The VLDB Journal (2024) 33:781–806
https://doi.org/10.1007/s00778-024-00838-z

REGULAR PAPER

Ingress: an automated incremental graph processing system

Shufeng Gong[1] · Chao Tian[2] · Qiang Yin[3] · Zhengdong Wang[3] · Song Yu[1] · Yanfeng Zhang[1] · Wenyuan Yu[4] · Liang Geng[5] · Chong Fu[1] · Ge Yu[1] · Jingren Zhou[4]

Received: 24 February 2023 / Revised: 10 October 2023 / Accepted: 1 January 2024 / Published online: 20 February 2024

Abstract

The graph data keep growing over time in real life. The ever-growing amount of dynamic graph data demands efficient techniques of incremental graph computation. However, incremental graph algorithms are challenging to develop. Existing approaches usually require users to manually design nontrivial incremental operators, or choose different memoization strategies for certain specific types of computation, limiting the usability and generality. In light of these challenges, we propose Ingress, an automated system for *incremental graph processing*. Ingress is able to deduce the incremental counterpart of a batch vertex-centric algorithm, without the need of redesigned logic or data structures from users. Underlying Ingress is an automated incrementalization framework equipped with four different memoization policies, to support all kinds of vertex-centric computations with optimized memory utilization. We identify sufficient conditions for the applicability of these policies. Ingress chooses the best-fit policy for a given algorithm automatically by verifying these conditions. In addition to the ease-of-use and generalization, Ingress outperforms state-of-the-art incremental graph systems by 12.14× on average (up to 49.23×) in efficiency.

Keywords Incrementalization · Flexible memoization · Graph computing systems

1 Introduction

In response to the increasing need for processing large-scale graphs, under various scenarios, *e.g.,* anti-fraud, bioinformatics, fintech, recommendation and social network analysis, a number of graph processing systems have been proposed [10, 14, 29, 47, 55, 56, 59]. However, these systems are mainly designed based on the assumption that computation is performed over static graphs. When the underlying graph is updated with input changes, *e.g.,* edge insertions and deletions, they have to reperform the entire computation on the updated graph starting from scratch. Such recomputation is costly as real-life graphs easily have billions of nodes and trillions of edges, *e.g.,* e-commerce graphs [27] and they are constantly changed, *e.g.,* the relationships between users and items in e-commerce transactions [5].

✉ Qiang Yin
q.yin@sjtu.edu.cn

Shufeng Gong
gongsf@mail.neu.edu.cn

Chao Tian
tianchao@buaa.edu.cn

Zhengdong Wang
lnwzd2009@sjtu.edu.cn

Song Yu
yusong@stumail.neu.edu.cn

Yanfeng Zhang
zhangyf@mail.neu.edu.cn

Wenyuan Yu
wenyuan.ywy@alibaba-inc.com

Liang Geng
geng.161@osu.edu

Chong Fu
fuchong@mail.neu.edu.cn

Ge Yu
yuge@mail.neu.edu.cn

Jingren Zhou
jingren.zhou@alibaba-inc.com

[1] Northeastern University, Shenyang, China

[2] Beihang University, Beijing, China

[3] Shanghai Jiao Tong University, Shanghai, China

[4] Alibaba Group, Hangzhou, China

[5] Ohio State University, Columbus, USA

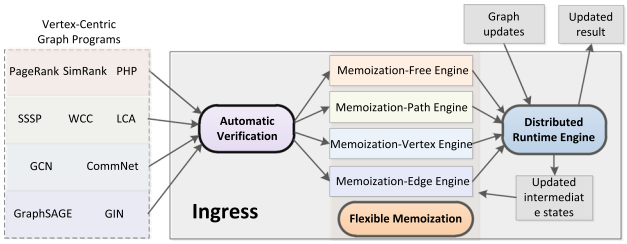


Fig. 1 Overall structure of Ingress

These highlight the need for incremental graph computation. That is, we apply a batch algorithm to compute the result over the original graph G once, followed by employing an *incremental algorithm* to adjust the old result in response to the input changes ΔG to G. It ensures that the adjusted result is the same as that recomputed by the batch algorithm on the updated graph. In practice, real-world changes are typically small, *e.g.,* English Wikipedia was expanded with on average less than 600 new articles per day out of 5 million articles during 2019 [46]. In addition, given small input changes ΔG, it is common to find a considerable overlap between the computation over G and the recomputation on the new graph updated with ΔG. Therefore, by making use of the memoized previous intermediate result, incremental computation can reduce unnecessary recomputation and is often more efficient.

The benefit of incremental computation has led to the development of several incremental graph processing systems, notably Tornado [44], GraphIn [43], KickStarter [50] and GraphBolt [31]. They adopt the *vertex-centric model*, where the same user-defined function is executed in parallel at each vertex, and vertices exchange updates with each other by message passing. The vertex-centric model can naturally express iterative graph computation, *e.g.,* PageRank [38] and single source shortest path (SSSP) [12].

While the existing systems [31, 43, 44, 50] have eliminated redundant recomputation, they are limited by two major drawbacks. (i) Nontrivial user intervention is required, *e.g.,* GraphIn [43] and GraphBolt [31] ask users to manually deduce the incremental operators, and Tornado [44] and KickStarter [50] require users to make sure that the corresponding batch computation satisfies certain properties. (ii) These systems use different memoization policies and achieve different levels of generality. For instance, GraphBolt, aiming at a high level of generality to support wide variety of applications, needs to memoize a large number of intermediate results. Although KickStarter, GraphIn and Tornado memoize small amount of states, they only support a specific class of graph computations that satisfy certain properties. Our new findings show that some incremental algorithms do not even employ any memoized intermediate states at all. However, checking the properties of the graph computations and choosing the best memoization policies manually is usually difficult for non-expert users.

Then two questions are naturally raised. Can we build an incrementalization framework that automatically converts a generic user-specified batch graph algorithm into an incremental algorithm? Furthermore, can this framework deduce incremental algorithms with different memoization policies such that the memoized intermediate results are as few as possible?

Ingress. To answer these questions, we develop Ingress, an automated vertex-centric system for incremental graph processing. The overall structure of Ingress is shown in Fig. 1. Given a batch algorithm $\mathcal{A}$, Ingress verifies the characteristics of $\mathcal{A}$ and deduces an incremental counterpart $\mathcal{A}_\Delta$ automatically. It selects an appropriate memoization engine to record none or part of run-time intermediate states. Upon receiving graph updates, Ingress executes $\mathcal{A}_\Delta$ to deliver updated results with the help of memoized states.

The rationale behind Ingress is (a) identifying the differences between the prior run and the recomputation over the new graph, and (b) enforcing their effects on the old intermediate results. For some graph computations, such effects can be *directly* applied on the previous final results, even without the need of memoizing other intermediate information. In other words, the differences across multiple steps of the iterative computation can be assembled and processed in a singe batch. This is fully leveraged by Ingress to achieve incrementalization with different memoization strategies.

Ingress provides the following features that differ from previous graph incremental processing systems.

Flexible memoization. Ingress aims to deduce an incremental counterpart of a batch vertex-centric algorithm with flexible memoization policies, *i.e.,* deducing $\mathcal{A}_\Delta$ from $\mathcal{A}$, under four different *memoization policies*. Specifically, (1) the *memoization-free* policy records no runtime states of the previous computation, (2) the *memoization-path* policy only records a small portion of critical states (messages), which form a set of paths, (3) the *memoization-vertex* policy records all the vertex states, and (4) the (default) *memoization-edge* policy records all the edge states, *i.e.,* old messages. They can be adopted to incrementalize the batch algorithms of *e.g.,* PageRank, SSSP, forward process of Graph Convolutional Network (GCN-forward) [24], and GraphSAGE [17] with mean aggregator, respectively (see Fig. 1). We also provide the sufficient conditions for these memoization modes that guarantee the correctness of incremental computation. With these four policies, flexible memoization is able to cover the need of incrementalizing all vertex-centric algorithms and support all kinds of incremental computation with optimized memory usage.

Automatic incrementalization. Ingress is able to deduce an incremental counterpart of a batch vertex-centric algorithm. There is no need to manually reshape the data structures or

Table 1 Performance comparison over UK-2005

Algorithm	System	Time (s)	Space (GB)
PageRank	Ingress	2.46	0.31
	GraphBolt	19.34	13.66
SSSP	Ingress	0.34	0.6
	KickStarter	3.65	1.39

the logic of the batch ones, improving ease-of-use. Based on the sufficient conditions that we establish for the applicability of memoization policies, it selects an appropriate memoization policy for each batch algorithm to conduct incrementalization, and guarantees the correctness. Moreover, by transforming sufficient conditions into first-order formulas and applying SMT solver Z3 [36], the satisfiability of the conditions can be automatically verified (Automatic Verification module in Fig. 1). Putting this together with the four incrementalization engines that derive incremental algorithms with the selected memoization policies, Ingress makes the process of incrementalization transparent to users.

Optional execution mode. Ingress incorporates two execution engines, a synchronous engine and an asynchronous one. The synchronous engine is the default one of Ingress since synchronous processing makes correctness and convergence analyses easy [29, 31]. However, synchronous processing also hinders the performance of incremental computation due to the synchronization semantics and global barriers. To address this, the asynchronous execution engine employs two optimizations, *selective processing* and *fast messages propagation*, respectively. Based on the characteristics of each deduced incremental algorithm $\mathcal{A}_\Delta$, Ingress chooses the best execution engine based on the characteristics of $\mathcal{A}_\Delta$.

High performance. In addition to the ease-of-use and generalized reduction of memory consumption, Ingress also achieves high performance runtime. Table 1 compares the performance of Ingress for PageRank and SSSP with GraphBolt and KickStarter, respectively, over the graph UK-2005 that consists of 39 million vertices and 0.9 billion edges. Despite the fact that PageRank (resp. SSSP) is well-supported in GraphBolt (resp. KickStarter), with 1% input graph updates, *i.e.,* $|\Delta G|{=}1\%|G|$, Ingress outperforms GraphBolt and KickStarter by 7.84× and 10.55×, respectively, in response time. Ingress also has the least space cost, thanks to its flexible memoization mechanism. It only incurs 2.26% and 34.26% the space cost of GraphBolt (resp. KickStarter) for PageRank (resp. SSSP).

Contributions and organization. We summarize our contributions as follows.

(1) A general framework for incrementalizing vertex-centric algorithms (Sect. 3). It models the operations of incremental computation in terms of the cancelation of old *invalid* messages and the compensation of new *missing* messages, which can be carried out with the help of different memoization policies.

(2) An analytical foundation for the correctness of the incrementalization *w.r.t.* different memoization policies, including the sufficient conditions for the applicability of the policies (Sect. 4).

(3) Two effective optimizations, namely *selective processing* and *fast message propagation*, to improve the performance of Ingress while still guaranteeing the correctness of incremental processing (Sect. 5).

(4) The automation techniques for selecting appropriate memoization policies for incrementalization, as well as a distributed runtime engine to perform incremental graph computation (Sect. 6).

(5) An extensive evaluation of the incremental graph processing system Ingress (Sect. 7).

This paper extends the conference version [13] in theoretical analyses, optimization techniques and experimental study. (1) We provide detailed formal proofs for the correctness of three incrementalization policies, including MF, MP and MV (Sect. 4). In contrast, only intuitive proof sketches were presented in [13]. (2) We propose two effective optimizations, selective processing and fast messages propagation, to improve the utilization of vertex programs and message propagating efficiency in incremental processing (Sect. 5). (3) We enhance the experimental study as follows. (a) We add new experiments to evaluate the impact of edge weight updates on the performance of Ingress (Sect. 7.2.2); (b) we implement the optimizations introduced in Sect. 5 and conduct experiments to verify their effectiveness (Sect. 7.5); (c) the entire experimental study is also enhanced by comparing Ingress with two state-of-the-art baselines, DZiG and RisGraph (Sects. 7.2–7.4).

2 Preliminaries

We start with a review of basic notations for vertex-centric algorithms and incremental graph computation.

Graphs. We consider *graphs* $G = (V, E, P_G)$, directed or undirected. Here V is a finite set of vertices, $E \subseteq V \times V$ is a set of edges, $P_G = \{P_V, P_E\}$ is a pair of functions such that each vertex v in V (resp. edge e in E) carries a property $P_V(v)$ (resp. $P_E(e)$), which indicates *e.g.,* weight, label or keyword and is possibly empty.

Vertex-centric model. In vertex-centric graph computation models [14, 29], a vertex program $\mathcal{A}$ is executed in parallel on all vertices in the input graph G iteratively. The program $\mathcal{A}$ can be represented by a triple $(\mathcal{H}, \mathcal{U}, \mathcal{G})$, where $\mathcal{H}$ is the *aggregation function* of $\mathcal{A}$, $\mathcal{U}$ is the *update function* and $\mathcal{G}$ is the *propagation function*. Following the Bulk Synchronous Parallel (BSP) model [49], the computation of $\mathcal{A}$ are sepa-

rated into super-steps. In each round i, $\mathcal{A}$ performs

$$\begin{aligned} m_v^i &= \mathcal{H}(M_v^{i-1}), \\ x_v^i &= \mathcal{U}(x_v^{i-1}, m_v^i), \\ m_{v,w}^i &= \mathcal{G}(x_v^i, m_v^i, P_E(v,w)) \quad (\forall w \in \mathsf{Nbr}(v)) \end{aligned} \tag{1}$$

at each vertex v. Here $M_v^{i-1}=\{m_{u,v}^{i-1}|(u,v) \in E\}$ refers to the set of messages received by v at the start of round i; x_v^i (resp. x_v^{i-1}) denotes the state of vertex v in round i (resp. $i-1$); m_v^i is the aggregated result of the messages; and $m_{v,w}^i$ denotes the message sent from v to w at round $i+1$, where w is in the neighbor set $\mathsf{Nbr}(v)$ of v. Intuitively, each vertex v first aggregates the received messages by $\mathcal{H}$; it then applies $\mathcal{U}$ to adjust its state to x_v^i with the aggregated result m_v^i; at last it generates a set of messages by $\mathcal{G}$ and propagates them to its neighbors. In practice, there are many cases that $\mathcal{H}$ and $\mathcal{U}$ have the same logic, *e.g.,* summing the values.

Starting from the initial round, each vertex executes $\mathcal{A}$ in parallel. They communicate via synchronous message passing. The process terminates when no more changes are made to vertex states, *i.e.,* the computation reaches a fixpoint and all vertices are halted [29].

In light of the simplicity and the distributed nature of the model, a large number of vertex-centric algorithms are developed (see [33] for a survey).

Example 1 We show four example vertex-centric algorithms, using the formulation of $(\mathcal{H}, \mathcal{U}, \mathcal{G})$.

(a) PageRank. Consider PageRank that computes the set $\{\mathsf{PR}_v \mid v \in V\}$ of PageRank scores, which is defined as the unique solution to the equations $\{\mathsf{PR}_v = d \times \mathsf{sum}_{(u,v)\in E}\mathsf{PR}_u/N_u + (1-d) \mid v \in V\}$. Here d is a constant damping factor and N_u denotes the number of outgoing neighbors of vertex u in graph G. As opposed to the standard PageRank algorithm that exploits the power method, a delta-based PageRank algorithm [56] can be represented as follows.

- $\mathcal{H}(M_v^{i-1}) = \mathsf{sum}(M_v^{i-1})$;
- $\mathcal{U}(x_v^{i-1}, m_v^i) = \mathsf{sum}(x_v^{i-1}, m_v^i)$;
- $\mathcal{G}(x_v^i, m_v^i, P_E(v,w)) = d \times m_v^i/N_v \quad (\forall w \in \mathsf{Nbr}(v))$.

Observe that $\mathcal{H} = \mathcal{U} = \mathsf{sum}$. Intuitively, each vertex uses its state x_v to store its PageRank score. In particular, $x_v = 0$ and $M_v^0 = \{1-d\}$ for all $v \in V$. Each time a vertex v aggregates messages from its neighbors and updates its state by sum. It converts the aggregated result m_v^i to $d \times m_v^i/N_v$ and propagates it to all its neighbors. As shown in [56], this delta-based PageRank algorithm computes the PageRank scores for all vertices correctly.

(b) PHP. We next consider the PHP (Penalized Hitting Probability) problem [16]. It is to compute a proximity score, in terms of penalized hitting probability, between a given source vertex s and every other vertex in a graph G. A vertex-centric PHP algorithm works as follows.

- $\mathcal{H}(M_v^{i-1}) = \mathsf{sum}(M_v^{i-1})$;
- $\mathcal{U}(x_v^{i-1}, m_v^i) = \mathsf{sum}(x_v^{i-1}, m_v^i)$;
- $\mathcal{G}(x_v^i, m_v^i, P_E(v,w)) = \beta m_v^i P_E(v,w)$.

Here $0 < \beta < 1$ is a predefined parameter and $P_E(v,w)$ is the weight of edge (v,w). Initially, we have $x_v = 0$ for all $v \in V$; $M_s^0 = 1$ and $M_v^0 = 0$ for all $v \neq s$. As in PageRank, we have $\mathcal{H} = \mathcal{U} = \mathsf{sum}$ in PHP. In one iteration, each vertex v aggregates messages from its neighbors and updates its state by sum. Then the aggregated message is transmitted to its neighbor w with probability proportional to the edge weight $P_E(v,w)$.

(c) SSSP. As another example, consider SSSP that computes the shortest distance from a given source s to all vertices in a directed graph G. A vertex-centric algorithm for SSSP operates as follows.

- $\mathcal{H}(M_v^{i-1}) = \mathsf{min}(M_v^{i-1})$;
- $\mathcal{U}(x_v^{i-1}, m_v^i) = \mathsf{min}(x_v^{i-1}, m_v^i)$;
- $\mathcal{G}(x_v^i, m_v^i, P_E(v,w)) = m_v^i + P_E(v,w), (\forall w \in \mathsf{Nbr}(v))$.

Here the state x_v of v indicates the shortest distance from s to v and $P_E(v,w)$ represents the length of (v,w). Initially, we have $x_v^0 = \infty$ and $M_v^0 = \emptyset$ for all $v \neq s$; and $x_s^0 = 0$, $M_s^0 = \{0\}$. Each vertex v aggregates messages and updates its state by using min for both $\mathcal{H}$ and $\mathcal{U}$. It creates and sends a message to each neighbor w, which represents the shortest length of paths through v to w. The algorithm terminates when all shortest distances no longer change, *i.e.,* it reaches a fixpoint.

(d) GCN-forward. Consider the GCN-forward [24] problem. Given a directed graph G and K weight matrices $W_1, \ldots, W_K$, it is to compute the features of each vertex v iterativley based on K weight matrices and the features of the neighbors that are within K-hops away from v. The weight matrices $W_1, \ldots, W_k$ are trained beforehand by multiple graphs; thus they are independent to the input graph G. An algorithm for GCN-forward can be defined as follows.

- $\mathcal{H}(M_v^{i-1}) = \mathsf{sum}(M_v^{i-1})$; $\quad \mathcal{U}(x_v^{i-1}, m_v^i) = \mathsf{relu}(m_v^i)$;
- $\mathcal{G}(x_v^i, m_v^i, P_E(v,w)) = x_v^i \cdot W_i \quad (\forall w \in \mathsf{Nbr}(v))$.

Initially, each x_v^1 is set to the input feature vector $\boldsymbol{v_0}$ of v and $M_v^0 = \emptyset$. At the i-th round, each vertex merges multiple vectors into one by summing the corresponding elements of the vectors in the messages M_v^{i-1}; it then updates its feature vector to $\mathsf{relu}(m_v^i)$. Here the operation relu just resets the negative values in the vector to zero. At last, it computes a

Table 2 Summary of notations

Notation	Definition
$\mathcal{A}, \mathcal{A}_\Delta$	Batch algorithm and incremental algorithm
$G, \Delta G$	Original graph and input updates to G
$\mathcal{H}, \mathcal{U}, \mathcal{G}$	Aggregation, update and propagation function
$\mathcal{U}^-$	The inverse function of $\mathcal{U}$
m_v^i	The aggregated result of the messages sent to v at round i
M_v^{i-1}	The set of messages sent to v at round i
M	A set of messages
$m_{v,w}^{i-1}$	A single message sent from v to w at round i
x_v^i	The state of vertex v at round i
$\mathbb{X}^i, \hat{\mathbb{X}}^i$	The collection of vertex states of G and $G \oplus \Delta G$ at round i
$\mathbb{M}^i, \hat{\mathbb{M}}^i$	The collection of all messages pertaining on G and $G \oplus \Delta G$ at round i
$G_\mathcal{R}$	reserved subgraph, *i.e.,* The subgraph of $G \oplus \Delta G$ induced by the unreset vertices
$G_\mathcal{C}$	Changed graph, *i.e.,* The subgraph of $G \oplus \Delta G$ that induced by the reset vertices
$\mathbb{X}_\mathcal{R}^i, \mathbb{X}_\mathcal{C}^i$	The collection of vertex states of $G_\mathcal{R}$ and $G_\mathcal{C}$ at round i
$\mathbb{M}_\mathcal{R}^i, \mathbb{M}_\mathcal{C}^i$	The collection of messages of $G_\mathcal{R}$ and $G_\mathcal{C}$ at round i

message of $\boldsymbol{v_i} \cdot W_i$, *i.e.,* $x_v^i \cdot W_i$, where $\cdot$ is matrix multiplication and propagates it to each outgoing neighbor w. The computation terminates at the round $K + 1$.

Incremental computation. The problem of incremental graph computation is formalized as follows.

- *Input*: A original graph G, the (old) output $\mathcal{A}(G)$ over G computed by a batch graph algorithm $\mathcal{A}$, and input updates ΔG to G.
- *Output*: The new output $\mathcal{A}(G \oplus \Delta G) = \mathcal{A}(G) \oplus \Delta O$.

Here the input batch update ΔG consists of a set of *unit updates*. To simplify our discussion, we consider the insertion or deletion of a single edge as a unit update in the sequel, which can simulate certain modifications, *e.g.,* updates on edge weight. For instance, each change to the weight on edge $e = (u, v)$ can be considered as deleting e and followed by adding another edge $e' = (u, v)$ with the new weight. Vertex updates are dual of edge updates and can also be readily handled by our proposed approaches. In addition, $G \oplus \Delta G$ denotes applying updates ΔG to G, similarly for $\mathcal{A}(G) \oplus \Delta O$, *i.e.,* ΔO denotes the changes to the old output in response ΔG.

Notations of the paper are summarized in Table 2.

3 Incrementalization framework

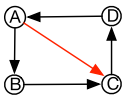
We next present the incrementalization framework underlying Ingress. It aims to directly deduce an incremental graph algorithm $\mathcal{A}_\Delta$ from a given batch vertex-centric algorithm $\mathcal{A}$, without the need of extra user-generated logic or data structures. In a nutshell, the deduced algorithm catches the differences between two runs of its batch counterpart with respect to the messages that should be transmitted. It carries out the corresponding adjustment of old results with the help of an effective memoization strategy.

Message-driven differentiation. In a vertex-centric model, the (final) state of each vertex v is decided by the messages that v receives in different rounds of the iterative computation. Due to this property, we can reduce the problem of finding the differences among two runs of a batch vertex-centric algorithm to identifying the changes to messages. Then for incremental computation, after fetching the messages that differ in one round of the runs over original and updated graphs, it suffices to replay the computation on the affected areas that receive such changed messages, for state adjustment. After that, the changes to the messages are readily obtained for the next round and the algorithm can simply perform the above operations until all changed messages are found and processed. This coincides with the idea of change propagation [1].

To distinguish the differences among messages, we introduce old *invalid messages* and new *missing messages*.

(1) Invalid messages. An old message transmitted during the run over the original graph G is called *invalid* if either its value becomes out-dated for the new graph $G \oplus \Delta G$ or the link for passing the message is disconnected due to input updates ΔG.

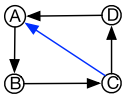
(2) Missing messages. A new message transferred in the run over the $G \oplus \Delta G$ is called *missing* if it is either a revised version of an old message *w.r.t.* G or is associated with a newly added edge in ΔG.



(a). Graph G

Messages	A	B	C	D
Cancellation	$-dx_A^*/2 \to B, C$	$\emptyset$	$-dx_C^* \to D$	$\emptyset$
Compensation	$dx_A^* \to B$	$\emptyset$	$dx_C^*/2 \to A, D$	$\emptyset$

(c). PageRank(x_A^*, x_C^* are PagreRank scores of A and C in G)



(b). $G \oplus \Delta G$

Messages	A	B	C	D
Cancellation	$\perp \to C$	$\emptyset$	$\perp \to D$	$\emptyset$
Compensation	$\emptyset$	$2 \to C$	$\emptyset$	$\emptyset$

(d). SSSP (source = A)

Fig. 2 Cancelation and compensation messages for PageRank and SSSP

Example 2 Consider running the delta-based PageRank algorithm of Example 1(a) on the graph G shown in Fig. 2a. Assume that a batch update ΔG to G removes the edge (A, C) and inserts the edge (C, A); and the resulting graph $G \oplus \Delta G$ is shown in Fig. 2b. Here invalid and missing messages can be identified by inspecting the batch run of PageRank algorithm over G from the perspective of $G \oplus \Delta G$. In the first round, vertex A receives a message $1-d$. It applies propagation function $\mathcal{G}$ and generates two messages, $d(1-d)/2$ and $d(1-d)/2$ for vertices B and C (see Example 1a), respectively. Both are *invalid w.r.t.* $G \oplus \Delta G$. Indeed, in the absence of edge (A, C), vertex A should only send one *missing* message $d(1-d)$ to B. Similarly, in the second round, since A receives $d(1-d)$ from D, it will send two invalid messages $(1-d)d^2/2$ and $(1-d)d^2/2$ to B and C, respectively; and one message to B is missing. In each round of the prior run over G, the insertion of (C, A) also triggers an invalid message from C to D and two missing messages from C to A and D.

As such, the incremental algorithm first discovers all the invalid and missing messages. It then reperforms the computation on affected areas of $G \oplus \Delta G$ by generateing *cancelation messages* (resp. *compensation messages*) to undo (resp. replay) their effects.

It is a common practice to memoize previous computed (intermediate) results in incremental processing [1, 18], similarly for identifying invalid and missing messages.

Memoization policies. A simple memoization strategy for detecting invalid and missing messages is to record *all* the old messages in M_v^i, for each vertex v and $i \geq 0$ in the batch run over G. Then the changed messages can be found by direct comparison between the messages created in the new run and those memoized ones. Guided by the changed messages, the incremental algorithm revises the states of the data iteratively as described above, *i.e.,* canceling (resp. recovering) the effects of invalid messages (resp. missing messages). Here the old states of each vertex can be restored from the stored messages without explicit memoization.

Although this solution is general enough to incrementalize *all* vertex-centric algorithms, it usually causes overwhelming memory overheads [31, 43], especially for algorithms that take a large number of rounds to converge. In light of this, we incorporate a *flexible memoization scheme* in the incrementalization framework to optimize memory usage to different extents whenever possible. The scheme consists of four memoization policies: (1) the *memoization-free policy* (MF) that records no runtime old messages, (2) the *memoization-path policy* (MP) that only records a small part of old messages, (3) the *memoization-vertex policy* (MV) that tracks the states of the vertices among different steps, and (4) the *memoization-edge policy* (ME) that keeps all the old messages.

*(1) Memoization-free (*MF*).* This policy does not record any old message at all. Instead, the incremental algorithms should handle the effects of invalid and missing messages directly on the previous batch run's converged states, *i.e.,* final results. This is doable for a class of vertex-centric algorithms performing *traceable aggregations*, in which the effects of multiple messages can be "assembled" into that of a single message. Moreover, the effects of old *invalid* messages can be "eliminated" by propagating their *inverse* version.

With the MF policy, an incremental algorithm first generates *summarized versions* of cancelation and compensation messages from the previous converged states. They are then processed with the same functions of the batch algorithm, to cancel (resp. compensate) the effects of invalid messages (resp. missing messages).

Example 3 Continuing with Example 2, in order to fix the PageRank scores, we regard the messages received by vertices in each round as "correct with noises" *w.r.t.* $G \oplus \Delta G$. We eliminate these noises by cancelation and compensation messages. For example, for each invalid message value m to B (resp. C), we can send $-m$, the *inverse* of m, as a cancelation message to undo its effect. Similarly, we need to process all the missing messages for B. A key observation is that instead of sending the cancelation (resp. compensation) messages one by one, we can just compute one summarized message to handle the effects of all invalid and missing messages for each vertex. This is because $\mathcal{H}$ and $\mathcal{U}$ of PageRank algorithm (*i.e.,* sum) embrace traceable aggregation. Indeed, in the batch run whenever vertex A *accumulates* a message of value m_i to its state x_v via function $\mathcal{U}$, it generates and sends two invalid messages of value $dm_i/2$ to B and C via function $\mathcal{G}$. Observe that $x_A^* = \mathsf{sum}_i\{m_i\}$, where x_A^* is the converged state of the prior run over G. The aggregation of invalid messages to B and C can be then expressed by $dx_A^*/2$. Thus to cancel the effects of these invalid messages, it suffices to send a summarized *cancelation message* $-dx_A^*/2$ to B (resp. C). The summarized *compensation message* to B can be deduced accordingly, whose value can be expressed by dx_A^*; it is used to enforce the effects of missing messages to B. Edge insertion of (C, A) can be processed along the same lines with cancelation and compensation messages.

All the cancelation and compensation messages are shown in Fig. 2c. The incremental algorithm restarts the computation of PageRank (Example 1a) with these messages. As will be clear in Sect. 4.1, it converges to revised PageRank scores for $G \oplus \Delta G$.

*(2) Memoization-path (*MP*).* This policy only records a *small* portion of old messages that are *effective*. In fact, in some vertex-centric computation with traceable aggregations, the final state x_v relies only on a subset of messages sent to vertex v, which can be referred to as *effective messages* and form a set of *paths*. Take SSSP as an example. The value of the shortest distance *w.r.t.* v is determined by the smallest messages received from the neighbors of v, which lie on the shortest paths from the source vertex. Hence there is no need to handle the effects of invalid messages that are not effective. Under this policy, the incremental algorithms store the paths of effective messages to process invalid messages. The missing messages can be handled as that in memoization-free policy.

Example 4 Recall graph G and input updates ΔG from Example 2 and assume that each edge in G has unit length. Consider running SSSP algorithm of Example 1(b) on G. Observe that the final shortest distance value x_C^* (resp. x_D^*) for vertex C (resp. D) is determined by the message 1 (resp. 2) that sent from A to C and (resp. C to D). Thus these two messages are *effective*. Similarly, there is an effective message 1 from A to B. The incremental algorithm for SSSP stores above three effective messages and works in two phases as follows.

(1) It first cancels the effects of the stored *effecitve* messages that become *invalid* for $G \oplus \Delta G$. Since edge (A, C) is removed, the effective message 1 cannot be passed from A to C and it becomes invalid. Then the incremental algorithm guides A to send a *cancelation message* $\perp$ to C, which indicates the invalidation of the effective message. It resets x_C to the initial state ∞. The cancelation message $\perp$ is further propagated to D, hence x_D is also reset to ∞. At this time, all the effects of invalid effective messages are canceled.

(2) The second phase is to restore the effects of *missing* messages. For each unrest vertex v that either is the source node of an inserted edge or is connected to a reset vertex, the algorithm generates a set of *compensation messages* via function $\mathcal{G}$, in which the converged states x_v^* suffice for the messages propagation purpose (see the definition of $\mathcal{G}$ in Example 1b). These messages are sent to reset vertices and the destination nodes of inserted edges, *i.e.,* a compensation message of value 2 is sent from B to C. That is, the message propagation of the batch algorithm for SSSP resumes with the compensation messages. The computation terminates when the correct revised distance values *w.r.t.* $G \oplus \Delta G$ are obtained.

*(3) Memoization-vertex (*MV*).* The memoization-vertex policy keeps track of the states *w.r.t.* the vertices across different rounds of the batch computation, in a stepwise manner. This is based on the observation that some vertex-centric algorithms directly transfer vertex states as messages. Hence it suffices to memoize the vertex states (aggregated results), from which the invalid and missing messages can be easily discovered in incremental processing. Despite the fact that multiple values will be kept for each vertex, it reduces the space cost from the scale of edges to vertices.

Example 5 With the memoization-vertex policy, an incremental algorithm for GCN-forward can be deduced from the batch algorithm of Example 1(c), by memoizing the aggregated result m_v^i for each vertex v ($v \in V$) at round i ($i \in [1, K+1]$). In particular, m_v^1 is defined as the input feature vector *w.r.t.* v. Given the graph G and updates ΔG of Example 2, the incremental algorithm asks vertex A to send a *cancelation message* $m_{A,C}^1 = -m_A^1 \cdot W_1$ to C in the initial round, to undo the effect of an *invalid* message $m_A^1 \cdot W_1$ transmitted during prior run. This is feasible since GCN-forward takes sum as $\mathcal{H}$. Upon receiving this, vertex C adjusts the cancelation message to $-\mathsf{relu}(m_C^2) \cdot W_2$ and propagates it to D; it also sends a new compensation message $\mathsf{relu}(\mathsf{sum}(m_C^2, m_A^1)) \cdot W_2$ to D. These two represent the difference between the messages transmitted during the two runs. The algorithm also updates m_C^2 to $\mathsf{sum}(m_C^2, m_A^1)$. Analogously, a *compensation message* $m_C^1 \cdot W_1$ is sent from C to A in the first round to enforce the effect of a *missing* message.

Summing up, during round i ($0 < i \leq K$) of the incremental algorithm, for each vertex v that receives messages, a cancelation message $-\mathsf{relu}(m_v^i) \cdot W_i$ and a compensation message $\mathsf{relu}(\mathsf{sum}(m_v^i, M_v^i)) \cdot W_i$ are created and propagated to the neighbors of v. Here M_v^i denotes the set of messages received by v in round i. The recorded m_v^i is also updated to $\mathsf{sum}(m_v^i, M_v^{i-1})$. Finally, computing $\mathsf{relu}(m_v^{K+1})$ can obtain the revised results for $G \oplus \Delta G$.

As observed in [17, 58], such incremental computation of GCN-forward is effective in anomaly detection in dynamic e-commerce graphs and link prediction in evolving social networks, where updated edges refer to new clicks on items and user relationships.

*(4) Memoization-edge (*ME*).* When a batch vertex-centric algorithm cannot be incrementalized with any of the above three policies, the incrementalization should proceed with memoization-edge policy. Here all the old messages of the prior run are memoized for identifying and processing invalid and missing messages. Therefore, the incremental algorithms just simply replay the computation on affected areas that receive evolved messages. With ME policy, we can handle any algorithm in the vertex-centric model of Sect. 2.

Space complexity. It is easy to see that besides the previous final results, the space complexity of the auxiliary information in the incremental algorithms deduced via MF (resp. MP, MV, ME) policy is $O(1)$ (resp. $O(|V|)$, $O(r|V|)$, $O(r|E|)$).

Here r is a variable representing the number of rounds in the batch runs.

Workflow. The workflow of incrementalization includes two parts, *policy selection* and *algorithm builder*.

(a) Policy selection. Given a batch vertex-centric algorithm $\mathcal{A}$, the framework first chooses a memoization policy for incrementalizing $\mathcal{A}$. As will be seen in Sect. 4, there are sufficient conditions for the applicability of different memoization policies so that the decision can be made according the properties of $\mathcal{A}$.

(b) Algorithm builder. The second part is to deduce the incremental algorithm with the selected memoization policy. Based on the sufficient conditions, such an algorithm $\mathcal{A}_\Delta$ can be easily constructed from $\mathcal{A}$ (see Sect. 4).

4 Flexible memoization

We next show how to deduce incremental algorithms $\mathcal{A}_\Delta$ from the given batch ones $\mathcal{A}$ with different memoization policies. To this end, we introduce sufficient conditions for adopting the memoization-free (Sect. 4.1), memoization-path (Sect. 4.2) and memoization-vertex (Sect. 4.3) policies in incrementalizing $\mathcal{A}$, respectively. We formally establish the correctness of the proposed sufficient conditions. We leave out memoization-edge since the incrementalization with this policy is simple and its process has been outlined in Sect. 3.

4.1 Incrementalization via memoization-free

As discussed in Sect. 3, with the memoization-free (MF) policy, the deduced incremental algorithms should initiate two sets of messages, *i.e., cancelation and compensation messages* directly from the converged states of batch runs, which are needed to handle invalid and missing messages, respectively. Intuitively, this is applicable for incrementalizing a class of batch algorithms $\mathcal{A}$, in which (1) the effects of messages can be canceled via their "inverse" form; and (2) the effects of messages can be clearly *traced*. We next formalize these as sufficient conditions for enforcing MF policy.

Conditions. The sufficient condition for the MF policy consists of three sub-conditions.

(1) The first condition says that the update function $\mathcal{U}$ has an *inverse* function $\mathcal{U}^-$ satisfying the following.

$$(\mathbf{C1})\ \mathcal{U}(M \backslash M') = \mathcal{U}(M \cup \{\mathcal{U}^- \circ \mathcal{U}(M')\}) \quad (\forall M' \subseteq M)$$

That is, in order to cancel the effects of a set M' of invalid messages, it suffices to propagate and enforce their inverse $\mathcal{U}^- \circ \mathcal{U}(M')$. Here $\circ$ is a function composition operator such that $\mathcal{U}^- \circ \mathcal{U}$ denotes applying function $\mathcal{U}$ followed by function $\mathcal{U}^-$.

(2) The other two conditions ensure that the effects of messages can be clearly *traced* across multiple iterations. That is, the effect of an invalid message $m^i_{u,v}$ to v sent in round $i+1$ can be traced from the vertex state x^j_v for any later round $j > i$. Combining this invariant with (**C1**), we can cancel the effects of invalid messages and compensate the missing messages without memoizing intermediate states. It is obvious that the aggregation function $\mathcal{H}$ and the update function $\mathcal{U}$ should be identical (*i.e.,* $\mathcal{H} = \mathcal{U}$); otherwise the traceability no longer exists due to the update function. An algorithm $\mathcal{A}$ with traceability should have the following properties.

$$(\mathbf{C2})\ \mathcal{U}(\{\mathcal{U}(M)\} \cup M') = \mathcal{U}(M \cup M')$$
$$(\mathbf{C3})\ \mathcal{U} \circ \mathcal{G} \circ \mathcal{U}(M) = \mathcal{U} \circ \mathcal{G}(M)$$

Intuitively, condition (**C2**) enables partial aggregation for function $\mathcal{U}$, so that we can directly measure the effects of partially aggregated messages (or even a single message). If condition (**C3**) holds, the embedded aggregations within the iterations can be "picked out" without affecting the result, *e.g.,* $\mathcal{U} \circ \mathcal{G} \circ \mathcal{U} \circ \mathcal{G} \circ \mathcal{U}(M) = \mathcal{U} \circ \mathcal{G} \circ \mathcal{G}(M)$. The condition (**C3**) also states that function $\mathcal{G}$ generates messages solely based on the input aggregated results and edge properties, without considering the vertex states. This is because it does not require applying function $\mathcal{U}$ as the prerequisite. Thus, here we use $\mathcal{G}(M)$ in (**C3**) instead of $\mathcal{G}(x_v, m_v, P_E(v, w))$. By conditions (**C2**) and (**C3**), the state of each vertex is the aggregation of all messages accumulated so far, *i.e.,* $x^t_v = \mathcal{U}(\bigcup_{i=0}^{t}\{m^i_v\})$. With this traceability, we do not store any intermediate vertex states.

We say that a vertex-centric batch algorithm $\mathcal{A}$ is MF-*applicable* if it satisfies the conditions (**C1**)(**C2**)(**C3**).

Example 6 Since $\mathsf{sum}(M \backslash M') = \mathsf{sum}(M, \{-\mathsf{sum}(M')\})$, where M (resp. M') consists of real numbers, we know that the PageRank algorithm of Example 1(a) satisfies (**C1**) and $\mathcal{U}^-$ computes the negative value of the input. The other two conditions also hold as function sum is associative and $\mathsf{sum}(d \times M_1/N_v, d \times M_2/N_v) = \mathsf{sum}(d \times \mathsf{sum}(M_1, M_2)/N_v)$, *i.e.,* $\mathcal{U} \circ \mathcal{G} \circ \mathcal{U}(M) = \mathcal{U} \circ \mathcal{G}(M)$.

We next show how to deduce the necessary messages for an MP-applicable algorithm $\mathcal{A}$.

Deducing messages. Suppose that G is updated with input changes ΔG. For each vertex v, we deduce a set M^-_v of cancelation messages and a set M^+_v of compensation messages as follows.

(1) Cancelation messages. Denote by $w_1, \ldots, w_k$ the neighbors of v in G. Given graph updates ΔG, the old message m^i_{v,w_j} sent from v to w_j could become invalid. This happens when $\mathcal{G}(x^i_v, m^i_v, P_E(v, w_j)) \neq \mathcal{G}(x'^i_v, m'^i_v, P'_E(v, w_j))$, where x'^i_v, m'^i_v and $P'_E(w, w_j)$ are the vertex state, aggregated result and edge property *w.r.t.* the new run over $G \oplus \Delta G$,

respectively. In this case, we call (v, w_j) is an *evolved edge* for transmitting messages. To eliminate the effects of invalid messages, we create a set M_v^- of *cancelation messages* as

$$M_v^- = \{\mathcal{F}(m_v^*, P_E(v, w_j)) \mid \text{evolved } (v, w_j) \text{ in } G\}. \quad (2)$$

Here m_v^* is the aggregation of initial and all received messages in the bath run over G and $\mathcal{F}$ is defined as the composition $\mathcal{U}^- \circ \mathcal{U} \circ \mathcal{G}$. In fact, all the messages propagated from v to w_j are $M_{v,w_j} = \bigcup_{i=0}^{\infty} \mathcal{G}(x_v^i, m_v^i, P_E(v, w_j)) = \bigcup_{i=0}^{\infty} \mathcal{G}(*, m_v^i, P_E(v, w_j))$, as the message generation does not depend on the vertex state x_v^i. Let all the messages received by w_j across iterations be $\bigcup_{i=0}^{\infty} M_{w_j}^i = M_{v,w_j} \cup M'$, where M' represents the messages from w_j's other incoming neighbors. Observe that $m_{w_j}^* = \mathcal{U}(M_{v,w_j} \cup M')$. Then removing the effects of messages M_{v,w_j} is equivalent to updating $x_{w_j}^*$ to $\mathcal{U}(M')$. By condition (**C1**), we have that $\mathcal{U}(\bigcup_{i=0}^{\infty} M_{w_j}^i \cup \{\mathcal{U}^- \circ \mathcal{U}(M_{v,w_j})\}) = \mathcal{U}(M')$. Thus it suffices to propagate $\mathcal{U}^- \circ \mathcal{U}(M_{v,w_j})$ from v to w_j. According to condition (**C2**) we have $\mathcal{U}(M_{v,w_j}) = \mathcal{U}(\bigcup_{i=0}^{\infty} \mathcal{G}(*, m_v^i, P_E(v, w_j)))$. Observe that $m_v^* = \mathcal{U}(\bigcup_{i=0}^{\infty} m_v^i)$, then by condition (**C3**) we have $\mathcal{U}(M_{v,w_j}) = \mathcal{U} \circ \mathcal{G}(*, m_v^*, P_E(v, w_j))$. It follows that the cancelation message sent to w_j can be expressed as $\mathcal{U}^- \circ \mathcal{U} \circ \mathcal{G}(*, m_v^*, P_E(v, w_j))$. However, we do not record m_v^*, we can deduce m_v^* by $m_v^* = \mathcal{U}^-(x_v^*, x_v^0)$ because $x_v^* = \mathcal{U}(x_v^0 \cup m_v^*)$.
(2) Compensation messages. The set M_v^+ of compensation messages can be computed as the dual of cancelation messages M_v^-. They will be utilized to enforce the effects of missing messages passed via evolved edges. More specifically, we derive the compensation messages M_v^+ by using the new edge properties *w.r.t.* $G \oplus \Delta G$ as

$$M_v^+ = \left\{ \mathcal{U} \circ \mathcal{G}(*, m_v^*, P_E'(v, w_j)) \;\middle|\; \begin{array}{l}\text{evolved } (v, w_j) \\ \text{in } G \oplus \Delta G\end{array} \right\}. \quad (3)$$

As discussed above, this is needed if there exist differences between the messages sent from v during the runs over G and $G \oplus \Delta G$. That is, $\mathcal{G}(*, m_v^*, P_E(v, w_j)) \neq \mathcal{G}(*, m_v^*, P_E'(v, w_j))$ for neighbor w_j of v.

We are now ready to show how to incrementalize a vertex-centric algorithm $\mathcal{A}$ that is MF-applicable.

Incremental algorithm. Given a graph G, input updates ΔG to G and the previous result $\{x_v^*\}_{v \in V}$ derived by an MF-applicable batch algorithm $\mathcal{A}$ over G, the deduced incremental Algorithm 1 computes the updated results for $G \oplus \Delta G$. It first finds those evolved edges (v, w_j) induced by input updates, *i.e.,* ΔG triggers invalid or missing messages (line 1). This is achieved by comparing the messages that directly created with the previous converged states as described above. For each evolved edge, it then initiates appropriate cancelation and compensation messages based on Eqs (2) and (3) (lines 2–5). Starting with the transmission of these messages to designated neighbors, it restores the iterative computation of $\mathcal{A}$ over $G \oplus \Delta G$ to get the updated results, *i.e.,* applying the same functions $\mathcal{H}, \mathcal{U}$ and $\mathcal{G}$ as batch counterpart $\mathcal{A}$ (line 6).

Algorithm 1: Incrementalization via MF policy

Input: Graph G, graph updates ΔG, previous computation result $\{x_v^*\}_{v \in V}$ of $\mathcal{A}$ *w.r.t.* G.
Output: Updated result $\{x_v'\}_{v \in V}$ *w.r.t.* $G \oplus \Delta G$.

1 find all evolved edges induced by ΔG;
2 **foreach** evolved edge (v, w_j) **do**
3 $\quad m_v^* = \mathcal{U}^-(x_v^*, x_v^0)$;
4 $\quad M_v^- \leftarrow M_v^- \cup \{\mathcal{U}^- \circ \mathcal{U} \circ \mathcal{G}(*, m_v^*, P_E(v, w_j))\}$;
5 $\quad M_v^+ \leftarrow M_v^+ \cup \{\mathcal{U} \circ \mathcal{G}(*, m_v^*, P_E'(v, w_j))\}$;
6 restore computation with messages M_v^-, M_v^+ $(\forall v \in V)$.

Example 7 Continuing with Example 3, we use Algorithm 1 to generate the cancelation and compensation messages as shown in Fig. 2c. Observe that both A and C pertain to evolved edges. We first apply $\mathcal{U}^- \circ \mathcal{U} \circ \mathcal{G}$ over G and generate two cancelation messages in M_A^-, one for B and one for C. Based on the definitions of $\mathcal{U}$ and $\mathcal{G}$ for PageRank (see Example 1), both messages are $-dx_A^*/2$. For M_A^+, we apply $\mathcal{U} \circ \mathcal{G}$ over $G \oplus \Delta G$ to generate a compensation message to B in M_A^+ with value dx_A^*. The messages M_C^- and M_C^+ can be computed similarly (see Fig. 2c).

Remark (1) To incrementalize MF-applicable algorithms like PageRank, prior systems such as GraphBolt [31] and DZiG [30] proposed optimizations to reduce space cost for intermediate results to some extent. With the MF policy, Ingress is the first system to incrementalize these algorithms without recording any intermediate results.
(2) Intuitively, MF-applicable algorithms like PageRank can derive cancelation and compensation messages without recording any intermediate vertex states due to two key factors (a) these algorithms employ accumulative aggregation algorithms, and (b) there exists an inverse function capable of eliminating the effects of previously accumulated messages from the vertex states. Apart from PageRank, many other algorithms are also MF-applicable, such as SimRank [20], Penalized Hitting Probability (PHP) [16], Katz Metric [22], Believe Propagation [39] and Adsorption [2], *i.e.,* they can be incrementalized with Algorithm 1.

Algorithm correctness. The correctness of incrementalization via MF policy is warranted by Theorem 1.

Theorem 1 (Correctness of MF policy) *The computation of* MF*-applicable* $\mathcal{A}$ *restored with messages* (M_v^-, M_v^+) *converges to the correct result* $\mathcal{A}(G \oplus \Delta G)$.

We first introduce the following notations to facilitate the proof of Theorem 1. Denote by (i) $\mathbb{X}^i = \{x_v^i\}_{v \in V}$ the collection of vertex states in the i-the round of computation; and

denote by $\mathbb{M}^i = \{M_v^i\}_{v \in V}$ the collection of messages used in the i-the round of computation. Specifically, $\mathbb{X}^0$ and $\mathbb{M}^0$ are the initial vertex states and messages. Observe that $\mathcal{H} = \mathcal{U}$ as required for an MF-applicable algorithm $\mathcal{A}$. Thus the aggregation function $\mathcal{H}$ and the update function $\mathcal{U}$ are combined into one function $\mathcal{U}$. With these notations, we can rewrite the vertex-centric computation of an MF-applicable algorithm $\mathcal{A}$ as follows:

$$\mathbb{X}^i = \mathcal{U}(\mathbb{X}^{i-1} \cup \mathbb{M}^{i-1}), \qquad \mathbb{M}^i = \mathcal{G}(\mathbb{M}^{i-1}). \tag{4}$$

Here we abuse the notation $\mathcal{U}$ to allow it to take a collection of vertex states and messages as input, which should be considered as a group-by aggregation operation over a collection of vertex states and messages. That is, $\mathcal{U}(\mathbb{X}^{i-1}, \mathbb{M}^{i-1})$ should be interpreted as applying $\mathcal{U}$ over x_v^{i-1} and M_v^{i-1} for each vertex v in V. Similarly, $\mathcal{G}$ in Eq. (4) should also be treated as a group-by operation. Intuitively, Eq (4) describes a global view of the computation for an MP-applicable algorithm $\mathcal{A}$. Starting from $(\mathbb{X}^0, \mathbb{M}^0)$, $\mathcal{A}$ iteratively updates $\mathbb{X}^i$ and generates new messages $\mathbb{M}^i$ to get the final result. We also write $\hat{\mathbb{X}}^i$ and $\hat{\mathbb{M}}^i$ for the vertex states and messages in the incremental computation over the updated graph $G \oplus \Delta G$, to distinguish them from those in the computation over the original graph G.

We prove Theorem 1 in three steps as shown below.
(1) We first characterize $\mathbb{X}^k$ in terms of the initial states $\mathbb{X}^0$ and messages $\mathbb{M}^i$ ($i \geq 0$) propagated during the computation of algorithm $\mathcal{A}$ (Lemma 1).
(2) We next analyze the initial vertex states $\hat{\mathbb{X}}^0$ and the initial messages $\hat{\mathbb{M}}^0$ generated by Algorithm 1 for the incremental computation of $\mathcal{A}$ over $G \oplus \Delta G$ (Lemma 2).
(3) With Lemma 1 and Lemma 2, we show that the computation over $G \oplus \Delta G$ starting from $(\hat{\mathbb{X}}^0, \hat{\mathbb{M}}^0)$ converges to the same result as the computation starting from $(\mathbb{X}^0, \mathbb{M}^0)$ over $G \oplus \Delta G$. Thus Theorem 1 follows.

Lemma 1 *Suppose that an* MF*-applicable algorithm $\mathcal{A}$ starts from $\mathbb{X}^0$ and $\mathbb{M}^0$, then the computation result on graph G after k rounds can be represented as $\mathbb{X}^k = \mathcal{U}\left(\mathbb{X}^0 \cup \bigcup_{i=0}^{k-1} \mathcal{G}^i(\mathbb{M}^0)\right)$, where $\mathcal{G}^\ell$ denotes ℓ times of applications of the function $\mathcal{G}$ and $\mathcal{G}^0(\mathbb{M}^0) = \mathbb{M}^0$.*

Lemma 2 *Suppose that the computation of an* MF*-applicable $\mathcal{A}$ over G converges after k iterations. The initial states $\hat{\mathbb{X}}^0$ and messages $\hat{\mathbb{M}}^0$ in the incremental computation of Algorithm 1 can be expressed as follows.*

$$\hat{\mathbb{X}}^0 = \mathcal{U}\left(\mathbb{X}^0 \cup \bigcup_{i=0}^{k-1} \mathcal{G}^i(\mathbb{M}^0)\right), \tag{5}$$

$$\hat{\mathbb{M}}^0 = \mathcal{U}\left(\hat{\mathcal{G}}\left(\bigcup_{i=0}^{k-1} \mathcal{G}^i(\mathbb{M}^0)\right) \ominus \mathcal{G}\left(\bigcup_{i=0}^{k-1} \mathcal{G}^i(\mathbb{M}^0)\right)\right). \tag{6}$$

Here we use (i) $\hat{\mathcal{G}}$ to represent the propagation function over the updated graph $G \oplus \Delta G$, to distinguish from the function $\mathcal{G}$ over G, and (ii) $\ominus$ to denote the applications of union $\cup$, inverse function $\mathcal{U}^-$ and aggregation function $\mathcal{U}$ for simplicity, i.e., $A \ominus B = A \cup \mathcal{U}^- \circ \mathcal{U}(B)$.

We are ready to prove Theorem 1.

Proof of Theorem 1 We show that on $G \oplus \Delta G$ the computation starting from $(\mathbb{X}^0, \mathbb{M}^0)$ converges to the same result as the incremental computation starting from $(\hat{\mathbb{X}}^0, \hat{\mathbb{M}}^0)$. Here $\hat{\mathbb{X}}^0$ and $\hat{\mathbb{M}}^0$ are the initial states and messages as stated in Lemma 2. Assume *w.l.o.g.* the computations of $\mathcal{A}$ on G and $G \oplus \Delta G$ both converge after k iterations. It suffices to establish the following.

$$\mathcal{U}\left(\hat{\mathbb{X}}^0 \cup \bigcup_{i=0}^{k-1} \hat{\mathcal{G}}^i(\hat{\mathbb{M}}^0)\right) = \mathcal{U}\left(\mathbb{X}^0 \cup \bigcup_{i=0}^{k-1} \hat{\mathcal{G}}^i(\mathbb{M}^0)\right) \tag{7}$$

By Lemma 1, starting from $(\hat{\mathbb{X}}^0, \hat{\mathbb{M}}^0)$, the computation result $\hat{\mathbb{X}}^k$ over $G \oplus \Delta G$ after k iterations can be computed as follows.

$$\begin{aligned}
&\mathcal{U}\left(\hat{\mathbb{X}}^0 \cup \bigcup_{i=0}^{k-1} \hat{\mathcal{G}}^i(\hat{\mathbb{M}}^0)\right) && \text{(a)}\\
&= \mathcal{U}\left(\hat{\mathbb{X}}^0 \cup \bigcup_{i=1}^{k} \hat{\mathcal{G}}^i\left(\bigcup_{j=0}^{k-1} \mathcal{G}^j(\mathbb{M}^0)\right)\right.\\
&\qquad \left.\ominus \bigcup_{i=0}^{k-1} \hat{\mathcal{G}}^i\left(\bigcup_{j=1}^{k} \mathcal{G}^j(\mathbb{M}^0)\right)\right) && \text{(b)}\\
&= \mathcal{U}\left(\hat{\mathbb{X}}^0 \cup \hat{\mathcal{G}}^k(\mathbb{M}^0) \cup \bigcup_{i=1}^{k-1} \hat{\mathcal{G}}^i(\mathbb{M}^0) \cup \bigcup_{j=0}^{k-1} \hat{\mathcal{G}}^k(\mathcal{G}^j(\mathbb{M}^0))\right.\\
&\qquad \left.\ominus \mathcal{G}^k(\mathbb{M}^0) \cup \bigcup_{j=1}^{k-1} \mathcal{G}^j(\mathbb{M}^0) \cup \bigcup_{i=0}^{k-1} \hat{\mathcal{G}}^i(\mathcal{G}^k(\mathbb{M}^0))\right) && \text{(c)}\\
&= \mathcal{U}\left(\hat{\mathbb{X}}^0 \cup \bigcup_{i=1}^{k-1} \hat{\mathcal{G}}^i(\mathbb{M}^0) \ominus \bigcup_{j=1}^{k-1} \mathcal{G}^j(\mathbb{M}^0)\right) && \text{(d)}\\
&= \mathcal{U}\left(\mathbb{X}^0 \cup \bigcup_{i=0}^{k-1} \hat{\mathcal{G}}^i(\mathbb{M}^0)\right) && \text{(e)}
\end{aligned}$$

Line (a) is due to Lemma 1, while line (b) is due to Lemma 2 and condition (**C2**). After unfolding the second union item and the third union item in line (b) we get line (c). This is because most of the unfolded items are the same and can be canceled out. Since $\mathcal{A}$ on G and $G \oplus \Delta G$ both converge after k iterations (*i.e.,* $\mathbb{X}^{k+1} = \mathbb{X}^k$ and $\hat{\mathbb{X}}^{k+1} = \hat{\mathbb{X}}^k$), we have that $\mathcal{G}^k(*) = \mathbf{0}$ and $\hat{\mathcal{G}}^k(*) = \mathbf{0}$ where $\mathbf{0}$ is the identity element of $\mathcal{U}$, *i.e.,* $\mathcal{U}(\mathbb{X} \cup \mathbf{0}) = \mathcal{U}(\mathbb{X})$ for any $\mathbb{X}$. Thus

Algorithm 2: Incrementalization via MP policy

Input: Graph G, graph updates ΔG, result $\{x_v^*\}_{v\in V}$ of $\mathcal{A}$ on G, and effective messages M_E.
Output: Updated $\{x_v'\}_{v\in V}$ *w.r.t.* $G \oplus \Delta G$.

1 **foreach** $m_c \in M_E$ sent via a deleted $(v, w) \in \Delta G$ **do**
2 initiate a cancelation message $\perp$ to be sent from vertex v to vertex w;
3 propagate $\perp$ along the paths formed by M_E and reset x_w to initial state for the receivers w of $\perp$;
4 **foreach** (v, w_j) that is evolved or is associated with a reset vertex **do**
5 $M_v^+ \leftarrow M_v^+ \cup \{\mathcal{U}\circ\mathcal{G}(*, x_v^*, P_E'(v, w_j))\}$;
6 restore the computation with messages M_v^+ $(\forall v \in V)$.

by ignoring the identity elements $\mathcal{G}^k(\mathbb{M}^0) = \hat{\mathcal{G}}^k(\mathbb{M}^0) = \bigcup_{i=0}^{k-1} \hat{\mathcal{G}}^k(\mathcal{G}^i(\mathbb{M}^0)) = \bigcup_{i=1}^{k} \hat{\mathcal{G}}^i(\mathcal{G}^k(\mathbb{M}^0)) = \mathbf{0}$, we obtain line (d) from line (c). Line (e) then follows from Eq. (5). □

4.2 Incrementalization via memoization-path

When the inverse function $\mathcal{U}^-$ required by the condition (**C1**) of MF-applicability is hard to find, one might be tempted to store the whole set of intermediate results and messages for removing invalid messages. However, not all is lost. Despite condition (**C1**), there are vertex-centric algorithms in which only part of the messages decide the final results. To this end, it suffices to consider the cancelation of those invalid messages that have impacts on the converged states. Putting this and the properties of traceability together, it is feasible to incrementalize another class of algorithms $\mathcal{A}$ via the memoization-path (MP) policy, where a small portion of the old effective messages are memoized. Since we still need traceability, the aggregation function $\mathcal{H}$ and update function $\mathcal{U}$ of batch algorithm $\mathcal{A}$ should be identical.

Conditions. The sufficient condition for applying MP policy in incrementalizaion also consists of two parts.

(1) The aggregation, *i.e.,* update function $\mathcal{U}$ in batch algorithm $\mathcal{A}$ selects as output a *single* element from the input set. That is,

$$(\mathbf{C4})\ \mathcal{U}(M) = m_c \in M.$$

The condition (**C4**) requires the existence of a specific input message m_c, referred to as an *effective message*.

(2) The vertex-centric algorithm $\mathcal{A}$ is endowed with the traceability property, *i.e.,* it satisfies conditions (**C2**) and (**C3**) of Sect. 4.1.

Intuitively, condition (**C4**) requires the output of $\mathcal{U}$ only depends on a single input message. Combining with (**C2**) and (**C3**), it implies a tree structure for the effective messages transferred between vertices. The traceroutes (or paths) of the effective messages can be clearly captured in the batch run. We say that a batch vertex-centric algorithm $\mathcal{A}$ is MP-*applicable* if $\mathcal{A}$ satisfies conditions (**C2**)(**C3**)(**C4**).

Example 8 The SSSP algorithm of Example 1(b) is MP-applicable. Obviously, the function min (*i.e.,* function $\mathcal{U}$) selects a single minimum value from the input sets, hence (**C4**) is satisfied. It also satisfies (**C2**)–(**C3**) as the computation of minimum distance values can be postponed until all messages are transmitted and accumulated.

Deducing messages. Similar to the MF policy, the cancelation and compensation messages are deduced under the MP policy, in response to the effects of invalid and missing messages. The difference is that we explicitly store all the effective messages after the batch run over G with MP policy, which form a set of *paths*.

(1) Cancelation messages. Each cancelation message, denoted as $\perp$, is initiated in regard to an effective message m_c whose transmitting route is broken due to the input updates ΔG. Intuitively, if an effective m_c was sent from vertex v to w during the batch run and edge (v, w) is deleted in ΔG, then it becomes invalid.

(2) Compensation messages. The compensation messages are derived along the same lines as that in the MF policy (Sect. 4.1), which will be propagated to enforce the effects of missing messages. The only difference is the senders of these messages (see below).

Incremental algorithm. The procedure for incrementalizing an MP-applicable algorithm $\mathcal{A}$ is shown as Algorithm 2. It consists of two phases. In the first phase (lines 1–3), it propagates cancelation messages $\perp$ along the paths that formed by the stored effective messages of the batch run. This process starts with deleted edges that have been used to transmit effective messages (lines 1–2), and cancels the effects of invalid effective messages by resetting states to initial version (line 3). After that, the second phase initiates compensation messages M_v^+ using the same strategy of Algorithm 1 (lines 4-5). Note that compensation messages are also generated at reset vertices v or those linked to reset vertices w_j, *i.e.,* v or w_j has been reset. They will be sent to w_j to adjust the states from the initial version and (v, w_j) can be regarded as an evolved edge. Finally, the iterative computation of $\mathcal{A}$ continues with M_v^+ (line 6).

One can verify that the logic of the incremental SSSP algorithm described in Example 4 exactly coincides with that of Algorithm 2. There also exist other MP-applicable algorithms, *e.g.,* Connected Components [3] and Lowest Common Ancestor [42].

Remark Note that in Algorithm 2, before restoring the iterative computation of $\mathcal{A}$ (line 6), Ingress first needs to roll back the vertex states to a safe approximation *w.r.t.* $G \oplus \Delta G$. As we have seen, Ingress takes a dependency guided reset-

ting approach for rollback[1], *i.e.,* it propagates "$\perp$" along the paths formed effective messages and resets the vertex states to their initial versions during the process. Other systems, like KickStarter, use a different rollback mechanism. Instead of resetting the states of affected vertices to the initial versions directly, they compute safe approximations based on *trimming* [50]. This is achieved by utilizing both (i) additional level information in value dependency and (ii) the neighbor states of affected vertices. We remark the following.

(1) Trimming-based rollback is more effective in reducing the cost of iterative computation after rollback. However, it also brings additional overheads since it involves more sophisticated computation logic and requires to maintain extra data structures such as the dependency level information. Thus, trimming-based rollback is not suitable for the case where a large portion of vertices are affected by the graph updates as the trimming cost may dominate the overall incremental processing cost.
(2) Resetting-based rollback is more efficient in identifying the vertices affected by graph updates. However, it may reset the states of more vertices to their initial versions compared to the trimming-based approach [50]. Nevertheless, we find that achieving a similar effect to trimming requires only a very small number of rounds of iterative computation after resetting, typically 1–2 rounds (see Sect. 7.7). When the number of rounds of iterative computation after rollback is large, adding a small number of rounds has only a minor impact. Therefore, resetting-based rollback is more suitable for cases where the number of affected vertices and that of rounds of iterative computation are both very large.

Algorithm correctness. The correctness of Algorithm 2 can be verified by the following theorem.

Theorem 2 (Correctness of MP policy) *After propagating cancelation messages $\perp$, the iterative computation of an* MP*-applicable algorithm $\mathcal{A}$ restored with messages M_v^+ converges to the correct result $\mathcal{A}(G \oplus \Delta G)$.*

Since an MP-applicable algorithm $\mathcal{A}$ satisfies conditions (**C2**) and (**C3**), the computation of $\mathcal{A}$ can also be characterized by Eq. 4. We thus borrow the notations $\mathbb{X}^i$, $\mathbb{M}^i$, $\hat{\mathbb{X}}^i$, $\hat{\mathbb{M}}^i$ from the proof of Theorem 1.

In addition, we define a *reserved subgraph* $G_{\mathcal{R}} = (V_{\mathcal{R}}, E_{\mathcal{R}}, P_{G_{\mathcal{R}}})$. Here $V_{\mathcal{R}}$ denotes the set of *unreset vertices*, which are *not reset* with the cancelation messages $\perp$ in Algorithm 2 and hence are not the receivers of $\perp$; $E_{\mathcal{R}} = \{e \mid e \in (V_{\mathcal{R}} \times V_{\mathcal{R}}) \cap E \cap E'\}$, where E' denotes the edge set of the updated graph $G' = G \oplus \Delta G = (V', E', P_{G'})$; and $P_{G_{\mathcal{R}}}$ is defined accordingly with $V_{\mathcal{R}}$ and $E_{\mathcal{R}}$. We also write $\mathbb{X}^i_{\mathcal{R}}$, $\mathbb{M}^i_{\mathcal{R}}$ and $\mathcal{G}_{\mathcal{R}}$ to represent the collections of vertex states and messages in round i when running algorithm $\mathcal{A}$ over the reserved graph $G_{\mathcal{R}}$, and the propagation function over $G_{\mathcal{R}}$, respectively. Intuitively, $G_{\mathcal{R}}$ is a subgraph of the intersection of G and $G \oplus \Delta G$. It includes all the vertices that have not been reset, which means the result of the batch run over G is reserved in $G_{\mathcal{R}}$.

We define *changed graph* $G_{\mathcal{C}} = (V_{\mathcal{C}}, E_{\mathcal{C}}, P_{G_{\mathcal{C}}})$ in a similar fashion, where $V_{\mathcal{C}} = V' \backslash V_{\mathcal{R}}$, $E_{\mathcal{C}} = \{e \mid e \in (V_{\mathcal{C}} \times V_{\mathcal{C}}) \cap E'\}$ and $P_{G_{\mathcal{C}}}$ is specified accordingly. We let $\mathbb{X}^i_{\mathcal{C}}$ and $\mathbb{M}^i_{\mathcal{C}}$ denote the collections of vertex states and messages in round i when invoking $\mathcal{A}$ on $G_{\mathcal{C}}$, respectively. In fact, the changed subgraph contains all reset vertices whose states are not yet determined in Algorithm 2 after transmitting cancelation messages $\perp$.

The proof of theorem 2 also consists of three steps.
(1) We first show that after the propagation of cancelation messages $\perp$ in Algorithm 2, the converged states of those vertices that have not been reset coincide with the ones obtained by directly running batch algorithm $\mathcal{A}$ over the reserved subgraph $G_{\mathcal{R}}$ (Lemma 3).
(2) Next, we investigate the components of the initial vertex states $\hat{\mathbb{X}}^0$ and (compensation) messages $\hat{\mathbb{M}}^0$ created in Algorithm 2, and characterize them in terms of the initial states $\mathbb{X}^0_{\mathcal{R}}$ and $\mathbb{X}^0_{\mathcal{C}}$, and messages $\mathbb{M}^0_{\mathcal{R}}$ and $\mathbb{M}^0_{\mathcal{C}}$ *w.r.t.* the reserved subgraph $G_{\mathcal{R}}$ and changed subgraph $G_{\mathcal{C}}$ (Lemma 4).
(3) With Lemma 3 and 4, we then prove that the computation of $\mathcal{A}$ restored with $\hat{\mathbb{X}}^0$ and $\hat{\mathbb{M}}^0$ converges to the same result as running $\mathcal{A}$ over $G \oplus \Delta G$ (Theorem 2).

Lemma 3 *Suppose that $\mathcal{A}$ converges after k rounds over G. Then after propagating cancelation messages $\perp$ along the paths formed by effective messages M_E, the states of the unreset vertices, denoted by $\mathbb{X}^k_N$, can be expressed as*

$$\mathbb{X}^k_N = \mathcal{U}\Big(\mathbb{X}^0_{\mathcal{R}} \cup \bigcup_{\ell=0}^{k-1} \mathcal{G}^{\ell}_{\mathcal{R}}(\mathbb{M}^0_{\mathcal{R}})\Big). \tag{8}$$

Lemma 4 *After propagating cancelation messages $\perp$, the initial vertex states $\hat{\mathbb{X}}^0$ and compensation messages $\hat{\mathbb{M}}^0$ in the incremental computation of Algorithm 2 can be expressed as follows.*

$$\hat{\mathbb{X}}^0 = \mathcal{U}\left(\mathbb{X}^0_{\mathcal{R}} \cup \bigcup_{\ell=0}^{k} \mathbb{M}^{\ell}_{\mathcal{R}}\right) \cup \mathbb{X}^0_{\mathcal{C}}, \tag{9}$$

$$\hat{\mathbb{M}}^0 = \mathbb{M}^0_{\mathcal{C}} \cup \bigcup_{\ell=0}^{k} \left(\mathcal{U} \circ \hat{\mathcal{G}}(\mathbb{M}^{\ell}_{\mathcal{R}}) \setminus \mathcal{U} \circ \mathcal{G}_{\mathcal{R}}(\mathbb{M}^{\ell}_{\mathcal{R}})\right). \tag{10}$$

Here $\hat{\mathcal{G}}$ represents the propagation function w.r.t. *$G \oplus \Delta G$ and the batch run of $\mathcal{A}$ terminates in k rounds.*

Proof of Theorem 2 We now verify that the computation of $\mathcal{A}$ restored with $(\hat{\mathbb{X}}^0, \hat{\mathbb{M}}^0)$ converges to the same result as

[1] This is essentially the VAD-Reset approach in [50].

running $\mathcal{A}$ over $G \oplus \Delta G$ with initial $(\mathbb{X}^0, \mathbb{M}^0)$. Since the MP-applicable algorithm $\mathcal{A}$ satisfies conditions (**C2**) and (**C3**), Lemma 1 still holds for the MP-applicable $\mathcal{A}$. Therefore, it suffices to show

$$\mathcal{U}\left(\hat{\mathbb{X}}^0 \cup \bigcup_{\ell=0}^{k-1} \hat{\mathcal{G}}^{\ell}(\hat{\mathbb{M}}^0)\right) = \mathcal{U}\Big(\mathbb{X}^0 \cup \bigcup_{\ell=0}^{k-1} \hat{\mathcal{G}}^{\ell}(\mathbb{M}^0)\Big). \tag{11}$$

Similar to the proof of Theorem 1, here we assume *w.l.o.g.* the computations of $\mathcal{A}$ on G and $G \oplus \Delta G$ both converge after k rounds. With the expressions of $\hat{\mathbb{X}}^0$ and $\hat{\mathbb{M}}^0$ given in Lemma 4, the computation result $\hat{\mathbb{X}}^k$ of $\mathcal{A}$ on $G \oplus \Delta G$ after k rounds can be expressed as

$$\begin{aligned}
&\mathcal{U}\left(\hat{\mathbb{X}}^0 \cup \bigcup_{\ell=0}^{k-1} \hat{\mathcal{G}}^{\ell}(\hat{\mathbb{M}}^0)\right) && \text{(a)}\\
&= \mathcal{U}\left(\bigcup_{\ell=0}^{k-1} \hat{\mathcal{G}}^{\ell}\left(\mathbb{M}_{\mathcal{C}}^0 \cup \bigcup_{\ell=0}^{k}\Big(\mathcal{U}\circ\hat{\mathcal{G}}(\mathbb{M}_{\mathcal{R}}^{\ell}) \setminus \mathcal{U}\circ\mathcal{G}_{\mathcal{R}}(\mathbb{M}_{\mathcal{R}}^{\ell})\Big)\right)\right.\\
&\quad \left.\cup\mathcal{U}\left(\mathbb{X}_{\mathcal{R}}^0 \cup \bigcup_{\ell=0}^{k}\mathbb{M}_{\mathcal{R}}^{\ell}\right) \cup \mathbb{X}_{\mathcal{C}}^0\right) && \text{(b)}\\
&= \mathcal{U}\left(\bigcup_{\ell=1}^{k} \hat{\mathcal{G}}^{k-\ell}\left(\bigcup_{\ell'=\ell}^{k}\Big(\mathcal{U}\circ\hat{\mathcal{G}}(\mathbb{M}_{\mathcal{R}}^{\ell'}) \setminus \mathcal{U}\circ\mathcal{G}_{\mathcal{R}}(\mathbb{M}_{\mathcal{R}}^{\ell'})\Big)\right)\right.\\
&\quad \left.\cup\mathbb{X}^0 \cup \bigcup_{\ell=0}^{k-1} \hat{\mathcal{G}}^{\ell}(\mathbb{M}^0)\right) && \text{(c)}\\
&= \mathcal{U}\left(\mathbb{X}^0 \cup \bigcup_{\ell=0}^{k-1} \hat{\mathcal{G}}^{\ell}(\mathbb{M}^0)\right). && \text{(d)}
\end{aligned}$$

Line (b) is due to Lemma 4. After unfolding the first and second terms of line (b), most of the unfolded items are identical and hence can be removed with the help of the set minus operator '\'. Then line (c) follows. Note that the first item of line (c) is an identity element **0**, we finally have line (d) and the correctness of Eq. 11. □

4.3 Incrementalization via memoization-vertex

We continue with the *memoization-vertex* (MV) policy. Unlike the MF and MP policies that record nothing or a small portion of effective messages, the MV policy records a state (aggregated result) for each vertex in every iteration. It deduces cancelation and compensation messages for incremental computation from the recorded states.

Conditions. The sufficient condition for applying MV policy in incrementalizaion is composed of two parts.

(1) The aggregation function $\mathcal{H}$ satisfies conditions **(C1)** and **(C2)**, *i.e.,* $\mathcal{H}$ has an inverse function $\mathcal{H}^-$ and supports partial aggregation.

Algorithm 3: Incrementalization via MV policy

Input: Graph G, graph updates ΔG, old results $\{x_v\}_{v\in V}$ of $\mathcal{A}$ over G and recorded states $\{m_v^i\}_{v\in V}$ for rounds $i = 1\ldots, K$.
Output: Updated results $\{x_v\}_{v\in V}$ *w.r.t.* $G\oplus\Delta G$.

```
for i = 1 ... K do
    foreach affected vertex v do
        m_v^{i'} ← U_a(m_v^i, M_v^{i-1});
        x_v^i ← U(x_v^{i-1}, m_v^i);   x_v^{i'} ← U(x_v^{i-1'}, m_v^{i'});
        foreach neighbor u of v do
            if G(x_v^i, P_E(v,u)) ≠ G(x_v^{i'}, P'_E(v,u)) then
                computes (M_v^-, M_v^+) via Eq. (12)-(13);
                propagate (M_v^-, M_v^+) and update M_u^i;
        m_v^i ← m_v^{i'}; // update the recorded states.
```

(2) The messages propagated in the i-th round are determined by vertex state alone. More specifically, we can rewrite the propagation function $\mathcal{G}$ can as

$$(\mathbf{C5})\ m_{v,w}^i = \mathcal{G}(x_v^i, *, P_E(v, w)).$$

where $*$ be any aggregated result. We say an algorithm $\mathcal{A}$ is MV-*applicable* if $\mathcal{A}$ satisfies conditions (**C1**)(**C2**)(**C5**).

Example 9 GCN-forward algorithm of Example 1(c) uses sum as its $\mathcal{H}$. Thus (**C1**) and (**C2**) hold. Condition (**C5**) also holds since the output of its propagation function can be written as $x_v^i \cdot W_i$.

Observe that MV policy shares two conditions on $\mathcal{H}$ with MF policy. The main difference is that in an MF-applicable algorithm, (i) the update function $\mathcal{U}$ and aggregation function $\mathcal{H}$ share the same logic; and (ii) the output message is generated based on the aggregated result only, *i.e.,* $m_{v,w}^i = \mathcal{G}(*, m_v^i, P_E(v, w))$. Instead, $\mathcal{U}$ and $\mathcal{H}$ of an MV-applicable algorithm can be very different, *e.g.,* sum and relu of GCN-forward algorithm. The message is created according to the latest vertex state, *i.e.,* condition (**C5**). As a result, an MV-applicable algorithm is required to track the context when applying $\mathcal{U}$. Fortunately, with (**C1**) and (**C2**), the recorded states suffice to produce cancelation and compensation messages in incremental computation. We next show how to incrementalize an MV-applicable algorithm $\mathcal{A}$ by deducing these messages.

Deducing messages. For a given MV-applicable algorithm $\mathcal{A}$, in the i-th round, each vertex v records a state m_v^i that represents the aggregated result after applying $\mathcal{H}$. Then cancelation and compensation messages are deduced in an iteration-wise manner.

Cancelation message. In the i-th round, suppose that a message from v to w_j is invalid. This can be decided as in MP policy, by verifying if $\mathcal{G}(x_v^i, m_v^i, P_E(v, w_j)) = \mathcal{G}(x_v'^i, m_v'^i, P_E'(v, w_j))$, where x_v^i and $x_v'^i$ are the vertex states

of v *w.r.t.* G and $G \oplus \Delta G$, respectively. If such verification fails, we define the cancelation messages M_v^- as

$$M_v^- = \{\mathcal{F}(x_v^i, *, P_E(v, w_j)) \mid \text{evolved } (v, w_j) \text{ in } G\}, \quad (12)$$

where $\mathcal{F} = \mathcal{H}^- \circ \mathcal{H} \circ \mathcal{G}$. As in MP policy, the correctness of M_v^- is warranted by conditions (**C1**) and (**C2**).

Compensation message. According to condition (**C5**), the compensation messages transmitted along evolved edges (v, w_j) can be generated directly from $x_v'^i$, *i.e.,* the updated vertex state. That is,

$$M_v^+ = \left\{ \mathcal{G}(x_v'^i, *, P'_E(v, w_j)) \,\middle|\, \begin{array}{l}\text{evolved } (v, w_j) \\ \text{in } G \oplus \Delta G\end{array} \right\}. \quad (13)$$

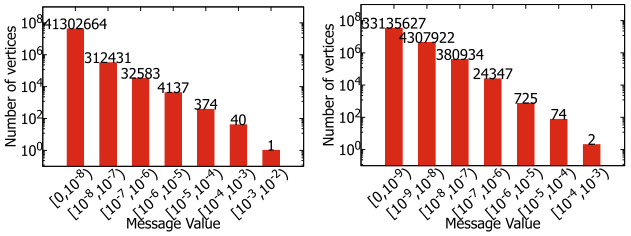
Incremental algorithm. Algorithm 3 outlines an incremental algorithm for MV-applicable programs. Starting from the initial round, it replays the computation on affected vertices with the recorded states and updates the results accordingly. Note that a vertex v is called *affected* if (i) v has received cancelation or compensation massages, or (ii) v is involved in the input updates ΔG. In each round i, the incremental algorithm first computes the new aggregated result $m_v'^i$ *w.r.t.* each affected vertex v, by aggregating the recorded state (aggregated result) m_v^i with messages M_v^{i-1} received from v's neighbors. (line 3). Here M_v^{i-1}, possibly empty, consists of cancelation and/or compensation messages. It then recovers the old vertex state x_v^i and derives the new state $x_v'^i$ directly using update function $\mathcal{U}$ (line 4). With x_v^i and $x_v'^i$ in place, it generates and sends cancelation and compensation messages when needed, *i.e.,* applying Eqs. (12) and (13) (lines 6–8). It also replaces the recorded state m_v^i by $m_v'^i$ for future use (line 9). The process terminates when all the previous rounds have been processed.

Intuitively, Algorithm 3 replays the computation to update affected vertex states, as in incremental GCN-forward of Example 5. In addition, many other GNN algorithms, *e.g.,* CommNet [45] are also MV-applicable.

Algorithm correctness. By induction on the rounds of the iterative computation, we verify the correctness of the incremental algorithm deduced via MV policy.

Theorem 3 (Correctness of MV policy) *Algorithm 3 correctly outputs the results* $\{x_v\}_{v \in V}$ w.r.t. $G \oplus \Delta G$, *for* MV*-applicable vertex-centric algorithms.*

Proof It is obvious that all the initial vertex states are correct in incremental computation in the initial round. Suppose that Algorithm 3 correctly updates all vertex states in $(k-1)$-th round of the iterative computation. That is, for each vertex, the vertex state $x_v'^{k-1}$ and the stored state $m_v'^{k-1}$ are correct as if they are obtained by running the batch MV-applicable algorithm $\mathcal{A}$ over the $G \oplus \Delta G$. We next analyze the states in round k.



(a) Twitter (b) UK-2005

Fig. 3 The distribution of deduced message values when performing PageRank

Denote by $\hat{x}_v'^k$ and $\hat{m}_v'^k$ the correct values of vertex state $x_v'^k$ and aggregated result $m_v'^i$ in the k-th round of $\mathcal{A}$ over $G \oplus \Delta G = (V', E', P'_E)$, respectively. By the definition of the vertex-centric model (Sect. 2) and condition (**C5**), we have $\hat{x}_v'^k = \mathcal{U}(\hat{x}_v'^{k-1}, \hat{m}_v'^k)$ and $\hat{m}_v'^k = \mathcal{H}\left(\{\mathcal{G}(\hat{x}_u'^{i-1}, *, P'_E(u, v)) \mid (u, v) \in E'\}\right)$. Note that $\hat{x}_v'^{k-1} = x_v'^{k-1}$ and $\hat{m}_v'^{k-1} = m_v'^{k-1}$ by the induction hypothesis. Hence $\hat{m}_v'^k$ can be computed by

$$\mathcal{H}\left(\{\mathcal{G}(\hat{x}_u'^{k-1}, *, P'_E(u, v)) \mid (u, v) \in E'\}\right) \quad \text{(a)}$$

$$= \mathcal{H}\Big(\{\mathcal{H}^- \circ \mathcal{H} \circ \mathcal{G}(x_u^{k-1}, *, P_E(u, v)) \mid (u, v) \in E\}$$

$$\cup \{\mathcal{G}(x_u'^{k-1}, *, P'_E(u, v)) \mid (u, v) \in E'\} \cup m_v^k\Big). \quad \text{(b)}$$

Here line (b) is true because of condition (**C1**). Observe that besides the stored old state m_v^k, the first and second terms of line (b) coincide with the forms of cancelation messages and compensation messages transferred in round k of Algorithms 3, respectively, for computing $m_v'^k$. The messages are communicated via evolved edges only, which triggers the actual change to the old state m_v^k. To this end, $m_v'^k$ is correctly updated due to condition (**C2**), *i.e.,* $m_v'^k = \hat{m}_v'^k$. As Algorithm 3 updates $x_v'^k$ based on the correct $m_v'^k$ and $x_v'^{k-1}$, we conclude $x_v'^k = \hat{x}_v'^k$. □

5 Asynchronous processing optimization

We introduce two effective optimizations, *selective processing* and *fast message propagation*, to further improve the performance of deduced incremental algorithms.

Observation. We start with two observations on message propagation of incremental computation. Recall that the computation of a vertex-centric algorithm $\mathcal{A}$ and its incremental counterpart $\mathcal{A}_\Delta$ can be treated as a message propagation process under the BSP synchronization model (see Sect. 2). While the BSP model makes it easy to reason about the correctness and convergence of both $\mathcal{A}$ and $\mathcal{A}_\Delta$, it also hinders the message propagation performance of $\mathcal{A}_\Delta$, as discussed below.

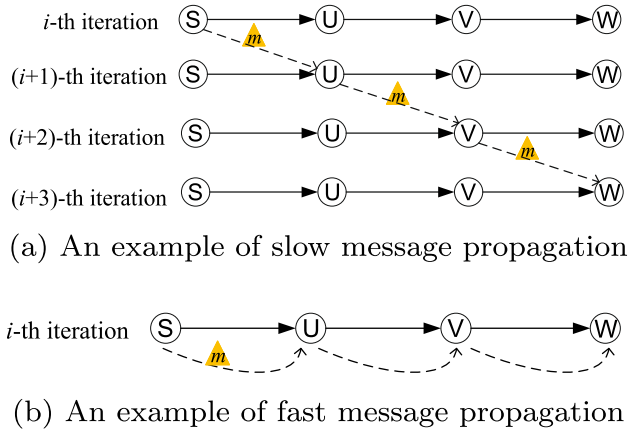


Fig. 4 Slow and fast message propagation

Observation 1. Most of the messages, including compensation and cancelation messages, are of "small values" in the sense that they make relatively small contributions to the convergence of $\mathcal{A}_\Delta$, while only a small portion of them are of "large values". This is because the messages propagated by $\mathcal{A}_\Delta$ are deduced and propagated from a small graph update ΔG. The BSP model takes a uniform treatment of all messages regardless of their values. This degrades performance, especially for accumulative algorithms like PageRank. A vertex-centric algorithm is accumulative if its computation can be characterized by Eq. (4). As shown in Fig. 3, with $|\Delta G| = 1\%|G|$, most of the messages processed by $\mathcal{A}_\Delta$ have small values; only about 2–3% of the messages have relatively large values. Here we treat the messages whose values are larger than 10^{-6} as large messages if the termination condition requires that every received message is smaller than 10^{-6}. That is, a large message makes a substantial contribution to the convergence. Intuitively, messages with small values have less effect on the vertex states. If for every receiving small message, $\mathcal{A}_\Delta$ executes the vertex program $(\mathcal{H}, \mathcal{U}, \mathcal{G})$ accordingly, the vertex program utilization can be largely reduced, resulting in performance degeneration of incremental computation. A better solution is to execute the vertex program on vertices with messages of large values or that have accumulated enough messages of small values.

Observation 2. The propagation of messages that are important for the convergence of $\mathcal{A}_\Delta$ can be slowed down due to the global synchronization barrier. In each iteration under the BSP model, a vertex only processes the messages received from the previous iteration. Take SSSP as an example and consider a sample graph as shown in Fig. 4a. Suppose that the *effective* messages for vertices u, v and w rely on the message originating from s. As shown in Fig. 4a, it takes 3 iterations for message m to propagate from vertex s to w before convergence. In contrast, the effective message can be propagated from s to w in one iteration if no global synchronization barrier is enforced (see Fig. 4b).

Optimizations for message propagation. We introduce two optimizations to address the inefficiency of message propagation when performing $\mathcal{A}_\Delta$. As will be seen shortly, both optimizations require that algorithm $\mathcal{A}$ is correct under the asynchronous execution model.

Selective processing. This optimization estimates the state change of a vertex v in the i-th iteration as $\Delta x_v = |x_v^i - x_v^{i-1}|$, and schedules those vertices with $\Delta x_v \geq \tau$ to execute their vertex programs, where τ is a predefined threshold. The vertices with $\Delta x_v < \tau$ are skipped to accumulate messages. Thus, selective processing improves message propagation performance by increasing the vertex program utilization of $\mathcal{A}_\Delta$.

Setting an appropriate value for the threshold parameter τ is crucial to ensure the convergence of algorithm $\mathcal{A}_\Delta$. If τ is set to an extremely small value, *e.g.*, $\tau = 0$, selective processing becomes ineffective because all the vertices that received messages will be selected for processing. On the contrary, if τ is set to a large value, the iteration may terminate before reaching the convergence condition as there may be no vertices with changes larger than τ to be processed, resulting in incorrect results. Therefore, the threshold should be set in accordance with the convergence condition of $\mathcal{A}_\Delta$.

In general, there are types of convergence conditions for vertex-centric algorithms (a) the change of each vertex between two consecutive iterations is smaller than a predefined value t, and (b) the sum of changes of all vertices between two consecutive iterations is smaller than a value T. For case (a), we can simply set $\tau = t$. When the change of each vertex is less than τ, no vertices will be selected for processing, indicating convergence. For case (b), τ can be set as $T/|V|$, where $|V|$ is the number of vertices. This choice ensures that when the change of each vertex is smaller than $T/|V|$, the sum of the changes of all vertices is also smaller than T, guaranteeing the convergence of iterative computation.

The next problem is to predict the changes of vertex states, which is not easy. Nevertheless, it is doable for accumulative algorithms. Indeed, by Eq. 4, we can use the aggregation of messages to predict the vertex state change. For example $\Delta x_v = \sum_{m \in M_v} m$ holds for PageRank, where M_v is the set of messages sent to v.

Fast message propagation. This optimization enables a vertex to update its state with all the received messages that have not been consumed, including those received in the current iteration, rather than only the messages from the previous iteration. As shown in Fig. 4b, the vertex u is updated with the message sent by s in the current iteration, similarly for vertices v and w. In this way, messages can be propagated more efficiently.

Fast message propagation helps accelerate the convergence of iterative computation. On the one hand, messages generated by the current iteration may make the states of ver-

tices closer to their converged states. If a vertex v is updated with the states of its neighbors that are closer to convergence, the state of v will also be closer to the converged state after updating. On the other hand, updating vertices with new messages, generating and propagating new messages to others further enables the messages to propagate more efficiently [56]. For example, in Fig. 4b, the message m is propagated from s to w only using 1 iteration instead of 3 iterations.

Optimization applicability and correctness. Observe that both optimizations require asynchronous message propagation and processing of $\mathcal{A}_\Delta$, *i.e.,* $\mathcal{A}_\Delta$ permits non-uniform processing of vertices and no global synchronization barrier is enforced. Both optimizations cannot be applied to the incremental algorithm obtained via the MV-policy since the incremental processing of $\mathcal{A}_\Delta$ is essentially synchronized, *i.e.,* $\mathcal{A}_\Delta$ restores the messages and replays the computation by following the BSP model. Similarly, the optimizations cannot be applied to an incremental algorithm $\mathcal{A}_\Delta$ obtained via the ME-policy. In contrast, the theorem below shows that if a vertex-centric algorithm is MF-applicable or MP-applicable, then both optimizations can be applied to the incremental counterpart $\mathcal{A}_\Delta$, without worrying about the correctness. The rationale is that if $\mathcal{A}$ is either MF-applicable or MP-applicable, then both $\mathcal{A}$ and $\mathcal{A}_\Delta$ are correct under the asynchronous model.

Theorem 4 (Correctness of optimizations) *The incremental counterpart $\mathcal{A}_\Delta$ of an* MF*-applicable or* MP*-applicable algorithm $\mathcal{A}$ converges correctly with the asynchronous optimizations.*

Proof The computation of $\mathcal{A}_\Delta$ has two phases. In the first phase, $\mathcal{A}_\Delta$ computes intermediate states $\hat{\mathbb{X}}^0$ over $G \oplus \Delta G$ and generates initial messages $\hat{\mathbb{M}}^0$; in the second phase, $\mathcal{A}_\Delta$ restores the computation from $(\hat{\mathbb{X}}^0, \hat{\mathbb{M}}^0)$ as in $\mathcal{A}$. By Theorem 1 and Theorem 2, the computation starting from $(\hat{\mathbb{X}}^0, \hat{\mathbb{M}}^0)$ converges to the correct result over $G \oplus \Delta G$ under the synchronous model. As pointed out in previous studies, if the update function $\mathcal{H}$ and the aggregation function $\mathcal{U}$ of $\mathcal{A}_\Delta$ share the same function and $\mathcal{A}_\Delta$ satisfies (**C2**) and (**C3**), then any asynchronous run of $\mathcal{A}_\Delta$ converges to the same answer as that deduced in synchronous model (see Theorem 1 of [56]). The result follows by observing that both MF-applicable and MP-applicable algorithms satisfy these prerequisites. □

Remark Fast message propagation accelerates message propagation and speeds up vertex updates, making it effective for all MF-applicable and MP-applicable algorithms. In contrast, the effectiveness of selective processing varies among different algorithms.

(1) Selective processing is typically effective with algorithms that employ sum or product as aggregation and update functions, such as PageRank and PHP. These algorithms aim for *high-precision* convergence, where vertex states continually change and convergence is only reached when changes are very small. Therefore, τ can be set to a value strictly larger than "0" and vertices whose change is greater than τ are selected for processing, ensuring both convergence and but also acceleration.

(2) If the aggregation and update functions involve min or max. To ensure convergence with selective processing, we must set the threshold $\tau = 0$. That is, selective processing becomes ineffective, as it degenerates to processing all vertices with message. Typical algorithms falling into this category includes SSSP and weakly connected components (see Sect. 7.5).

6 Ingress

As a proof of concept, we design and implement an incremental graph computing system Ingress.

6.1 Vertex-centric API

Following the vertex-centric model of Sect. 2, Ingress provides the API, shown in Fig. 5, to users for writing batch vertex-centric algorithms. Here `D` and `W` are the template types of vertex states and edge properties, respectively. In addition, the initial values of the vertex states and messages should be set via the `init_v` and `init_m` interfaces, respectively. function, which is specified The aggregation function $\mathcal{H}$ is implemented using the `aggregate` interface. Note that `aggregate` has only two input parameters, while function $\mathcal{H}$ can naturally take any number of inputs. However, `aggregate` can be generalized to support different numbers of input parameters if $\mathcal{H}$ has the associative property (*i.e.,* condition (**C2**) of Sect. 4 holds). That is, $\mathcal{H}(x_0, x_1, x_2) = \mathcal{H}(\mathcal{H}(x_0, x_1), x_2)$. We let `aggregate` have two input parameters for the simplicity of operator extraction, which will be used in automatic condition checking (see below). Without loss of generality, we also provide another interface for function $\mathcal{H}$, which can take a vector of elements as input. The update function $\mathcal{U}$ of the vertex-centric model is specified by the `update` interface, for adjusting vertex states; and the interface `generate` in the API corresponds to propagation function $\mathcal{G}$, for generating messages.

Using this API, the implementation of the batch SSSP algorithm of Example 1(b) is shown in Fig. 6.

6.2 Automatic memoization policy selection

As presented in Sect. 3, there exist multiple memoization policies for incrementalization, which lead to different space costs. Though their formal applicability conditions are provided in Sect. 4, it is nontrivial for non-expert users to choose

```
template <class D, class W>
interface IteratorKernel{
    virtual void init_m(Vertex v, D m) = 0;
    virtual void init_v(Vertex v, D d) = 0;
    virtual D aggregate(D m1, D m2) = 0;
    virtual D update(D v, D m) = 0;
    virtual D generate(D v, D m, W w) = 0;
}
```

Fig. 5 The vertex-centric API of Ingerss

the best-fit one. Ingress automatically selects the optimal memoization policy with the help of Satisfiability Modulo Theories (SMT) solver Z3 [36]. SMT studies the problem of deciding whether a given first-order formula is satisfiable, *i.e.,* if there is an assignment of proper values to uninterpreted functions and constant symbols to make the formula to be true. The SMT solver Z3 asserts a formula and may return "satisfiable" (sat), "unsatisfiable" (unsat) or "unknown".

The initial step of policy selection uses a parser in Ingress to extract the three functions $\mathcal{H}$,$\mathcal{U}$ and $\mathcal{G}$ of the vertex-centric model, from the implementations of interfaces `aggregate`, `update` and `generate`, respectively. Then the sufficient conditions on these functions (see Sect. 4) are converted into different Z3 formulas by our predefined Z3 templates. For instance, condition (**C1**) states whether $\mathcal{U}$ has a reverse function $\mathcal{U}^-$. Its Z3 assertion template is shown as follows:

```
(assert(forall ((x1 Real) (x2 Real) (x3 Real))
(= (f x1 x3)) (f x1 (f (f (f1 x2) x2) x3)))).
```

Here `f` represents the update function $\mathcal{U}$ extracted from user's program, and `f1` is a declared function (*i.e.,* $\mathcal{U}^-$) to be searched for. Additionally, the whole set of variables `x1`, `x2` and `x3` corresponds to the input set M in condition (**C1**), while `x2` itself constitutes the subset M'. If the assertion formula gets "sat" in Z3, (**C1**) is satisfied and the satisfiable function `f1` (*i.e.,* $\mathcal{U}^-$) can be automatically found. Conditions (**C2**) and (**C3**) are the same as the that of monotonic recursive aggregation defined in [51], so we reuse their Z3 templates. The Z3 template for condition (**C4**) is shown as follows:

```
(assert (not (forall ((x1 Real) (x2 Real))
(or (= (f x1 x2) x1) (= (f x1 x2) x2))))).
```

It states that `f` returns either one of its two inputs. Note that Z3 cannot determine "whether a formula Y is always true?", but only answers "whether it is satisfiable?". To verify a property Y that should always be true, we convert "Y is always true" into "NOT Y is not satisfiable". Therefore, if the above Z3 assertion returns "unsat", condition (**C4**) is verified true. Condition (**C5**) can be simply validated by static program analysis (*i.e.,* whether the output of `generate` depends on only one of its input parameters).

```
class SSSPKernel: public IteratorKernel{
    void init_m(Vertex v, double m){m = DBL_MAX;}
    void init_v(Vertex v, double d){
      v.d = ((v.id == source) ? 0 : DBL_MAX);
    }
    double aggregate(double m1, double m2){
      return m1 < m2 ? m1 : m2;
    }
    double update(double v, double m){
      return aggregate(v, m);
    }
    double generate(double v, double m, double w){
      return v + w;
    }
}
```

Fig. 6 The implementation of SSSP algorithm

With such automated condition verification mechanism, Ingress automatically chooses the memoization policy as follows. At first, if $\mathcal{H}$ and $\mathcal{U}$ are identical and conditions (**C2**) and (**C3**) are both satisfied, it prefers to select MF and MP as candidate policies. Next, if condition (**C1**) is satisfied, then MF policy is chosen; otherwise when condition (**C4**) holds, MP policy is chosen. If the first two preferable memoization policies are not feasible, Ingress chooses the MV policy by checking whether conditions (**C1**), (**C2**) and (**C5**) hold. For the rest cases, the ME policy is selected by default.

Policy selection can be conducted offline, whose cost depends on the characteristics of the vertex-centric programs only, *i.e.,* $\mathcal{H}$, $\mathcal{U}$ and $\mathcal{G}$, rather than the large-scale graphs. In fact, deciding the satisfiability of a first-order formula is undecidable in general [15], *e.g.,* in the presence of integer arithmetic with multiplication [32]. However, as verified in our experiments, for functions of most common vertex-centric algorithms, *e.g.,* those in Example 1, Z3 can respond quickly when checking the above formulae (see Sect. 7).

6.3 Distributed runtime engine

The distributed runtime engine of Ingress is developed on top of libgrape-lite [28] (an open-source version of GRAPE [10]), which is designed to be a highly efficient, flexible, and scalable platform for distributed graph computation. The graph structure and the computation states are stored independently in libgrape-lite, which is supremely suitable for incremental graph computation since the state maintenance and the graph structure adjustment have to be separated in incremental processing. Ingress inherits the graph storage backend and graph partitioning strategies from libgrape-lite. Besides, it has the following new modules.

Vertex-centric programming. Libgrape-lite only supports block-centric programming. Ingress extends it to achieve vertex-centric programming. Specifically, Ingress spawns a new process on each worker to handle the assigned subgraph.

It adopts the CSC/CSR optimized graph storage of libgrape-lite for fast query processing of the underlying graphs. For each vertex, it invokes the user-specified vertex-centric API to perform the `aggregate`, `update`, and `generate` computations. The generated messages are batched and sent out together after processing the whole subgraph in each iteration. Ingress relies on the message passing interface of libgrape-lite for efficient communication with other workers.

Data maintenance. Ingress launches an initial batch run on the original input graph. It preserves the computation states during the batch iterative computation, guided by the selected memoization policy, *e.g.,* preserving the converged vertex states only as in MF policy or the effective messages with MP policy. After that, Ingress is ready to accept graph updates and execute the deduced incremental algorithms to update the states. The graph updates can include edge insertions and deletions, as well as newly added vertices and deleted vertices. In particular, the changed vertices with no incident edges are encoded in "dummy" edges with one endpoint only. Furthermore, changes to edge proprieties are represented by deletions of old edges and edge insertions with the new properties.

Optimized asynchronous execution. The parallel execution engine of libgrape-lite run in the synchronous model by default. Ingress extends it to support both synchronous and asynchronous execution models. The asynchronous execution model is equipped with selective processing and fast message propagation optimizations as discussed in Sect. 5. Ingress employs Z3 *implicitly* to ensure the correctness when running in the asynchronous model. This is because Ingress determines its execution model based on the memoization policy automatically selected by Z3. For a given vertex-centric algorithm, if MF or MP policy is applied, Ingress utilizes the asynchronous execution model since MF-applicable and MP-applicable algorithms can run in the asynchronous model without worrying the correctness (Theorem 4). Otherwise, Ingress defaults to the synchronous model.

Incremental processing. Ingress starts the incremental computation from those vertices involved in the input graph updates, which are referred to as *affected vertices*. Using the message deduction techniques presented in Sect. 4, for each of these affected vertices, Ingress will generate the cancelation messages and compensation messages based on the new edge properties and the preserved states. These messages are sent to corresponding neighbors. Only the vertices that receive messages are activated by Ingress to perform the vertex-centric computation, and only the vertices whose states are updated can propagate new messages to their neighbors. This process proceeds until the convergence condition is satisfied.

Table 3 Real-life graphs

Graph	#Vertices	#Edges
Twitter-2009 (TW) [40]	41,652,230	1,468,365,183
UK-2005 (UK) [48]	39,459,925	936,364,282
Euro-Road (ER) [6]	50,912,018	108,109,320
US-Road (UR) [41]	23,947,347	57,708,624
Friendster (FS) [53]	65,608,366	1,806,067,139

7 Experimental study

We evaluate Ingress thoroughly from various aspects.

7.1 Experimental setup

We evaluated Ingress with five incremental algorithms deduced from (i) two MF-applicable algorithms PageRank and PHP [16], (ii) two MP-applicable algorithms SSSP and Connected Components (CC) [3], and (iii) one MV-applicable algorithm GCN-forward. The batch CC algorithm aims to finds all connected components, where $\mathcal{U}{=}\mathcal{H}{=}\min$ and $\mathcal{G}(x_v^i, m_v^i, P_E(v,w)){=}m_v^i$. It is MP-applicable, *i.e.,* (**C2**)-(**C4**) hold. In the inference process of GCN-forward, we used $K = 3$ randomly generated weight matrices, where the sizes of the matrices are 128×64, 64×32 and 32×16, respectively. By default, for PageRank, PHP, SSSP and CC, Ingress runs in the asynchronous model with selective processing and fast message propagation enabled, and runs in the synchronous model for GCN-forward. Both PageRank and PHP offer two ways to set up the convergence condition. Unless stated otherwise, we terminate them when the sum of the change of all vertices is smaller than $T = 10^{-2}$.

Datasets and updates. We used four real-life graphs in our evaluation (see Table 3), including social networks Twitter (TW) [40] and Friendster (FS) [53], web graph UK-2005 (UK) [48], and road networks Euro-Road (ER) [6] and US-Road (UR) [41]. Here the largest dataset FS is used to evaluate the distributed runtime performance of Ingress. We also designed a graph generator for evaluating the performance of the systems on generated synthetic graphs.

We constructed graph updates ΔG by randomly adding new edges to G and removing existing edges from G. The number of added edges and deleted edges are the same, unless stated otherwise. The updates ΔG refer to topological changes by default.

Competitors. We compared Ingress with five state-of-the-art vertex-centric incremental graph processing systems, Torando [44], GraphBolt [31], DZiG[30], KickStarter [50], and RisGraph[11]. We also implemented a competitor on top of libgrape-lite [28], denoted as IngressR, which reperforms the vertex-centric computation over the updated graph start-

ing from scratch. It is used to validate the effectiveness of incremental processing.

Since no prior system covers as many algorithms as Ingress for incrementalization, we only compared Ingress with each competitor on the algorithms it specializes in. KickStarter and RisGraph have the state-of-art performance on MP-applicable SSSP and CC, but they cannot handle PageRank, PHP and GCN-forward due to its single-dependency requirement on the vertex states. Tornado returns erroneous results for SSSP, CC and GCN-forward since the initial states have impact on the final result. GraphBolt and DZiG is supposed to support all, but its implementations for SSSP, CC and GCN-forward are nontrivial and not open-sourced. In light of this, we only tested PageRank and PHP (resp. SSSP and CC) on GraphBolt, DZiG, and Tornado (resp. KickStarter and RisGraph).

Environments. We used AliCloud ecs.r6.13xlarge instance (52vCPU, 384GB memory) for experiments conducted on a single machine. To evaluate Ingress in a distributed environment, we adopted a cluster of 32 AliCloud ecs.r6.6xlarge instances (24vCPU, 192GB memory).

7.2 Overall performance

We first evaluated the overall performance of Ingress, including both the response time and space cost, by comparing it with competitors. Since GraphBolt, DZiG, KickStarter and RisGraph can only run on a single machine with multi-core support, this set of experiments and the following ones were conducted on a single machine except the distributed evaluation in Sect. 7.8 below.

7.2.1 Performance on graph topology updates

We first evaluated the performance of Ingress with graph topology updates. We fixed the size of topological updates as $|\Delta G|=1\%|G|$. When reporting response time, we omit the cost for policy selection since it can be done within 50 milliseconds for all tested algorithms.

Response time. Figure 7 shows the *normalized* response time of each algorithm executed in different systems. Here the response time of IngressR is treated as the baseline, *i.e.,* IngressR finishes in unit time. We can see that Ingress outperforms others except Risgraph in all the cases. Specifically, Ingress achieves 6.71×−47.47× (17.83× on average) speedup over GraphBolt, 3.41×−24.05× (10.99× on average) speedup over DZiG, 2.0×−35.85× (11.88× on average) speedup over Tornado, 1.34×−7.22× (2.45× on average) speedup over KickStarter, and 1.20×−49.23× (12.03× on average) speedup over IngressR. RisGraph has comparable performance to Ingress on SSSP and CC, e.g., for SSSP, Ingress outperforms RisGraph on datasets UR and UK, while RisGraph outperforms Ingress on TW and ER. Ingress is indeed very efficient, *e.g.,* taking only 2.5 s for PHP on the UK dataset, as opposed to 61, 19, and 26 s by GraphBolt, DZiG and Tornado, respectively. The MF policy (for PageRank and PHP) and the MP policy (for SSSP and CC) exhibit substantial superiority compared with the MV policy (for GCN-forward). This is under expectation because MF and MP require less amount of recomputation and memoized states.

Space cost. We measured the size of the memory for storing computation states. Figure 8 depicts the space cost of each system. We find that Ingress benefits greatly from its flexible memoization strategy. It is much more memory efficient than GraphBolt and DZiG for PageRank and PHP, as shown in Fig. 8a, b. This is because the MF engine of Ingress does not store any intermediate states, while GraphBolt and DZiG maintain states across iterations. Tornado starts from the previously converged states, which needs no additional memory either. IngressR also holds no intermediate results. Therefore, one can find similar space costs for Ingress, IngressR, and Tornado. However, Ingress is much faster than the other two (Fig. 7a, b). For SSSP and CC, Ingress chooses the MP engine to store a small set of paths. This incurs more space than IngressR, which is under expectation. However, Ingress requires less memory than KickStarter and RisGraph (Fig. 8c, d). This is because KickStarter and RisGraph sorts additional information for the paths [11, 50]. For GCN-forward, Ingress adopts the MV engine, recording more intermediate results, hence takes morespace than IngressR, *i.e.,* recomputation (Fig. 8e).

7.2.2 Performance on edge weight updates

We then evaluated the performance of Ingress with edge weight updates. For a fair comparison, when evaluating the performance of each system, each edge weight update is simulated by deleting an edge first and then adding a new edge with an updated weight. Similar to the evaluation on topology updates, we generated edge weight updates randomly and fixed the update size by $|\Delta G| = 1\%|G|$. We only tested PHP and SSSP since the two can be executed on weighted graphs.

Figure 9 reports the normalized response time of algorithms PHP and SSSP executed in each system, where we still use the response time of IngressR as the baseline. Figure 10 shows the space cost of each system. It is shown that similar to the graph topology change, Ingress also outperforms other systems when changing the edge weights and uses fewer memory footprints thanks to its different memoization policy.

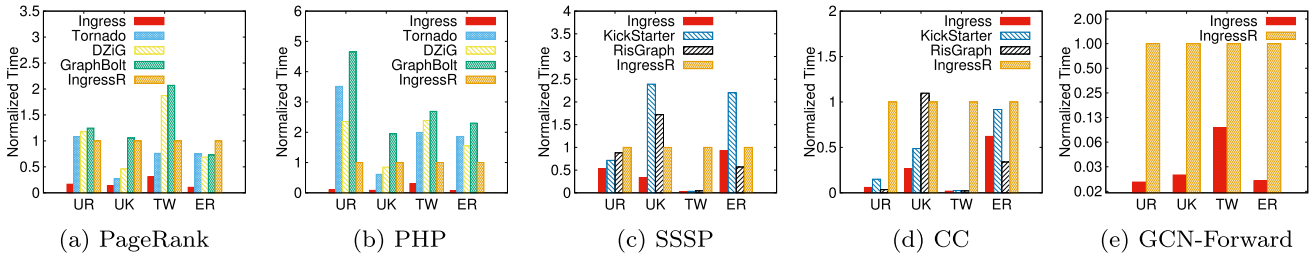


Fig. 7 Running time comparison

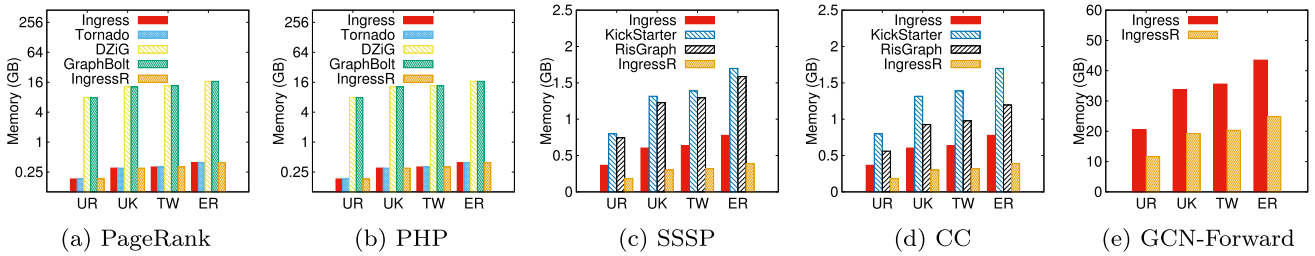


Fig. 8 Space cost comparison

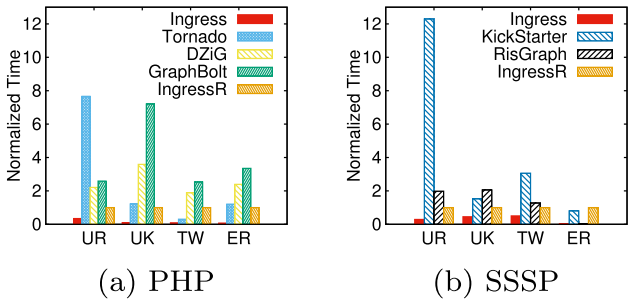


Fig. 9 Runtime comparison with edge weight change

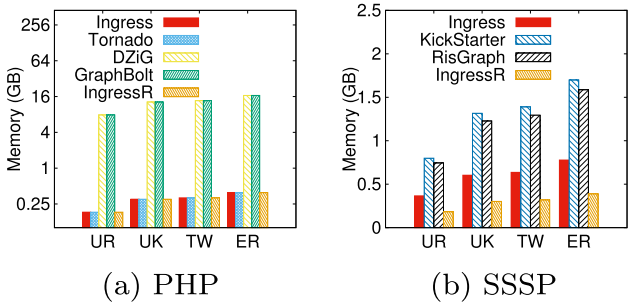


Fig. 10 Space cost comparison with edge weight change

7.3 Sensitivity to updates

We next evaluated the impact of the input updates on the performance of incremental graph processing. Varying the size $|\Delta G|$ of input updates ΔG from 1 to 20% of the size $|G|$ of the original graph G, Fig. 11 shows the running time for the incremental computation of PageRank, SSSP and GCN-forward in different systems over the dataset UK. We find the following.

(1) For PageRank and GCN-forward, almost all the incremental graph processing systems take longer to process larger graph updates ΔG, as expected. For SSSP, the response time of all systems is not sensitive to the size of ΔG. This is because even if the size of ΔG is small, *i.e.,* $|\Delta G| = 1\%|G|$, ΔG still contains quite a few "upstream" edges close to the source vertex. Observe that an upstream edge update can result in a large number of vertices that require updates.

(2) For PageRank and SSSP, Ingress consistently outperforms other incremental processing systems. For GCN-forward, Ingress takes more time than IngressR when $|\Delta G| \geq 10\%|G|$. Regarding this result, we analyzed the vertex activation log and found that the input updates affect almost all the vertices, making the cost of incremental processing close to that of recomputation *i.e.,*IngressR. As a result, Ingress spends more time than IngressR due to additional state maintenance costs.

(3) Since the number of inserted edges and that of deleted edges are the same in our randomly created input updates ΔG, the size of the updated graph remains the same, *i.e.,* $|G \oplus \Delta G| = |G|$. IngressR reperforms the batch computation on the updated graph with a fixed size, so it is not sensitive to $|\Delta G|$. Observe that the runtime of IngressR for SSSP fluctuated. This is because the runtime of SSSP is sensitive to the topology of the graph. A tiny topology change of the graph may cause the execution time of SSSP to vary greatly.

(4) Ingress is very effective in the incremental computation of MF-applicable algorithm PageRank. In fact, it takes less

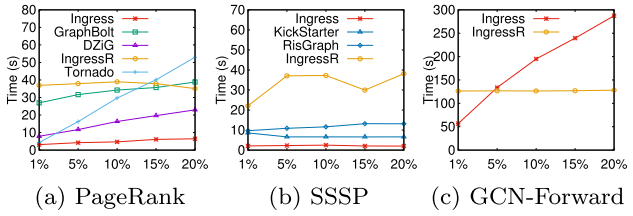


Fig. 11 Sensitivity to $|\Delta G|$

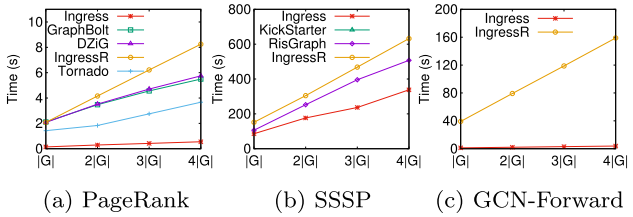


Fig. 12 Sensitivity to $|G|$ (time)

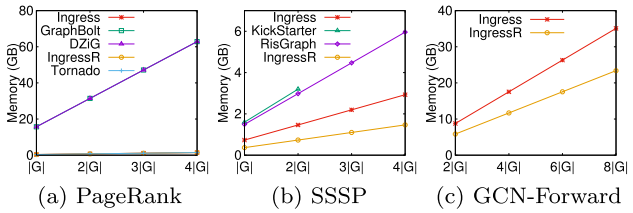


Fig. 13 Sensitivity to $|G|$ (space)

than 2 seconds when $|\Delta G|$ is up to 20% of $|G|$, and is still faster than recomputation (*i.e.,* IngressR).

The space costs of all the systems are almost stable when varying $|\Delta G|$. This is because that the memory usage of these systems only depends on the size $|G|$ of original graph, rather than $|\Delta G|$.

7.4 Sensitivity to graph sizes

We also conducted experiments to evaluate the impact of the graph size $|G|$ on the performance of incremental graph processing. Here we used synthetic graphs produced by a generator. The graphs have up to 47 million vertices and 115 million edges and follow the node degree distribution of real-life graphs, *e.g.,* UR. We fixed $|\Delta G| = 0.57M$, *i.e.,* 0.57 million of edge updates. Varying the size of synthetic graphs from 47 million vertices and 115 million edges to 191 million vertices and 461 million edges, denoted as $|G|$ to $4|G|$, Figs. 12 and 13 report the running time and space costs of different systems, respectively.

(1) We find that the response time of all systems gradually increases as the graph size grows. This is because the number of vertices that are affected by $|\Delta G|$ also increases slowly. Compared to IngressR that conducts recomputation and other incremental graph processing systems, the response time of our Ingress is less sensitive to the increase of $|G|$. This is because the time complexity of Ingress mainly depends on $|\Delta G|$, rather than $|G|$. Tornado updates previous results by directly starting the iterative computation on the new graph with the converged states, so its running time also depends on $|G|$ and exhibits a fast-increasing rate. GraphBolt and DZiG are more sensitive to the size of $|G|$ than Ingress. This is because more vertices are affected by ΔG and both systems update more vertex intermediate states than Ingress, which takes more time.

(2) Thanks to our memoization-free (MF) technique, Ingress shows substantial superiority over GraphBolt and DZiG on space cost. Ingress is also more space efficient than KickStarter and RisGraph with its MP policy. We find that KickStarter and RisGraph use more space to maintain the dependency information than Ingress. These are consistent with the results in Sect. 7.2. In practice, the user can benefit more from the high space efficiency of Ingress when processing larger graphs, not to mention that Ingress is able to automatically choose the best-fit memoization engine for different algorithms without users' intervention.

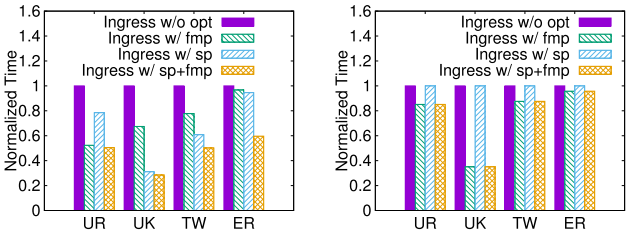


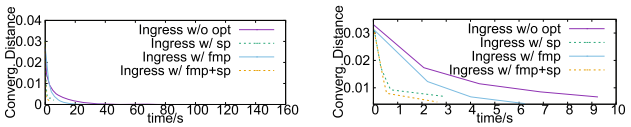
Fig. 14 The effectiveness of asynchronous optimizations

7.5 Effectiveness of asynchronous optimizations

To verify the effectiveness of optimizations proposed in Sect. 5, we first ran Ingress without any optimizations (Ingress w/o opt), we then turned on the functionalities of selective processing (Ingress w/ sp), the fast message propagation (Ingress w/ fmp), and both optimizations (Ingress w/ sp+fmp). Since both optimizations can only be applied on MF-applicable and MP-applicable algorithms. We took PageRank and SSSP as the test workloads in this evaluation. The size of the graph updates was set as $|\Delta G| = 1\%|G|$. Figure 14 shows the normalized response time of Ingress with different optimizations. We find the following.

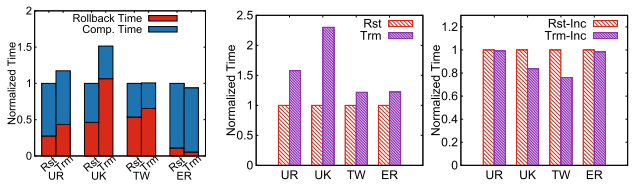
(1) Fast message propagation is effective for both PageRank and SSSP. It alone improves PageRank and SSSP by 1.42× and 1.55× on average, respectively. This is because fast message propagation accelerates the message propagation and speeds up vertex updates.

(2) Selective processing alone improves the performance of PageRank by 1.79× on average, up to 3.21×. However, it



(a) $\tau = t$ (b) $\tau = \frac{T}{|V|}$

Fig. 15 The impact of threshold parameter setting methods on PageRank convergence



(a) Breakdown (b) Rollback Time (c) Comp. Time

Fig. 16 Resetting versus trimming on SSSP

has little effect on SSSP. This is because SSSP takes min as its aggregation and update functions and in that case selective processing degenerates to processing all vertices with messages (see Sect. 5).

(3) With both optimizations enabled, the performance of PageRank is improved by 1.67 × −3.51×.

7.6 Impact of threshold parameter setting methods

We further evaluated the impact of two threshold parameter setting methods in selective processing on convergence acceleration. PageRank serves as the workload since it adapts to both methods, $\tau = t$ and $\tau = T/|V|$. In order to use the same τ value, we set $T = 10^{-2}$ and $t = T/|V|$, so that $\tau = 10^{-2}/|V|$ in both cases. Then a vertex is selected for processing only if its change is larger than $10^{-2}/|V|$. Fixing $|\Delta G| = 1\%|G|$, Fig. 15 reports the convergence states of PageRank over time on the UK dataset. To this end, we pre-computed convergence states x_v^* offline for each vertex v and used $\sum_{v \in V} |x_v - x_v^*|$ as the distance to convergence.

(1) PageRank correctly converges in 2.63 s (2.87 s without fmp) and 3.48 s (5.81 s without fmp) for $\tau = t$ and $\tau = T/|V|$, respectively. This verifies the correctness of selective processing in ensuring convergence.

(2) Selective processing significantly accelerates the convergence of PageRank, whether with or without fast message propagation optimization. On average, selective processing accelerates the convergence of PageRank by 13.58×, up to 26.38×. This is because selective processing greatly improves the utilization of vertex programs by skipping vertices with accumulated changes below the threshold τ. In addition, selective processing improves the convergence speed better than fast message propagation in both settings.

(3) Selective processing has a better convergence acceleration effect for the case $\tau = t$ than for the case $\tau = T/|V|$. On average, it improves the convergence by 24.36× for $\tau = t$, while for the case $\tau = T/|V|$, it improves the convergence by 2.79×. Note that we set $\tau = 10^{-2}/|V|$ in both cases. The difference in acceleration is due to the performance gap of the corresponding baselines without selective processing. On average, it takes 77 s (153 s without fmp) for PageRank to converge when requiring the change of every vertex between two consecutive iterations is below $10^{-2}/|V|$, while in the other case, it is only 6.23 s (9.25 s without fmp).

7.7 Resetting versus trimming

We next evaluated the effectiveness of the resetting-based rollback (Rst) for MP-applicable algorithms, comparing it with the trimming-based rollback (Trm) adopted by KickStarter (see Sect. 4.2). To ensure a fair comparison and eliminate the impact of system implementation, we have made our best to implement Trm within Ingress. Fixing $|\Delta G| = 1\%|G|$, we took SSSP as the test workload since its incremental computation requires rollback. We recorded the normalized total time of rollback and iterative computation, along with their breakdown, in Fig. 16a. Here the time of Rst plus the time for the subsequent iterative computation is the baseline. The comparison of rollback time and incremental iterative computation time after rollback are presented in Fig. 16b, c respectively. We find the following.

(1) The overall performance of SSSP using Rst and Trm is generally comparable. While Rst incurs less cost than Trm, Trm beats Rst in the subsequent iterative computation after rollback. On the UK dataset, Rst outperforms Trm by 1.51×. In that case where the overall cost is dominated by the rollback process, Rst incurs substantially fewer overheads than Trm (see below).

(2) As anticipated, Trm constantly takes more time than Rst on all datasets. On dataset UK, Trm takes 2.3× longer than Rst. This is because 95% of the vertices are affected by ΔG, leading to more overheads in Trm than in Rst since Trim has more sophisticated processing logic.

(3) The iterative computation after Trm performs slightly better than that after Rst. We analyzed the log and found the following: (i) on average 41% fewer vertices are reset to the initial values in Trm compared to Rst; (ii) after two rounds of iterations following Rst, the number of vertices with initial values is already less than that after Trm. In other words, Trm can be effectively replaced by Rst plus two subsequent iterative computation. When the number of iterations after rollback is very large, such as more than 100 rounds in UR and ER, the performance gains brought by Trim are reduced significantly.

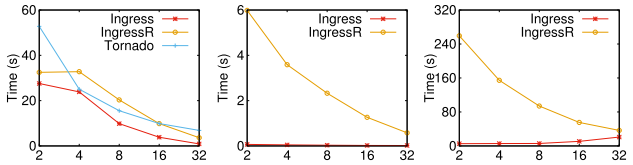


Fig. 17 Distributed runtime performance over FS

7.8 Distributed runtime performance

We evaluated the distributed runtime of Ingress, which is essential for handling large-scale graphs. As GraphBolt and KickStarter do not support distributed computation, we compared Ingress with Tornado and IngressR only in the distributed environment. We applied PageRank, SSSP and GCN-forward over the large FS graph, on our Alicloud cluster. Varying the number of workers from 2 to 32 in the cluster, Fig. 17 shows the response time of different systems. One can see that Ingress needs shorter running time than IngressR and Tornado on different-sized clusters and shows good scalability. In particular, for SSSP, Ingress is much faster than the recomputation-based IngressR, say 31×–88× speedup. This highlights the need of incremental processing for big graphs. For GCN-forward, Ingress becomes slower when the number of workers increases from 16 to 32. This is due to the increased communication cost, which is larger than the actual computational cost for incremental processing that is already very small.

8 Related work

Incremental graph processing systems. There have been systems developed for incremental graph processing, *e.g.,* [21, 31, 43, 44, 50, 57]. Tornado [44] is an incremental iterative processing system that is built on top of Storm. It only focuses on those graph computations that can converge to the same state from various initial states. GraphIn [43] incrementally handles dynamic graphs through fixed-sized batches. KickStarter [50], RisGraph [11] and GraphBolt [31] are three dependency-driven systems. KickStarter and RisGraph are able to execute graph algorithms that are monotonic, and deduce safe approximation results upon edge deletions, to fix the approximation errors via iterative computation. GraphBolt keeps track of the dependencies through the memoized aggregated values among iterations. When input updates arrive, it refines the dependencies iteration-by-iteration to do incremental computation. Tripoline [21] uses graph triangle inequalities to accelerate incremental graph computation by employing the previous computation results. TDGraph [57] is the first programmable accelerator to augment the many-core processor for high-performance streaming graph processing. Apart from these, [54] proposes a new message passing policy for vertex-centric programming, which only exchanges meaningful results via Δ-messages. Although it helps reduce the transmitted messages, changes to input graphs are not allowed. Extending timely dataflow [37], differential dataflow [35] achieves streaming processing by enforcing a partial order on the versions of computations. However, it stills needs to maintain a number of intermediate versions. There has been work on incrementalizing generic programs, *e.g.,* [1, 4, 26], often at the instruction level. They are hard to be applied for incremental graph processing directly.

This work differs from the prior work in the following. (1) We target the incrementalization of generic vertex-centric algorithms, beyond the scope of specific classes of computations that satisfy certain conditions [44, 50]. (2) We introduce four types of memoization policies to facilitate the incrementalization and provide sufficient conditions for their applicability, which have not been considered in previous work. (3) We make the process of incrementalization accessible to non-expert users, rather than asking nontrivial operators from the users [31].

There have also been systems proposed, *e.g.,* Cavs [52] for improving the performance of training and inference of dynamic neural network models. They focus on the changes to models rather than the updates to data graphs, hence are orthogonal to this work.

Incremental graph algorithms. A number of incremental graph algorithms have been proposed for, *e.g.,* regular path queries [7], strongly connected components [19], subgraph isomorphism [23], k-cores [25], graph partitioning [8] and triangle counting [34]. In contrast to these ad hoc methods, we propose to automatically deduce incremental algorithms from the batch counterparts by a generic approach, making incremental graph processing easier. Close to this work is [9], which focuses on incrementalizing graph-centric algorithms based on fixepoint computation. Instead, this work targets on vertex-centric algorithms and introduces a flexible memoization scheme to optimize the memory usage in incremental graph processing, which is not study in [9].

9 Conclusion

We have proposed Ingress, a system that optimizes memory usage for incremental vertex-centric algorithms. Ingress employs four memoization policies to handle various vertex-centric algorithms, along with identified conditions for their application. Our experiments confirm Ingress as a promising tool for incremental graph processing. One topic for future work is to extend Ingress to support the incrementalization of graph-centric algorithms.

Acknowledgements The work is supported by the National Natural Science Foundation of China (62202301, 62072082, U2241212, 62202088, 62137001, 62302027), the 111 Project (B16009), the Fundamental Research Funds for the Central Universities (N2216012, N2216015), and a research Grant from Alibaba Innovative Research (AIR) Program.

Appendix

A1. Proof of Lemma 1

Proof Following the computation process of an MF-applicable $\mathcal{A}$ as shown in Eq. (4), we have that

$$\begin{aligned}
\mathbb{X}^k &= \mathcal{U}(\mathbb{X}^{k-1} \cup \mathbb{M}^{k-1}) && \text{(a)}\\
&= \mathcal{U}\left(\mathcal{U}(\mathbb{X}^{k-2} \cup \mathbb{M}^{k-2}) \cup \mathbb{M}^{k-1}\right) && \text{(b)}\\
&= \mathcal{U}(\mathbb{X}^0 \cup \mathbb{M}^0 \cup \mathbb{M}^1 \cup \mathbb{M}^2 \cup \cdots \cup \mathbb{M}^{k-1}) && \text{(c)}\\
&= \mathcal{U}\big(\mathbb{X}^0 \cup \mathbb{M}^0 \cup \mathcal{G}(\mathbb{M}^0) \cup \cdots \cup \mathcal{G}^{k-1}(\mathbb{M}^0)\big) && \text{(d)}\\
&= \mathcal{U}\left(\mathbb{X}^0 \cup \bigcup_{\ell=0}^{k-1} \mathcal{G}^{\ell}(\mathbb{M}^0)\right). && \text{(e)}
\end{aligned}$$

In the above equations, lines (a) and (b) are due to the property (**C2**) as defined in Sect. 4.1; line (c) follows from a simple induction; and line (d) is true because of Eq. (4) and the property (**C3**). □

A2. Proof of Lemma 2

Proof The incremental MF-applicable $\mathcal{A}$ on $G \oplus \Delta G$ starts from the previous computation status (see Algorithm 1). By Lemma 1, we have that $\hat{\mathbb{X}}^0 = \mathbb{X}^k = \mathcal{U}\left(\mathbb{X}^0 \cup \bigcup_{\ell=0}^{k-1} \mathcal{G}^{\ell}(\mathbb{M}^0)\right)$. It remains to verify $\hat{\mathbb{M}}^0$. Let $\mathbb{M}_v = \bigcup_{\ell=0}^{k-1} m_v^{\ell}$ be the messages received by v in the computation over G. Observe the following.

(1) Algorithm 1 generates two sets of messages, M_v^+ and M_v^-, in case that the edges related to v are *evolved*, *i.e.*, $\mathcal{G}(\mathbb{M}_v) \neq \hat{\mathcal{G}}(\mathbb{M}_v)$. By Algorithm 1, we have that $\mathcal{U}\left(\hat{\mathcal{G}}(\mathbb{M}_v) \ominus \mathcal{G}(\mathbb{M}_v)\right) = \mathcal{U}(M_v^+ \cup M_v^-)$.

(2) If the edges related to v are not evolved, we have that $\mathcal{G}(\mathbb{M}_v) = \hat{\mathcal{G}}(\mathbb{M}_v)$. By (**C1**) and (**C2**), it holds that:

$$\begin{aligned}
\mathcal{U}\left(\hat{\mathcal{G}}(\mathbb{M}_v) \ominus \mathcal{G}(\mathbb{M}_v)\right) &= \mathcal{U}\left(\hat{\mathcal{G}}(\mathbb{M}_v) \cup \mathcal{U}^- \circ \mathcal{U} \circ \mathcal{G}(\mathbb{M}_v)\right)\\
&= \mathcal{U}\left(\hat{\mathcal{G}}(\mathbb{M}_v) \setminus \mathcal{G}(\mathbb{M}_v)\right) = \mathcal{U}(\mathbf{0}).
\end{aligned}$$

That is, the messages in $\hat{\mathcal{G}}(\mathbb{M}_v)$ and $\mathcal{G}(\mathbb{M}_v)$ can be canceled out *w.r.t.* function $\mathcal{U}$. Thus there is no need to generate these messages in incremental computation over $G \oplus \Delta G$.

According to Algorithm 1, the initial messages $\hat{\mathbb{M}}^0$ for the incremental computation consist of $\mathcal{U}(M_v^+ \cup M_v^-)$ for vertices with evolved edges. By the above analysis, vertices without evolved edges contribute *empty* even if the corresponding messages are also generated. As a consequence, we can express $\hat{\mathbb{M}}^0$ as follows:

$$\begin{aligned}
\hat{\mathbb{M}}^0 &= \mathcal{U}\left(\bigcup_{v \in V} \hat{\mathcal{G}}(\mathbb{M}_v) \ominus \bigcup_{v \in V} \mathcal{G}(\mathbb{M}_v)\right)\\
&= \mathcal{U}\left(\hat{\mathcal{G}}\left(\bigcup_{v \in V} \bigcup_{\ell=0}^{k-1} m_v^{\ell}\right) \ominus \mathcal{G}\left(\bigcup_{v \in V} \bigcup_{\ell=0}^{k-1} m_v^{\ell}\right)\right)\\
&= \mathcal{U}\left(\hat{\mathcal{G}}\left(\bigcup_{\ell=0}^{k-1} \mathcal{G}^{\ell}(\mathbb{M}^0)\right) \ominus \mathcal{G}\left(\bigcup_{\ell=0}^{k-1} \mathcal{G}^{\ell}(\mathbb{M}^0)\right)\right).
\end{aligned}$$

□

A3. Proof of Lemma 3

Proof Since the batch algorithm $\mathcal{A}$ is MP-applicable, *i.e.*, condition (**C4**) holds, each state x_v^k of an unreset vertex v in $\mathbb{X}_N^k$ is determined by an effective message generated when running $\mathcal{A}$ on G, denoted as m_v^c. Hence $x_v^k = m_v^c$. Observe that the right-hand side of Eq. (8) refers to the result of executing $\mathcal{A}$ over the reserved graph $G_{\mathcal{R}}$ (see Lemma 1). Then if for each unreset vertex v, the effective message m_v^c is the same as the corresponding effective message generated for v when invoking $\mathcal{A}$ over $G_{\mathcal{R}}$, then Eq. (8) holds. To show this precondition, it suffices to prove that m_v^c is transmitted from another unreset vertex v' and (v', v) is not a deleted edge in ΔG. This is because the computation of $\mathcal{A}$ strictly follows the propagation of messages and it can be formally verified by induction on the rounds of the computation.

We next prove that each effective message m_v^c is sent from another unreset vertex via an edge that is not deleted by contradiction. Assume by contradiction that (i) a reset vertex v' sends m_v^c to the unreset v or (ii) m_v^c is transmitted along a deleted edge (v', v). (i) If v' has been reset, then v must also be a reset vertex as Algorithm 2 propagates $\perp$ along the paths formed by all effective messages, including m_v^c. (ii) If (v', v) is a deleted edge, then the state of v should also be reset since the deleted (v', v) initiates a cancelation message in Algorithm 2. Hence either case leads to a contradiction. The correctness of Eq. 8 follows. □

A4. Proof of Lemma 4

Proof When messages $\perp$ are propagated in Algorithm 2, the initial $\hat{\mathbb{X}}^0$ includes the current states for both reset and unreset vertices. By Lemma 3 and the definition of changed graph

$G_{\mathcal{C}}$, we have that $\hat{\mathbb{X}}^0 = \mathcal{U}\big(\mathbb{X}^0_{\mathcal{R}} \cup \bigcup_{\ell=0}^{k-1} \mathcal{G}^{\ell}_{\mathcal{R}}(\mathbb{M}^0_{\mathcal{R}})\big) \cup \mathbb{X}^0_{\mathcal{C}} = \mathcal{U}\left(\mathbb{X}^0_{\mathcal{R}} \cup \bigcup_{\ell=0}^{k} \mathbb{M}^{\ell}_{\mathcal{R}}\right) \cup \mathbb{X}^0_{\mathcal{C}}$.

We next analyze the initial compensation messages $\hat{\mathbb{M}}_0$ initiated by Algorithm 2. By its operations, we can see that each initial compensation message can be sent from either an unreset vertex in $V_{\mathcal{R}}$ or a reset vertex in $V_{\mathcal{C}}$. In light of this, we denote by $\hat{\mathbb{M}}^r$ and $\hat{\mathbb{M}}^c$ the collections of initial compensation messages sent from unreset and reset vertices, respectively.

Suppose that an unreset vertex v sends an initial compensation message to v'. Due to the logic of Algorithm 2 (line 4), v' must be a reset vertex or (v, v') is an evolved edge. Hence edge (v, v') cannot appear in the reserved graph $G_{\mathcal{R}}$ under both cases. Recall that M_v^+ denotes the initial compensation messages sent from v. Then for each unreset vertex $v \in V_R$, $M_v^+ = \mathcal{U} \circ \hat{\mathcal{G}}(x_v^k) \backslash \mathcal{U} \circ \mathcal{G}_{\mathcal{R}}(x_v^k)$. This expression also indicates that no actual message is created on unreset v if none of v's adjacent edges is evolved or covers reset vertices. Based on this observation and Lemma 3, we have

$$\begin{aligned}\hat{\mathbb{M}}^r &= \bigcup_{v \in V_{\mathcal{R}}} \big(\mathcal{U} \circ \hat{\mathcal{G}}(x_v^k) \setminus \mathcal{U} \circ \mathcal{G}_{\mathcal{R}}(x_v^k)\big) \\ &= \bigcup_{v \in V_{\mathcal{R}}} \bigcup_{\ell=0}^{k} \big(\mathcal{U} \circ \hat{\mathcal{G}}(m_v^{\ell}) \setminus \mathcal{U} \circ \mathcal{G}_{\mathcal{R}}(m_v^{\ell})\big) \\ &= \bigcup_{\ell=0}^{k} \big(\mathcal{U} \circ \hat{\mathcal{G}}(\mathbb{M}^{\ell}_{\mathcal{R}}) \setminus \mathcal{U} \circ \mathcal{G}_{\mathcal{R}}(\mathbb{M}^{\ell}_{\mathcal{R}})\big).\end{aligned}$$

Putting this and the fact that $\hat{\mathbb{M}}^c = \mathbb{M}^0_{\mathcal{C}}$ together, *i.e.,* the initial messages constructed at reset vertices are the same as their counterparts created in the changed graph, we have $\hat{\mathbb{M}}^0 = \mathbb{M}^0_{\mathcal{C}} \cup \bigcup_{\ell=0}^{k} \left(\mathcal{U} \circ \hat{\mathcal{G}}(\mathbb{M}^{\ell}_{\mathcal{R}}) \backslash \mathcal{U} \circ \mathcal{G}_{\mathcal{R}}(\mathbb{M}^{\ell}_{\mathcal{R}})\right)$. □

References

1. Acar, U.A.: Self-adjusting computation. Ph.D. thesis, CMU (2005)
2. Baluja, S., Seth, R., Sivakumar, D., Jing, Y., Yagnik, J., Kumar, S., Ravichandran, D., Aly, M.: Video suggestion and discovery for youtube: taking random walks through the view graph. In: WWW, pp. 895–904 (2008)
3. Bang-Jensen, J., Gutin, G.Z.: Digraphs-Theory, Algorithms and Applications, 2nd edn. Springer, Cham (2009)
4. Cai, Y., Giarrusso, P.G., Rendel, T., Ostermann, K.: A theory of changes for higher-order languages: incrementalizing λ-calculi by static differentiation. In: PLDI, pp. 145–155 (2014)
5. Chang, X., Liu, X., Wen, J., Li, S., Fang, Y., Song, L., Qi, Y.: Continuous-time dynamic graph learning via neural interaction processes. In: CIKM, pp. 145–154 (2020)
6. Europe-OSM. https://www.cise.ufl.edu/research/sparse/matrices/DIMACS10/europe_osm.html (2010)
7. Fan, W., Hu, C., Tian, C.: Incremental graph computations: Doable and undoable. In: SIGMOD, pp. 155–169 (2017)
8. Fan, W., Liu, M., Tian, C., Xu, R., Zhou, J.: Incrementalization of graph partitioning algorithms. PVLDB **13**(8), 1261–1274 (2020)
9. Fan, W., Tiao, C., Xu, R., Yin, Q., Yu, W., Zhou, J.: Incrementalizing graph algorithms. pp. 459–471 (2022)
10. Fan, W., Xu, J., Wu, Y., Yu, W., Jiang, J., Zheng, Z., Zhang, B., Cao, Y., Tian, C.: Parallelizing sequential graph computations. In: SIGMOD, pp. 495–510 (2017)
11. Feng, G., Ma, Z., Li, D., Chen, S., Zhu, X., Han, W., Chen, W.: Risgraph: A real-time streaming system for evolving graphs to support sub-millisecond per-update analysis at millions ops/s. In: SIGMOD, pp. 513–527 (2021)
12. Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. J. ACM **34**(3), 596–615 (1987)
13. Gong, S., Tian, C., Yin, Q., Yu, W., Zhang, Y., Geng, L., Yu, S., Yu, G., Zhou, J.: Automating incremental graph processing with flexible memoization. PVLDB **14**(9), 1613–1625 (2021)
14. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: Powergraph: distributed graph-parallel computation on natural graphs. In: OSDI, pp. 17–30 (2012)
15. Grädel, E., Kolaitis, P.G., Vardi, M.Y.: On the decision problem for two-variable first-order logic. Bull. Symb. Log. **3**(1), 53–69 (1997)
16. Guan, Z., Wu, J., Zhang, Q., Singh, A.K., Yan, X.: Assessing and ranking structural correlations in graphs. In: SIGMOD, pp. 937–948 (2011)
17. Hamilton, W.L., Ying, Z., Leskovec, J.: Inductive representation learning on large graphs. In: NIPS (2017)
18. Hammer, M.A., Khoo, Y.P., Hicks, M., Foster, J.S.: Adapton: composable, demand-driven incremental computation. In: PLDI, pp. 156–166 (2014)
19. Holm, J., de Lichtenberg, K., Thorup, M.: Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. J. ACM **48**(4), 723–760 (2001)
20. Jeh, G., Widom, J.: Simrank: a measure of structural-context similarity. In: KDD, pp. 538–543 (2002)
21. Jiang, X., Xu, C., Yin, X., Zhao, Z., Gupta, R.: Tripoline: generalized incremental graph processing via graph triangle inequality. In: EuroSys, pp. 17–32 (2021)
22. Katz, L.: A new status index derived from sociometric analysis. Psychometrika **18**(1), 39–43 (1953)
23. Kim, K., Seo, I., Han, W., Lee, J., Hong, S., Chafi, H., Shin, H., Jeong, G.: Turboflux: a fast continuous subgraph matching system for streaming graph data. In: SIGMOD, pp. 411–426 (2018)
24. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. In: ICLR (2017)
25. Li, R., Yu, J.X., Mao, R.: Efficient core maintenance in large dynamic graphs. TKDE **26**(10), 2453–2465 (2014)
26. Liu, Y.A.: Efficiency by incrementalization: an introduction. High. Order Symb. Comput. **13**(4), 289–313 (2000)
27. Luo, X., Liu, L., Yang, Y., Bo, L., Cao, Y., Wu, J., Li, Q., Yang, K., Zhu, K.Q.: Alicoco: Alibaba e-commerce cognitive concept net. In: SIGMOD, pp. 313–327 (2020)
28. Libgrape-Lite. https://github.com/alibaba/libgrape-lite
29. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: SIGMOD, pp. 135–146 (2010)
30. Mariappan, M., Che, J., Vora, K.: Dzig: Sparsity-aware incremental processing of streaming graphs. In: EuroSys, pp. 83–98 (2021)
31. Mariappan, M., Vora, K.: Graphbolt: Dependency-driven synchronous processing of streaming graphs. In: EuroSys, pp. 1–16 (2019)
32. Matijasevič, Y.V.: Diophantine representation of recursively enumerable predicates. In: Studies in Logic and the Foundations of Mathematics, vol. 63, pp. 171–177 (1971)
33. McCune, R.R., Weninger, T., Madey, G.: Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. ACM Comput. Surv. **48**(2), 1–39 (2015)

34. McGregor, A., Vorotnikova, S., Vu, H.T.: Better algorithms for counting triangles in data streams. In: PODS, pp. 401–411 (2016)
35. McSherry, F., Murray, D.G., Isaacs, R., Isard, M.: Differential dataflow. In: CIDR (2013)
36. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS, pp. 337–340 (2008)
37. Murray, D.G., McSherry, F., Isaacs, R., Isard, M., Barham, P., Abadi, M.: Naiad: a timely dataflow system. In: SOSP, pp. 439–455 (2013)
38. Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: bringing order to the web. Technical report, Stanford InfoLab (1999)
39. Pearl, J.: Reverend Bayes on inference engines: a distributed hierarchical approach. In: AAAI, pp. 129–138 (1982)
40. Rossi, R.A., Ahmed, N.K.: The network data repository with interactive graph analytics and visualization. In: AAAI (2015). http://networkrepository.com
41. Road-USA-Graph. https://www.cise.ufl.edu/research/sparse/matrices/DIMACS10/road_usa.html (2011)
42. Schieber, B., Vishkin, U.: On finding lowest common ancestors: simplification and parallelization. SIAM J. Comput. **17**(6), 1253–1262 (1988)
43. Sengupta, D., Sundaram, N., Zhu, X., Willke, T.L., Young, J., Wolf, M., Schwan, K.: Graphin: an online high performance incremental graph processing framework. In: Euro-Par, pp. 319–333 (2016)
44. Shi, X., Cui, B., Shao, Y., Tong, Y.: Tornado: a system for real-time iterative analysis over evolving data. In: SIGMOD, pp. 417–430 (2016)
45. Sukhbaatar, S., Szlam, A., Fergus, R.: Learning multiagent communication with backpropagation. In: NIPS (2016)
46. Size of Wikipedia (2020). https://en.wikipedia.org/wiki/Wikipedia:Size_of_Wikipedia
47. Tian, Y., Balmin, A., Corsten, S.A., Tatikonda, S., McPherson, J.: From "think like a vertex" to "think like a graph". PVLDB **7**(3), 193–204 (2013)
48. UK-2005. https://www.cise.ufl.edu/research/sparse/matrices/LAW/uk-2005.html (2005)
49. Valiant, L.G.: A bridging model for parallel computation. CACM **33**(8), 103–111 (1990)
50. Vora, K., Gupta, R., Xu, G.: Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In: ASPLOS, pp. 237–251 (2017)
51. Wang, Q., Zhang, Y., Wang, H., Geng, L., Lee, R., Zhang, X., Yu, G.: Automating incremental and asynchronous evaluation for recursive aggregate data processing. In: SIGMOD, pp. 2439–2454 (2020)
52. Xu, S., Zhang, H., Neubig, G., Dai, W., Kim, J.K., Deng, Z., Ho, Q., Yang, G., Xing, E.P.: Cavs: an efficient runtime system for dynamic neural networks. In: ATC, pp. 937–950 (2018)
53. Yang, J., Leskovec, J.: Defining and evaluating network communities based on ground-truth. In: ICDM, pp. 1–8 (2015)
54. Zakian, T.A.K., Capelli, L.A.R., Hu, Z.: Incrementalization of vertex-centric programs. In: IPDPS, pp. 1019–1029 (2019)
55. Zhang, Y., Gao, Q., Gao, L., Wang, C.: Priter: a distributed framework for prioritized iterative computations. In: SOCC, pp. 1–14 (2011)
56. Zhang, Y., Gao, Q., Gao, L., Wang, C.: Maiter: an asynchronous graph processing framework for delta-based accumulative iterative computation. TPDS **25**(8), 2091–2100 (2013)
57. Zhao, J., Yang, Y., Zhang, Y., Liao, X., Gu, L., He, L., He, B., Jin, H., Liu, H., Jiang, X., Yu, H.: Tdgraph: a topology-driven accelerator for high-performance streaming graph processing. In: ISCA, pp. 116–129 (2022)
58. Zheng, L., Li, Z., Li, J., Li, Z., Gao, J.: Addgraph: anomaly detection in dynamic graph using attention-based temporal GCN. In: IJCAI (2019)
59. Zhu, X., Chen, W., Zheng, W., Ma, X.: Gemini: A computation-centric distributed graph processing system. In: OSDI, pp. 301–316 (2016)