented based on Aspen [7], the idea of triangle inequality-based optimization may also be adopted in other streaming graph systems. To demonstrate its generality, we examined the potential of adopting it in a state-of-the-art general-purpose streaming framework, called *Differential Dataflow* (DD) [24, 26].

In fact, the latest version of DD also supports inter-query sharing, called *shared arrangements* [24]. In earlier versions of DD, each query [8] needs to maintain an indexed state over the input stream independently. In the context of streaming graphs, this means that each query needs to maintain its own indexed graph (for outgoing and/or incoming edges) over a stream of edge pairs. This creates unnecessary redundancies when different graph queries want to access the same input stream (edge-pair stream). Shared arrangements address this issue by allowing different queries to share the same indexed state (graph), rather than maintaining its own copy. Note that shared arrangements are an orthogonal improvement to the proposed triangle inequality optimization—the former shares the indexed graph data structure across queries while the latter "shares" the query evaluation state, that is, the vertex values (e.g., distances of all vertices to the source vertex in SSSP) across queries. The latter requires to establish the triangle inequalities to be applicable.

Table 7. Performance of Differential Dataflow with Triangle Inequality Optimization on LJ and TW at 60% and 100%.

(DD-SA: differential dataflow with shared arrangements; DD-SA-Tri: DD-SA with triangle inequality optimization)

Graph	Method	BFS	SSSP	SSWP
LJ-60	DD-SA	0.97s	6.90s	3.50s
	DD-SA-Tri	0.93s	2.68s	0.48s
	Speedup	[1.04×]	[2.57×]	[7.29×]
TW-60	DD-SA	6.91s	42.88s	22.97s
	DD-SA-Tri	7.23s	10.74s	5.75s
	Speedup	[0.96×]	[3.99×]	[3.99×]
LJ-100	DD-SA	1.10s	8.41s	4.63s
	DD-SA-Tri	1.11s	3.24s	0.52s
	Speedup	[0.99×]	[2.60×]	[8.90×]
TW-100	DD-SA	10.69s	58.63s	32.68s
	DD-SA-Tri	10.71s	14.72s	7.74s
	Speedup	[1.00×]	[3.98×]	[4.22×]

Table 8. Reduction of `reduce` Operations for DD-SA with Triangle Inequality Optimization on LJ-100.

Graph	Method	BFS	SSSP	SSWP
LJ-100	DD-SA	9156594	30418846	20622003
	DD-SA-Tri	8956638	17570555	6292821
	Reduction	[1.02×]	[1.73×]	[3.28×]

Experiment Setup. We pulled the latest version of DD from its GitHub repository [9]. To integrate the triangle inequality optimization, we added a `filter` to its graph processing dataflow. The `filter` applies a predicate to each element of a collection, and removes those for which the predicate returns `false`. In specific, the predicates are constructed based on triangle inequality $\Delta(u, r) \geq property(u, x)$ (see Equations 2 and 9). For other operators used in the dataflow (such as `join_map`, `concat`, and `reduce`), we kept them intact.

Note that the above integration may not be the only way to adopt triangle inequality optimization into DD. We choose this design for its simplicity and modularity - it isolates the modifications to one dataflow operator, leaving other parts of the graph processing dataflow intact. A more intrusive integration that yields better performance might be possible, but requires a redesign of the existing DD to some extent.

Due to space limits, we focus our evaluation on three query benchmarks (BFS, SSSP, and SSWP) and two graphs (LJ and TW), at 60% and 100% loaded ratios. For each configuration, we issued 256 queries (the same as the prior experiments) and collected the average time of query evaluation.

[8] Here, a query refers to a type of queries in our context.

[9] https://github.com/TimelyDataflow/differential-dataflow, Jan 22, 2021.

Performance Results. Table 7 reports the performance with and without the triangle inequality optimization. Note that the baseline (DD-SA) is the DD with shared arrangements enabled. In general, the results are of similar trends as those reported for Tripoline (see Table 3): (i) for SSSP and SSWP, the speedups are more significant, ranging from 2.57× to 3.99× for SSSP and 3.99× to 8.90× for SSWP; (ii) by contrast, the speedups for BFS are limited, actually they are close to one. In the context of DD, the effectiveness of triangle inequality optimization can be reflected by the number of invocations of the downstream `reduce` operator, which are shown in Table 8. For SSSP and SSWP, there are significant reduction in the invocations of `reduce` operator, while for BFS, the reduction is very limited. These results align with the speedups of the three types of queries.

7 Related Work

When encountered with streaming graphs, for the evaluation of graph queries, the following two approaches have been considered: general solutions that apply to wide range of graph problems; and custom solutions that apply to only specific problems. While these works employ incremental processing, none of them perform incremental processing in the scenario of a *new query* addressed by Tripoline.

General Solutions for Streaming Graphs. These systems allow users to express a broad range of graph algorithms. For example, targeting general iterative analytics, Tornado [35] organizes the computations into a "main loop" and several branches. The former computes the approximate results, based on which the latter finish the user query evaluation. Sharing the high-level design, Kineograph [3] focuses more on incremental graph analytics over fast-changing streaming graphs along with push and pull models. Some other general streaming graph systems include GIM-V [39] which employs an incremental graph processing based on matrix-vector operations, and Naiad [26] which incorporates differential data flow to perform iterative and incremental algorithms. The high-level system design of Tripoline is inspired by the above systems. However, the key difference lies in the *generalization* of incremental evaluation for queries that are different from those in the "main loop".

Besides the above systems, there are also designs of data structures for supporting high-throughput graph updates, such as STINGER [9] and Aspen [7]. As mentioned earlier, the graph update engine used by Tripoline is Aspen.

Fan and others discussed graph computations that can be incrementally performed [12]. Their work focused on the theoretical *boundness* of incremental computation. However, they assume that the queries for incremental evaluation are known a priori, whereas this work focuses on incremental evaluation of queries without a priori knowledge.

Custom Solutions for Streaming Graphs. These works develop specialized streaming algorithms to solve different graph problems. For example, Ediger and others [8] focuses on the clustering coefficients in streaming data analytics and designed an approximation method using bloom filters. A similar technique is also used to correctly maintain *connected components* in social networks [10, 29]. The basic idea is to use the set intersection of neighborhood vertices to quickly determine connectivity and construct a spanning tree for each component. The algorithm relies on multiple concurrent graph traversals to maximize parallelism. Some other custom solutions have also been developed for *graph clustering* [48], *graph partitioning* [38, 40], and *connectivity checks* [30, 36].

Other Related Scenarios and Solutions. Besides the above, some prior studies [16, 19, 28, 42] focus more on the evolving nature of graphs and the analysis of multiple snapshots of a changing graph. In this scenario, the snapshots are available a priori. For example, Chronos [16] is a storage and execution engine to process temporal graphs that also uses incremental processing. Vora and others [42] reorder computations to to aggregate communications across graph snapshots and leverage previously computed (potentially partial) results to perform incremental processing across graph snapshots. GraphTau [19] maintains a history of vertex values over time and rectifies inaccuracies by reverting back the values.

Finally, existing static graph systems [2, 14, 15, 21, 22, 31, 32, 34, 37, 44, 45] can process graph snapshots one after the other. Obviously, this approach would not be able to take advantage of incremental processing.

8 Conclusion

This work reveals a fundamental limitation in the existing streaming graph systems – lack of incremental evaluation for queries without a priori knowledge. To address the limitation, this work proposes to leverage the graph triangle inequalities that can be naturally derived from vertex-specific graph problems to enable such capabilities. This idea leads to a generalized incremental processing design for vertex-specific queries, in which the correctness is ensured by the triangle inequality and proper design of the vertex functions, and the efficiency is optimized based on the "distance" between the user and standing queries. Finally, our evaluation of the developed system Tripoline confirms the effectiveness of the proposed techniques for a spectrum of graph problems on real-world graphs.

Acknowledgements

We thank the reviewers and our shepherd Dr. Christopher Rossbach for their constructive feedback and help. This material is based upon the work supported in part by National Science Foundation Grants CCF-2028714, CCF-2002554 and CCF-1813173. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Markus Bläser. A new approximation algorithm for the asymmetric tsp with triangle inequality. *ACM Transactions on Algorithms (TALG)*, 4(4):1–15, 2008.

[2] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing (TOPC)*, 5(3):1–39, 2019.

[3] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 85–98, 2012.

[4] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proc. of the VLDB Endowment*, 8(12):1804–1815, 2015.

[5] Atish Das Sarma, Sreenivas Gollapudi, Marc Najork, and Rina Panigrahy. A sketch-based distance oracle for web-scale graphs. In *Proceedings of the third ACM international conference on Web search and data mining*, pages 401–410, 2010.

[6] Fethi Demim, Kahina Louadj, and Abdelkrim Nemra. Path planning for unmanned ground vehicle. In *2018 5th International Conference on Control, Decision and Information Technologies (CoDIT)*, pages 748–750. IEEE, 2018.

[7] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 918–934, 2019.

[8] David Ediger, Karl Jiang, Jason Riedy, and David A Bader. Massive streaming data analytics: A case study with clustering coefficients. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE, 2010.

[9] David Ediger, Rob McColl, Jason Riedy, and David A Bader. Stinger: High performance data structure for streaming graphs. In *IEEE Conference on High Performance Extreme Computing*, pages 1–5. IEEE, 2012.

[10] David Ediger, Jason Riedy, David A Bader, and Henning Meyerhenke. Tracking structure of streaming social networks. In *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1691–1699. IEEE, 2011.

[11] Charles Elkan. Using the triangle inequality to accelerate k-means. In *Proceedings of the 20th international conference on Machine Learning (ICML-03)*, pages 147–153, 2003.

[12] Wenfei Fan, Chunming Hu, and Chao Tian. Incremental graph computations: Doable and undoable. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 155–169, 2017.

[13] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, and Jiaxin Jiang. Grape: Parallelizing sequential graph computations. *Proceedings of the VLDB Endowment*, 10(12):1889–1892, 2017.

[14] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, 2012.

[15] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. GraphX: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation*, pages 599–613, 2014.

[16] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: a graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[17] Thomas Little Heath et al. *The thirteen books of Euclid's Elements*. Courier Corporation, 1956.

[18] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Sublinear-time decremental algorithms for single-source reachability and shortest paths on directed graphs. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 674–683, 2014.

[19] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. Time-evolving graph processing at scale. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, pages 1–6, 2016.

[20] Jüri Lember, Dario Gasbarra, Alexey Koloydenko, and Kristi Kuljus. Estimation of Viterbi path in Bayesian hidden Markov models. *METRON*, 77(2):137–169, 2019.

[21] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. GraphLab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

[22] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[23] Mugilan Mariappan and Keval Vora. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[24] Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Timothy Roscoe. Shared arrangements: practical inter-query sharing for streaming dataflows. *Proceedings of the VLDB Endowment*, 13(10):1793–1806, 2020.

[25] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. *ACM Sigplan Notices*, 47(8):117–128, 2012.

[26] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455, 2013.

[27] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. Tigr: Transforming irregular graphs for gpu-friendly graph processing. *ACM SIGPLAN Notices*, 53(2):622–636, 2018.

[28] Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. On querying historical evolving graph sequences. *Proceedings of the VLDB Endowment*, 4(11):726–737, 2011.

[29] Jason Riedy and Henning Meyerhenke. Scalable algorithms for analysis of massive, streaming graphs. *SIAM Parallel Processing for Scientific Computing*, 2012.

[30] Liam Roditty and Uri Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. *SIAM Journal on Computing*, 45(3):712–733, 2016.

[31] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 410–424, 2015.

[32] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488, 2013.

[33] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. Subway: minimizing data transfer during out-of-gpu-memory graph processing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[34] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, pages 1–12, 2013.

[35] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. Tornado: A system for real-time iterative analysis over evolving data. In *Proceedings of the 2016 International Conference on Management of Data*, pages 417–430, 2016.

[36] Yossi Shiloach and Shimon Even. An on-line edge-deletion problem. *Journal of the ACM (JACM)*, 28(1):1–4, 1981.

[37] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 135–146, 2013.

[38] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1222–1230, 2012.

[39] Toyotaro Suzumura, Shunsuke Nishii, and Masaru Ganse. Towards large-scale graph stream processing platform. In *Proceedings of the 23rd International Conference on World Wide Web*, pages 1321–1326, 2014.

[40] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, pages 333–342, 2014.

[41] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[42] Keval Vora, Rajiv Gupta, and Guoqing Xu. Synergistic analysis of evolving graphs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(4):1–27, 2016.

[43] Keval Vora, Rajiv Gupta, and Guoqing Xu. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 237–251, 2017.

[44] Keval Vora, Guoqing Xu, and Rajiv Gupta. Load the edges you need: A generic I/O optimization for disk-based graph processing. In *2016 USENIX Annual Technical Conference (USENIX ATC'16)*, pages 507–522, 2016.

[45] Kai Wang, Guoqing Xu, Zhendong Su, and Yu David Liu. Graphq: Graph query processing with abstraction refinement—scalable and programmable analytics over very large graphs on a single {PC}. In *2015 USENIX Annual Technical Conference (USENIX ATC'15)*, pages 387–401, 2015.

[46] Zheng Wang and Jon Crowcroft. Quality-of-service routing for supporting multimedia applications. *IEEE Journal on selected areas in communications*, 14(7):1228–1234, 1996.

[47] Kaige Yang and Laura Toni. Graph-based recommendation system. In *2018 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pages 798–802. IEEE, 2018.

[48] Mindi Yuan, Kun-Lung Wu, Gabriela Jacques-Silva, and Yi Lu. Efficient processing of streaming graphs for evolution-aware clustering. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 319–328, 2013.