Automating Incremental Graph Processing with Flexible Memoization

Shufeng Gong[1], Chao Tian[2], Qiang Yin[2], Wenyuan Yu[2], Yanfeng Zhang[1],
Liang Geng[2], Song Yu[1], Ge Yu[1], Jingren Zhou[2]
Northeastern University[1] Alibaba Group[2]
{gongsf, yusong}@stumail.neu.edu.cn, {tianchao.tc, qiang.yq, wenyuan.ywy, guanyi.gl, jingren.zhou}@alibaba-inc.com,
{zhangyf, yuge}@mail.neu.edu.cn

ABSTRACT

The ever-growing amount of dynamic graph data demands efficient techniques of incremental graph processing. 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 incrementalize batch vertex-centric algorithms into their incremental counterparts as a whole, 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 15.93× on average (up to 147.14×) in efficiency.

PVLDB Reference Format:
Shufeng Gong, Chao Tian, Qiang Yin, Wenyuan Yu, Yanfeng Zhang, Liang Geng, Song Yu, Ge Yu, Jingren Zhou.Automating Incremental Graph Processing with Flexible Memoization. PVLDB, 14(9): 1613 - 1625, 2021.
doi:10.14778/3461535.3461550

1 INTRODUCTION

Most of the current graph systems are designed to perform computation over static graphs [14, 17, 30, 44, 51, 52, 54]. When the graph is updated with input changes, they have to reperform the entire computation on the new graph starting from scratch. Such recomputation is costly as real-life graphs easily have trillions of edges, *e.g.,* e-commerce graphs [29] and they are constantly changed, *e.g.,* the relationships between users and items [10].

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. In practice, real-world changes are typically small, *e.g.,*

Proceedings of the VLDB Endowment, Vol. 14, No. 9 ISSN 2150-8097.
doi:10.14778/3461535.3461550

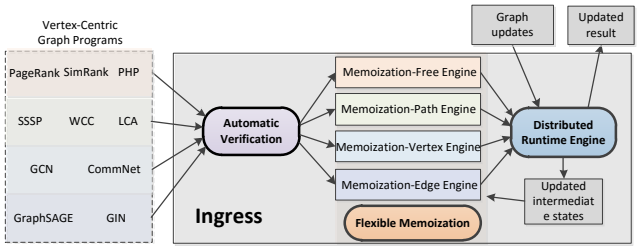


Figure 1: Overall structure of Ingress

English Wikipedia was expanded with on average less than 600 new articles per day out of 5 million articles during 2019 [5]. 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 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 [42], GraphIn [41], KickStarter [46] 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 [37] and single source shortest path (SSSP) [16].

While the existing incremental graph systems [31, 41, 42, 46] help eliminate redundant recomputation, they are limited by two major drawbacks. First, they need nontrivial user intervention, *e.g.,* GraphIn [41] and GraphBolt [31] ask users to manually deduce the incremental operators, and Tornado [42] and KickStarter [46] require users to make sure that the corresponding batch computation satisfies certain properties. Second, 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 satisfying the properties. Our new findings show that some incremental algorithms even do not employ any memoized intermediate states at all. However, checking the properties of the graph computations and choosing the best memoization policies bring heavy burdens to 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

Table 1: Performance comparison over dataset Twitter-2009

Algorithm	System	Time (s)	Space (GB)
PageRank	Ingress	6.67	0.31
	GraphBolt	59.35	13.66
SSSP	Ingress	0.59	0.6
	KickStarter	16.69	1.39

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 Figure 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.

Ingress features the followings that differ from previous systems.

Flexible memoization. Ingress supports a flexible memoization scheme and can perform the incrementalization, *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) [26], and GraphSAGE [20] with mean aggregator, respectively (see Figure 1). 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 incrementalizes generic batch vertex-centric algorithms into their incremental counterparts as a whole. There is no need to manually reshape the data structures or 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 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 [11], the satisfiability of the conditions can be automatically verified (Automatic Verification module in Figure 1). Putting this together with the four incrementalization engines that derive incremental algorithms with the selected memoization policies (Figure 1), Ingress makes the process of incrementalization transparent to users.

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.

High performance runtime. 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 Twitter-2009 that consists of 1.5 billion vertices and 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 8.89× and 28.28×, 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. We summarize our contributions as follows.

(1) A general framework for incrementalizing vertex-centric algorithms (Section 3). It models the operations of incremental computation in terms of the cancellation 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 (Section 4).

(3) The automation techniques for selecting appropriate memoization policies for incrementalization, as well as a distributed runtime engine to perform incremental graph computation (Section 5).

(4) An extensive evaluation of the incremental graph processing system Ingress, demonstrating its efficacy (Section 6).

2 PRELIMINARIES

We start with a review of basic notations for vertex-centric programming 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 models [17, 30], 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 [45], the computation of $\mathcal{A}$ are separated into super-steps. In each round i, $\mathcal{A}$ performs

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

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 [30].

In light of the simplicity and the distributed nature of the model, a large number of vertex-centric algorithms are proposed [33].

Example 1: We show three example vertex-centric algortihms.

(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 [52] can be represented as follows.

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

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 [52], this delta-based PageRank algorithm computes the PageRank scores for all vertices correctly.

(b) 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 works as follows.

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

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 can represent the shortest length of paths through v to w. The algorithm terminates when all shortest distances are fixed.

(c) GCN-Forward. Consider the GCN-forward [26]. 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 $\{W_1, \cdots, W_k\}$ 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.

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

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

Table 2: Summary of notations

Notation	Definition
$\mathcal{A}$, $\mathcal{A}_\Delta$	batch algorithm and incremental algorithm
$\mathcal{H}$, $\mathcal{U}$, $\mathcal{G}$	aggregation, update and propagation function, resp.
$\mathcal{U}^-$	the inverse function of $\mathcal{U}$
x_v^i	the state of vertex v at round i
m_v^i	the aggregated result of the messages sent to v at round i
M_v^{i-1}, M	the set of messages sent to v at round i, a set of messages
$m_{v,w}^{i-1}$	a single message sent from v to w in round i

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

- *Input*: A graph G, the (old) output $\mathcal{A}(G)$ 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 to 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 [6].

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.

Example 2: Consider running the delta-based PageRank algorithm of Example 1(a) on the graph G shown in Figure 2(a). Assume that a batch update ΔG to G removes the edge (A, C) and inserts the edge (C, A); and $G \oplus \Delta G$ is shown in Figure 2(b). 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 1(a)), 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. □

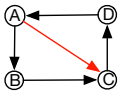
As such, the incremental algorithm first discovers all the invalid and missing messages. It then reperforms the computation on affected areas of $G \oplus \Delta G$ by generateing *cancellation messages* (resp. *compensation messages*) to undo (resp. replay) their effects.

It is a common practice to memoize previous computed (intermediate) results in incremental processing [6, 21], 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, 41], 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.

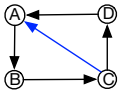
*(1) Memoization-free (*MF*).* This policy does not record any old message at all. Instead, the incremental algorithms should handle the effects of invalid and missing messages directly on the previous batch run's converged states, *i.e.,* final results. This is doable for a



(a). Graph G

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

(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	$\perp \rightarrow C$	$\emptyset$	$\perp \rightarrow D$	$\emptyset$
Compensation	$\emptyset$	$2 \rightarrow C$	$\emptyset$	$\emptyset$

(d). SSSP (source = A)

Figure 2: Sample graph and messages for PageRank **and** SSSP

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 cancellation 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, 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 cancellation 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 cancellation 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 cancellation (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^*=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 *cancellation 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 cancellation and compensation messages.

All the cancellation and compensation messages are shown in Figure 2(c). The incremental algorithm restarts the computation of PageRank (Example 1(a)) with these messages. As will be clear in Section 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 *cancellation message* $\perp$ to C, which indicates the invalidation of the effective message. It resets x_C to the initial state ∞. The cancellation 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 1(b)). 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 algorithms. 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 *cancellation message* $m_{A,C}^1 = -m_A^1 \bullet W_1$ to C in the initial round, to undo the effect of an *invalid* message $m_A^1 \bullet W_1$ transmitted during prior run. This is feasible since GCN-forward takes sum as $\mathcal{H}$. Upon receiving this, vertex C adjusts the cancellation message to $-\text{relu}(m_C^2) \bullet W_2$ and propagates it to D; it also sends a new compensation message $\text{relu}(\text{sum}(m_C^2, m_A^1)) \bullet W_2$ to D. These two represent the difference between the messages transmitted during the two runs. The algorithm also updates m_C^2 to $\text{sum}(m_C^2, m_A^1)$. Analogously, a *compensation message* $m_C^1 \bullet W_1$ 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 cancellation message $-\text{relu}(m_v^i) \bullet W_i$ and a compensation message $\text{relu}(\text{sum}(m_v^i, M_v^i)) \bullet W_i$ 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 $\text{sum}(m_v^i, M_v^{i-1})$. Finally, computing $\text{relu}(m_v^{K+1})$ can obtain the revised results for $G \oplus \Delta G$.

As observed in [20, 53], 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 item clicks 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 Section 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.

(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 Section 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 Section 4).

4 FLEXIBLE MEMOIZATION

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

4.1 Incrementalization via Memoization-Free

As discussed in Section 3, with the memoization-free (MF) policy, the deduced incremental algorithms should initiate two sets of messages, *i.e., cancellation and compensation messages* directly from the converged states of batch runs, which are needed to handle invalid and missing messages, respectively. In fact, 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 has three sub-conditions.

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

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

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 condition (**C1**), we can cancel the effects of invalid messages and compensate the missing messages without memoizing any intermediate states. It is obvious that aggregation function $\mathcal{H}$ and 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.

(C2) $\mathcal{U}(\{\mathcal{U}(M)\} \cup M') = \mathcal{U}(M \cup M')$
(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.,* $\mathcal{U}\circ\mathcal{G}\circ\mathcal{U}\circ\mathcal{G}\circ\mathcal{U}(M) = \mathcal{U}\circ\mathcal{G}\circ\mathcal{G}(M)$. It 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.,* $x^t_v = \mathcal{U}(\bigcup^t_{i=0} m^i_v)$. With this traceability, we do not store any intermediate vertex states.

If a vertex-centric batch algorithm $\mathcal{A}$ satisfies the above conditions, we say that $\mathcal{A}$ is MF-*applicable*.

Example 6: Since $\mathsf{sum}(M \setminus M') = \mathsf{sum}(M, \{-\mathsf{sum}(M')\})$, where M (resp. M') consists of real numbers, we know that 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 to directly adjust the vertex states under the above conditions.

Deducing messages. Suppose that G is updated with input changes ΔG. For each vertex v, we deduce two sets of messages.

(1) Cancellation messages. Denote by $w_1, \ldots, w_k$ the neighbors of v in G. Given 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 $P'_E(w, w_j)$ refer to 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 *cancellation messages* as

$$M^-_v = \{\mathcal{U}^- \circ \mathcal{U} \circ \mathcal{G}(*, m^*_v, P_E(v, w_j)) \mid \text{evolved } (v, w_j) \text{ in } G\} \quad (1)$$

where m^*_v is the aggregation of initial and all received messages. In fact, all the messages propagated from v to w_j in the batch run are $M_{v,w_j} = \bigcup^\infty_{i=0} \mathcal{G}(x^i_v, m^i_v, P_E(v, w_j)) = \bigcup^\infty_{i=0} \mathcal{G}(*, m^i_v, P_E(v, w_j))$, as the message generation does not depend on the vertex state x^i_v. Let all the messages received by w_j across iterations be $\bigcup^\infty_{i=0} M^i_{w_j} = M_{v,w_j} \cup M'$, where M' represents the messages from w_j's other neighbors. Observe that finally $m^*_{w_j} = \mathcal{U}(M_{v,w_j} \cup M')$. 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 $\mathcal{U}(\bigcup^\infty_{i=0} M^i_{w_j} \cup \{\mathcal{U}^- \circ \mathcal{U}(M_{v,w_j})\}) = \mathcal{U}(M')$. Thus it suffices to propagate $\mathcal{U}^- \circ \mathcal{U}(M_{v,w_j})$ from v to w_j. By conditions (**C2**) and (**C3**), we have that $\mathcal{U}(M_{v,w_j}) = \mathcal{U}(\bigcup^\infty_{i=0} \mathcal{G}(*, m^i_v, P_E(v, w_j))) = \mathcal{U}(\mathcal{G}(*, \mathcal{U}(\bigcup^\infty_{i=0} m^i_v), P_E(v, w_j)))$. As $m^*_v = \mathcal{U}(\bigcup^\infty_{i=0} m^i_v)$ in the end, $\mathcal{U}(M_{v,w_j}) = \mathcal{U} \circ \mathcal{G}(*, m^*_v, P_E(v, w_j))$ and the cancellation 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^*, we can deduce m^* by $m^* = \mathcal{U}^-(x^*_v, x^0_v)$ because $x^* = \mathcal{U}(x^0_v \cup m^*_v)$.

(2) Compensation messages. The set M^+_v of compensation messages can be computed as the dual of cancellation 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 by using the new edge properties *w.r.t.* $G \oplus \Delta G$ as follows:

$$M^+_v = \{\mathcal{U} \circ \mathcal{G}(*, m^*_v, P'_E(v, w_j)) \mid \text{evolved } (v, w_j) \text{ in } G \oplus \Delta G\} \quad (2)$$

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 descried above. For each evolved edge, it then initiates appropriate cancellation and compensation messages based on Equations (1) and (2) (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, updates ΔG, result $\{x^*_v\}_{v \in V}$ computed by $\mathcal{A}$.
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 $\quad m^*_v = \mathcal{U}^-(x^*_v, x^0_v)$;
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 cancellation and compensation messages as shown in Figure 2(c). Observe that both A and C pertain to involved edges. We first apply $\mathcal{U}^- \circ \mathcal{U} \circ \mathcal{G}$ over G and generates two cancellation 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 Figure 2(c)). □

The correctness of Algorithm 1 is verified by the following.

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

Proof sketch: Let $\mathbb{X}^i = \{x_v^i \mid v \in V\}$ (resp. $\mathbb{M}^i$) be the collections of vertex states (resp. messages) in the i-the round of computation. 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 run-time. We next analyze the initial vertex states $\hat{\mathbb{X}}^0$ and the initial messages $\hat{\mathbb{M}}^0$ generated by Algorithm 1 for incremental computation. With these, we are able to show that the incremental computation on $G \oplus \Delta G$ starting from $(\hat{\mathbb{X}}^0, \hat{\mathbb{M}}^0)$ converges to the same result as the computation from $(\mathbb{X}^0, \mathbb{M}^0)$, *i.e.,* they share an identical characterization. □

Apart from PageRank, many other algorithms are also MF-applicable, such as SimRank [23], Penalized Hitting Probability (PHP) [19], Katz Metric [24], Believe Propagation [38] and Adsorption [7], *i.e.,* they can be incrementalized with Algorithm 1.

Relative boundedness. The measure of *relative boundedness* is proposed in [12], which inspects whether the cost of an incremental algorithm $\mathcal{A}_\Delta$ can be expressed by a function of the differences of two runs of a batch algorithm $\mathcal{A}$. If so, it incurs necessary cost for incrementalizing $\mathcal{A}$. Since the incremental algorithms deduced with MF policy only propagate the differences of the messages, we have that Algorithm 1, denoted as $\mathcal{A}_\Delta$, is bounded relative to $\mathcal{A}$.

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 cancellation 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 is also composed 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,

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

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

Algorithm 2: Incrementalization via MP policy

Input: G, ΔG, $\{x_v^*\}_{v \in V}$ as in Algo. 1, effective messages M_E.
Output: Updated $\{x'_v\}_{v \in V}$ *w.r.t.* $G \oplus \Delta G$ and algorithm $\mathcal{A}$.

1 **foreach** $m_c \in M_E$ sent via a deleted (v, w) in ΔG **do**
2 initiate cancellation message $\perp$ to be sent from v to w;
3 propagate messages $\perp$ 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 has 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 of $\mathcal{A}$ with messages M_v^+ ($\forall v \in V$);

(2) The vertex-centric algorithm $\mathcal{A}$ is endowed with the traceability property, *i.e.,* it satisfies conditions (**C2**) and (**C3**) of Section 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. If a batch vertex-centric algorithm $\mathcal{A}$ satisfies conditions (**C2**)-(**C4**), we say that $\mathcal{A}$ is MP-*applicable*.

Example 8: The SSSP algorithm of Example 1(b) is MP-applicable. Obviously 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 cancellation 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) Cancellation messages. Each cancellation 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 (Section 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 cancellation 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 evolved edge. Finally the iterative computation of $\mathcal{A}$ continues with M_v^+ (line 6).

Algorithm 2 can correctly adjust the previous converged states.

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

Proof sketch: We first prove that after all cancellation messages are communicated, those previous converged states of unreset vertices coincide with the aggregated results of a *subset* of the correct messages *w.r.t.* $G \oplus \Delta G$. Hence the unreset vertices induce a *reserved subgraph* preserving the result of the batch computation. By analyzing the initial vertex states $\hat{\mathbb{X}}^0$ and compensation messages $\hat{\mathbb{M}}^0$ generated in Algorithm 2 in regards to the reserved subgraph, we next show 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 $G \oplus \Delta G$. □

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 algortihms, *e.g.,* Connected Components [8] and Lowest Common Ancestor [40].

4.3 Incrementalization via Memoization-Vertex

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

Conditions. The sufficient condition for applying memoization-vertex policy in incrementalizaion consists 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.

(2) The messages propagated in the i-th round is determined by vertex state alone, *i.e.,* the propagation function $\mathcal{G}$ can be written as

(C5) $m^i_{v,w}=\mathcal{G}(x^i_v, *, P_E(v,w))$, where $*$ is any aggregated result.

An algorithm satisfying the above conditions is MV-*applicable*.

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 $x^i_v \bullet W_i$. □

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.,* $m^i_{v,w}=\mathcal{G}(*, m^i_v, P_E(v,w))$. 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 cancellation 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, in the i-th round, each vertex v records a state m^i_v that represents the aggregated result after applying $\mathcal{H}$. Then cancellation and compensation messages are deduced in an iteration-wise manner.

Cancellation 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 $\mathcal{G}(x^i_v, m^i_v, P_E(v,w_j))=\mathcal{G}(x'^i_v, m'^i_v, P'_E(v,w_j))$, where x^i_v and x'^i_v are the vertex states of v *w.r.t.* G and $G \oplus \Delta G$, respectively. If such verification fails, we define the cancellation messages M^-_v as

$$M^-_v = \{\mathcal{H}^- \circ \mathcal{H} \circ \mathcal{G}(x^i_v, *, P_E(v,w_j)) \mid \text{evolved } (v,w_j) \text{ in } G\}. \quad (3)$$

As in MP policy, the correctness of M^-_v is warranted by (**C1**)-(**C2**).

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

$$M^+_v = \{\mathcal{G}(x'^i_v, *, P'_E(v,w_j)) \mid \text{evolved } (v,w_j) \text{ in } G \oplus \Delta G\}. \quad (4)$$

Incremental algorithm. We now outline the incremental algorithm deduced via the MV policy, which is referred to as Algorithm 3. 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 cancellation 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'^i_v *w.r.t.* each affected vertex v, by aggregating the recorded state (aggregated result) m^i_v with messages M^{i-1}_v received from v's neighbors. Here M^{i-1}_v, possibly empty, consists of cancellation and/or compensation messages. It then recovers the old vertex state x^i_v and derives the new state x'^i_v directly using update function $\mathcal{U}$. With x^i_v and x'^i_v in place, it generates and sends cancellation and compensation messages when needed, *i.e.,* applying Equations (3)-(4). It also replaces the recorded state m^i_v by m'^i_v for future use. The process terminates when all the previous rounds has 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 [43] are also MV-applicable. It is routine to verify the following by induction on the rounds of the iterative computation.

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

5 INGRESS

As a proof of concept, we design and implement the system Ingress.

5.1 Vertex-centric API

Following the vertex-centric model of Section 2, Ingress provides the API, shown in in Figure 3, 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. 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 Section 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

```
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;
}
```

Figure 3: The Vertex-centric API of Ingerss

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 Figure 4.

5.2 Automatic Memoization Policy Selection

As presented in Section 3, there exist multiple memoization policies for incrementalization, which lead to different space costs. Though their formal applicability conditions are provided in Section 4, it is nontrivial for non-expert users to choose the best-fit one. Ingress automatically selects the optimal memoization policy with the help of Satisfiability Modulo Theories (SMT) solver Z3 [11]. 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 Section 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

```
(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 [47], 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 be always 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;}
}
```

Figure 4: 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 [18], *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 Section 6).

5.3 Distributed Runtime Engine

The distributed runtime engine of Ingress is developed on top of libgrape-lite [4] (an open-source version of GRAPE [14]), 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. Following GRAPE, 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.

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 Section 4, for each of these affected vertices, Ingress will generate the cancellation 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.

6 EXPERIMENTAL STUDY

6.1 Experimental Setup

We evaluated Ingress with five incremental algorithms deduced from (i) two MP-applicable algorithms PageRank and Penalized Hitting Probability (PHP) [19], (ii) two MP-applicable algorithms SSSP and Connected Components (CC) [8], and (iii) one MV-applicable algorithm GCN-forward. PHP is used to measure the proximity between a given source s and any other vertex v. In batch PHP algorithm, $\mathcal{U}$=$\mathcal{H}$=sum and $\mathcal{G}(x_v^i, m_v^i, P_E(v,w))$=$\beta m_v^i P_E(v,w)$, where $0<\beta<1$ is a fixed parameter and $P_E(v,w)$ is edge weight. It satisfies conditions (**C1**)-(**C3**) and is MF-applicable. Batch CC algorithm 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.

Datasets and updates. Four real-life graphs were used (see Table 3), including social networks Twitter (TW) [39] and Friendster (FS) [49], web graph UK-2005 (UK) [1], and road networks Euro-Road (ER) [2] and US-Road (UR) [3]. 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. We also randomly generated changes to edge weights to test the performance in processing updates of weights.

Competitors. We compared Ingress with three state-of-the-art vertex-centric incremental systems, Torando [42], GraphBolt [31] and KickStarter [46]. We also implemented a competitor on top of libgrape-lite [4], denoted as IngressR, which reperforms the vertex-centric computation over the updated graph starting from scratch. It is used to validate the effectiveness of incremental processing.

In fact, KickStarter 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 because the initial states have impact on the output in these cases. GraphBolt 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 and Tornado (resp. KickStarter).

Graph	#Vertices	#Edges	Size
Twitter-2009 (TW) [39]	41,652,230	1,468,365,183	23.99GB
UK-2005 (UK) [1]	39,459,925	936,364,282	16,45GB
Euro-Road (ER) [2]	50,912,018	108,109,320	0.94GB
US-Road (UR) [3]	23,947,347	57,708,624	0.49GB
Friendster (FS) [49]	65,608,366	1,806,067,139	30.14GB

Table 3: Real-life graphs

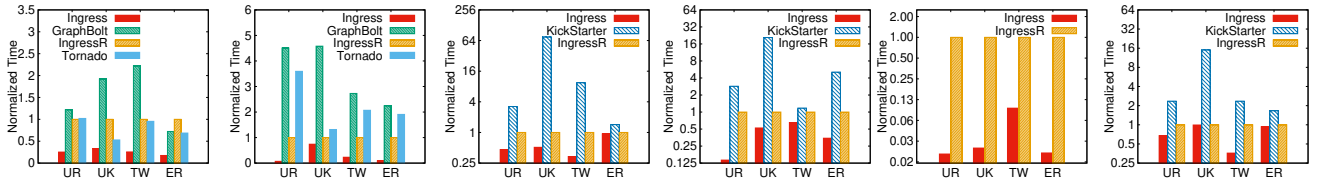
Environments. We used AliCloud ecs.r6.13xlarge instance (52vCPU, 384GB memory) for experiments conducted on single machine. To evaluate Ingress in a distributed environment, we adopted a cluster of 32 AliCloud ecs.r6.6xlarge instances (24vCPU, 192GB memory).

6.2 Overall Performance

We first evaluated the overall performance of Ingress, including both the response time and space cost, by comparing it with competitors. Note that GraphBolt and KickStarter can only run on a single machine with multi-core support, hence this set of experiments and the following ones in Sections 6.3 and 6.4 were conducted on a single machine. We fixed the size of either topological updates or weight changes as $|\Delta G|$=1%$|G|$, and used all real-life graphs except the FS dataset. When reporting response time, we omit the cost for policy selection since it can be done within 50 milliseconds for all tested algorithms.

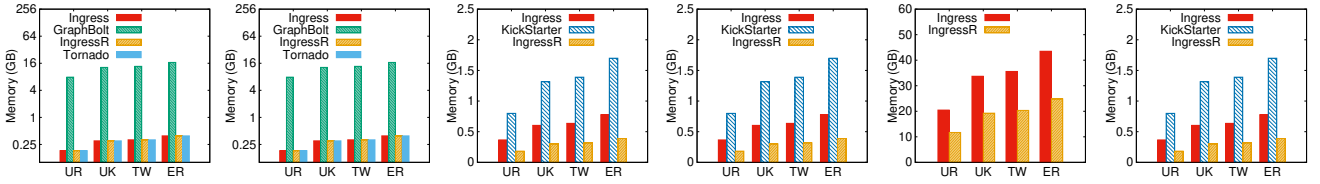
Response time. Figure 5 shows the *normalized* response time of each algorithm executed in different systems. Here the response time of IngressR is treated as the baseline, *i.e.,* IngressR finishes in unit time (*i.e.,* 1). In particular, Figure 5f reports the response time for processing edge weight updates, while the rest are for topological updates. We can see that Ingress outperforms others in all the cases and the improvement in handling weight changes is consistent with that for processing topological changes. More specifically, Ingress achieves 4.7×-66.83× (16.31× on average) speedup over GraphBolt, 1.52×-147.14× (23.95× on average) speedup over KickStarter, 1.44×-50.47× (11.65× on average) speedup over Tornado, and 1.07×-47.55× (10.84× on average) speedup over IngressR. Ingress is indeed very efficient, *e.g.,* taking only 11.76 seconds for PHP over the TW dataset, as opposed to 139 and 105 seconds by GraphBolt 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 6 depicts the space cost of each system, in which Figure 6f corresponds to the case of edge weight updates. We find that Ingress benefits greatly from its flexible memoization strategy. It is much more memory efficient than GraphBolt for PageRank and PHP, as shown in Figs. 6a-6b. This is because the MF engine of Ingress does not store any intermediate states, while GraphBolt maintains 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.



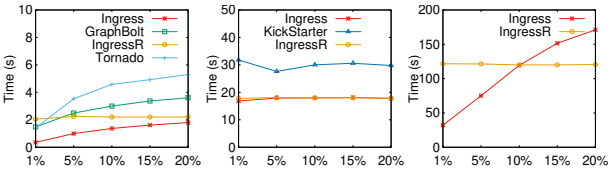
(a) PageRank (b) PHP (c) SSSP (d) CC (e) GCN-Forward (f) SSSP (weight updates)

Figure 5: Running time comparison



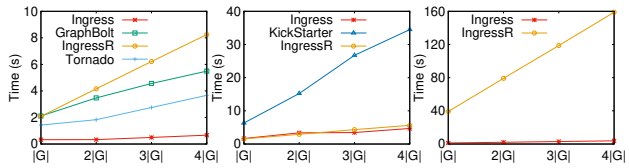
(a) PageRank (b) PHP (c) SSSP (d) CC (e) GCN-Forward (f) SSSP (weight updates)

Figure 6: Space cost comparison



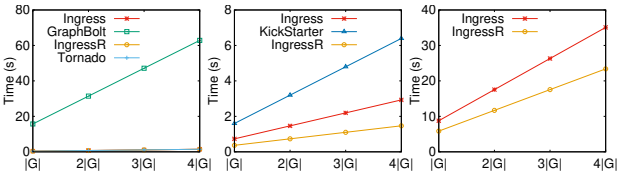
(a) PageRank (b) SSSP (c) GCN-Forward

Figure 7: Sensitivity to $|\Delta G|$



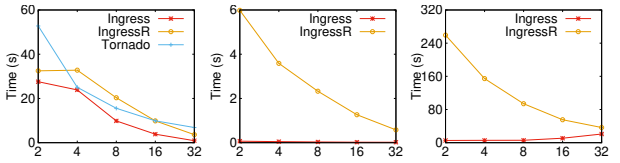
(a) PageRank (b) SSSP (c) GCN-Forward

Figure 8: Sensitivity to $|G|$ (time)



(a) PageRank (b) SSSP (c) GCN-Forward

Figure 9: Sensitivity to $|G|$ (space)



(a) PageRank (b) SSSP (c) GCN-Forward

Figure 10: Performance in distributed environment

However, Ingress is much faster than the other two (Figs. 5a-5b). 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 (Figs. 6c-6d and 6f). This is because KickStarter sorts additional level information for the paths [46]. For GCN-forward, Ingress adopts the MV engine, recording more intermediate results, hence takes more space than IngressR, *i.e.,* recomputation (Figure 6e).

6.3 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, Figure 7 shows the running time for the incremental computation of PageRank, SSSP and GCN-forward in different systems over the ER graph. We find the following.

(1) Almost all the incremental graph processing systems take longer to process larger input updates ΔG, as expected.

(2) 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 $|\Delta G|$.

(3) For PageRank, Ingress consistently outperforms other incremental processing systems. For SSSP, Ingress shows comparable performance with KickStarter since they rely on similar approach to achieve incremental computation (*i.e.,* the strategies adopted in MP policy (Section 4.2)). For GCN-forward, Ingress requires more running time than IngressR when $|\Delta G| \geq 10\%|G|$. Regarding this result, we analyze the vertex activation log and find that the input updates affect almost all the vertices, making the cost of incremental computation close to that of rerun (*i.e.,* IngressR). Due to additional state maintenance cost, Ingress spends more time than IngressR.

(4) Ingress is very effective in the incremental computation of PageRank. In fact, it takes less 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 $|\Delta G|$. We omit the detailed space costs for the lack of space.

6.4 Sensitivity to Graph Sizes

The performance of incremental graph computation is obviously expected to be sensitive to $|\Delta G|$, but the sensitivity to $|G|$ is not that clear. To investigate this, we conducted experiments to evaluate the impact of the size $|G|$ of the original graph. 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 $|\Delta G| = 0.23M$, *i.e.,* 0.23 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 $4|G|$, Figures 8 and 9 report the running time and space cost of different systems, respectively.

(1) Compared with IngressR that conducts recomputation, the response time of incremental systems Ingress, GraphBolt and KickStarter are less sensitive to the increase of $|G|$. This is because their time complexity 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 fast increasing rate.

(2) Although the increasing rates of response time of Ingress, GraphBolt and KickStarter are similar for running different algorithms, their space costs are very different when $|G|$ increases. Thanks to our memoization-free (MF) technique, Ingress shows substantial superiority over GraphBolt on space cost. Ingress is also more space efficient than KickStarter with its MP policy. We find that KickStarter uses more space to maintain the dependency information than Ingress. These are consistent with the results in Section 6.2. In practice, the user can benefit more from the 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.

6.5 Distributed Runtime Performance

We finally 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, Figure 10 shows the response time of different system. 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 workers increase 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.

7 RELATED WORK

Incremental graph processing systems. There have been systems developed for incremental graph processing, *e.g.,* [31, 41, 42, 46]. Tornado [42] 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 [41] incrementally handles dynamic graphs through fixed-sized batches. KickStarter [46], RisGraph [15] 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. Apart from these, [50] 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 [36], 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.,* [6, 9, 28], 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 [42, 46]. (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 [48] 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 [12], strongly connected components [22], subgraph isomorphism [25], k-cores [27], graph partitioning [13] 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.

8 CONCLUSION

We have proposed Ingress, a system to incrementalize vertex-centric algorithms with optimized memory consumption. Ingress incorporates a framework with four memoization policies to deal with various vertex-centric algorithms. We have identified sufficient conditions for the applicability of the memoization policies. We have also shown that Ingress can largely automate the incrementalization based on this framework. Our experimentally study verifies that Ingress is a promising tool for incremental graph processing.

One topic for future work is to extend Ingress to support the incrementalization of graph-centric algorithms. Another topic is to incrementalize neural network training.

ACKNOWLEDGMENTS

This work was supported by the National Key R&D Program of China (2018YFB1003404), National Natural Science Foundation of China (62072082, U1811261), Key R&D Program of Liaoning Province (2020JH2/10100037), and a research grant from Alibaba Group through Alibaba Innovative Research (AIR) Program and CCF-Huawei Database Innovation Research Funding.

REFERENCES

[1] 2005. uk-2005. *https://www.cise.ufl.edu/research/sparse/matrices/LAW/uk-2005.html.*

[2] 2010. europe-osm. *https://www.cise.ufl.edu/research/sparse/matrices/DIMACS10/europe_osm.html.*

[3] 2011. road-usa-graph. *https://www.cise.ufl.edu/research/sparse/matrices/DIMACS10/road_usa.html.*

[4] 2020. libgrape-lite. *https://github.com/alibaba/libgrape-lite.*

[5] 2020. Size of Wikipedia. *https://en.wikipedia.org/wiki/Wikipedia:Size_of_Wikipedia.*

[6] Umut A. Acar. 2005. *Self-Adjusting Computation.* Ph.D. Dissertation. CMU.

[7] Shumeet Baluja, Rohan Seth, D. Sivakumar, Yushi Jing, Jay Yagnik, Shankar Kumar, Deepak Ravichandran, and Mohamed Aly. 2008. Video suggestion and discovery for youtube: taking random walks through the view graph. In *WWW.*

[8] Jørgen Bang-Jensen and Gregory Z. Gutin. 2009. *Digraphs - Theory, Algorithms and Applications, Second Edition.* Springer.

[9] Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. 2014. A theory of changes for higher-order languages: incrementalizing λ-calculi by static differentiation. In *PLDI.*

[10] Xiaofu Chang, Xuqin Liu, Jianfeng Wen, Shuang Li, Yanming Fang, Le Song, and Yuan Qi. 2020. Continuous-Time Dynamic Graph Learning via Neural Interaction Processes. In *CIKM.*

[11] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS.*

[12] Wenfei Fan, Chunming Hu, and Chao Tian. 2017. Incremental Graph Computations: Doable and Undoable. In *SIGMOD.*

[13] Wenfei Fan, Muyang Liu, Chao Tian, Ruiqi Xu, and Jingren Zhou. 2020. Incrementalization of Graph Partitioning Algorithms. *PVLDB* 13, 8 (2020), 1261–1274.

[14] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, Jiaxin Jiang, Zeyu Zheng, Bohan Zhang, Yang Cao, and Chao Tian. 2017. Parallelizing Sequential Graph Computations. In *SIGMOD.*

[15] Guanyu Feng, Zixuan Ma, Daixuan Li, Xiaowei Zhu, Yanzheng Cai, Wentao Han, and Wenguang Chen. 2020. RisGraph: A Real-Time Streaming System for Evolving Graphs. *arXiv preprint arXiv:2004.00803* (2020).

[16] Michael L. Fredman and Robert Endre Tarjan. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM* 34, 3 (1987), 596–615.

[17] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI.*

[18] Erich Grädel, Phokion G. Kolaitis, and Moshe Y. Vardi. 1997. On the decision problem for two-variable first-order logic. *Bull. Symb. Log.* 3, 1 (1997), 53–69.

[19] Ziyu Guan, Jian Wu, Qing Zhang, Ambuj K. Singh, and Xifeng Yan. 2011. Assessing and ranking structural correlations in graphs. In *SIGMOD.*

[20] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *NIPS.*

[21] Matthew A. Hammer, Yit Phang Khoo, Michael Hicks, and Jeffrey S. Foster. 2014. Adapton: composable, demand-driven incremental computation. In *PLDI.*

[22] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. 2001. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM* 48, 4 (2001), 723–760.

[23] Glen Jeh and Jennifer Widom. 2002. SimRank: a measure of structural-context similarity. In *KDD.*

[24] Leo Katz. 1953. A new status index derived from sociometric analysis. *Psychometrika* 18, 1 (1953), 39–43.

[25] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. 2018. TurboFlux: A Fast Continuous Subgraph Matching System for Streaming Graph Data. In *SIGMOD.*

[26] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *ICLR.*

[27] Rong-Hua Li, Jeffrey Xu Yu, and Rui Mao. 2014. Efficient Core Maintenance in Large Dynamic Graphs. *TKDE* 26, 10 (2014), 2453–2465.

[28] Yanhong A. Liu. 2000. Efficiency by Incrementalization: An Introduction. *High. Order Symb. Comput.* 13, 4 (2000), 289–313.

[29] Xusheng Luo, Luxin Liu, Yonghua Yang, Le Bo, Yuanpeng Cao, Jinghang Wu, Qiang Li, Keping Yang, and Kenny Q. Zhu. 2020. AliCoCo: Alibaba E-commerce Cognitive Concept Net. In *SIGMOD.*

[30] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *SIGMOD.*

[31] Mugilan Mariappan and Keval Vora. 2019. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In *EuroSys.*

[32] Yu V Matijasevič. 1971. Diophantine representation of recursively enumerable predicates. In *Studies in Logic and the Foundations of Mathematics.* Vol. 63. 171–177.

[33] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing. *ACM Comput. Surv.* 48, 2 (2015), 1–39.

[34] Andrew McGregor, Sofya Vorotnikova, and Hoa T. Vu. 2016. Better Algorithms for Counting Triangles in Data Streams. In *PODS.*

[35] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *CIDR.*

[36] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *SOSP.*

[37] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

[38] Judea Pearl. 1982. Reverend Bayes on Inference Engines: A Distributed Hierarchical Approach. In *AAAI.*

[39] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI. http://networkrepository.com.*

[40] Baruch Schieber and Uzi Vishkin. 1988. On Finding Lowest Common Ancestors: Simplification and Parallelization. *SIAM J. Comput.* 17, 6 (1988), 1253–1262.

[41] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L Willke, Jeffrey Young, Matthew Wolf, and Karsten Schwan. 2016. Graphin: An online high performance incremental graph processing framework. In *Euro-Par.*

[42] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. 2016. Tornado: A system for real-time iterative analysis over evolving data. In *SIGMOD.*

[43] Sainbayar Sukhbaatar, Arthur Szlam, and Rob Fergus. 2016. Learning Multiagent Communication with Backpropagation. In *NIPS.*

[44] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From "Think Like a Vertex" to "Think Like a Graph". *PVLDB* 7, 3 (2013), 193–204.

[45] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *CACM* 33, 8 (1990), 103–111.

[46] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *ASPLOS.*

[47] Qiange Wang, Yanfeng Zhang, Hao Wang, Liang Geng, Rubao Lee, Xiaodong Zhang, and Ge Yu. 2020. Automating Incremental and Asynchronous Evaluation for Recursive Aggregate Data Processing. In *SIGMOD.*

[48] Shizhen Xu, Hao Zhang, Graham Neubig, Wei Dai, Jin Kyu Kim, Zhijie Deng, Qirong Ho, Guangwen Yang, and Eric P. Xing. 2018. Cavs: An Efficient Runtime System for Dynamic Neural Networks. In *ATC.*

[49] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. In *ICDM.*

[50] Timothy A. K. Zakian, Ludovic A. R. Capelli, and Zhenjiang Hu. 2019. Incrementalization of Vertex-Centric Programs. In *IPDPS.*

[51] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. 2011. PrIter: A distributed framework for prioritized iterative computations. In *SOCC.*

[52] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. 2013. Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation. *TPDS* 25, 8 (2013), 2091–2100.

[53] Li Zheng, Zhenpeng Li, Jian Li, Zhao Li, and Jun Gao. 2019. AddGraph: Anomaly Detection in Dynamic Graph Using Attention-based Temporal GCN. In *IJCAI.*

[54] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *OSDI.*