Incrementalizing Graph Algorithms

Wenfei Fan[1,2], Chao Tian[3], Ruiqi Xu[1], Qiang Yin[3], Wenyuan Yu[3], Jingren Zhou[3]
[1]University of Edinburgh [2]Shenzhen Institute of Computing Sciences [3]Alibaba Group
{wenfei@inf.,ruiqi.xu@}ed.ac.uk,{tianchao.tc,qiang.yq,wenyuan.ywy,jingren.zhou}@alibaba-inc.com

ABSTRACT

Incremental algorithms are important to dynamic graph analyses, but are hard to write and analyze. Few incremental graph algorithms are in place, and even fewer offer performance guarantees.

This paper approaches this by proposing to incrementalize existing batch algorithms. We identify a class of incrementalizable algorithms abstracted in a fixpoint model. We show how to deduce an incremental algorithm $\mathcal{A}_\Delta$ from such an algorithm $\mathcal{A}$. Moreover, $\mathcal{A}_\Delta$ can be made *bounded relative to* $\mathcal{A}$, *i.e.,* its cost is determined by the sizes of changes to graphs and changes to the affected area that is necessarily checked by batch algorithm $\mathcal{A}$. We provide generic conditions under which a deduced algorithm $\mathcal{A}_\Delta$ warrants to be correct and relatively bounded, by adopting the same logic and data structures of $\mathcal{A}$, at most using timestamps as an additional auxiliary structure. Based on these, we show that a variety of graph-centric algorithms can be incrementalized with relative boundedness. Using real-life and synthetic graphs, we experimentally verify the scalability and efficiency of the incrementalized algorithms.

CCS CONCEPTS

• **Mathematics of computing → Graph algorithms**.

KEYWORDS

incrementalization; boundedness; fixpoint algorithm

ACM Reference Format:
Wenfei Fan, Chao Tian, Ruiqi Xu, Qiang Yin, Wenyuan Yu and Jingren Zhou. 2021. Incrementalizing Graph Algorithms. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3452796

1 INTRODUCTION

When we talk about graph algorithms for a class $\mathcal{Q}$ of queries, we typically refer to *batch algorithms* for $\mathcal{Q}$. A batch algorithm takes as input a query $Q \in \mathcal{Q}$ and a graph G, and computes the answer $Q(G)$ to Q in G. As an example, consider the single-source shortest path problem (SSSP). Our familiar Dijkstra's algorithm [25] for SSSP is a batch algorithm that, given a weighted directed graph G and a node s in G as query Q, computes the set $Q(G)$ of the shortest distances from source s to all the other nodes in graph G.

SIGMOD '21, June 20–25, 2021, Virtual Event, China
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3452796

Batch algorithm are developed for static graphs. However, graphs in the real world are often *dynamic, i.e.,* they are constantly updated. Suppose now that graph G is updated by ΔG, *e.g.,* a sequence of edge insertions or deletions. We want to compute $Q(G \oplus \Delta G)$, *i.e.,* the shortest distances from v in the updated graph, where $G \oplus \Delta G$ applies changes ΔG to G. A brute-force approach is to recompute $Q(G \oplus \Delta G)$ with Dijkstra's algorithm starting from scratch.

Another approach is to develop an *incremental algorithm* $\mathcal{A}_\Delta$. Given a query Q, a graph G, updates ΔG to G and the old output $Q(G)$, $\mathcal{A}_\Delta$ computes changes ΔO to $Q(G)$ such that

$$Q(G \oplus \Delta G) = Q(G) \oplus \Delta O.$$

That is, $\mathcal{A}_\Delta$ computes new output $Q(G \oplus \Delta G)$ by finding changes ΔO to $Q(G)$, by reusing old computation of $Q(G)$ as much as possible.

The need. The need for incremental algorithms is evident.

(1) Real-life graphs are often big and constantly updated, *a.k.a.* the volume and velocity of big data. Graph queries are costly on large-scale graphs. Moreover, we often need to repeatedly run queries of *e.g.,* SSSP, graph simulation [26], depth-first search [43], connectivity [12] and local clustering coefficient [47] for e-commerce recommendation [34], road network analysis [49] and anomaly detection [53] when graphs are updated. It is too costly to run a batch algorithm starting from scratch in response to every update.

(2) When changes ΔG to a graph G are small, incremental algorithms $\mathcal{A}_\Delta$ typically work better than batch algorithms $\mathcal{A}$ that recompute $Q(G \oplus \Delta G)$ [22, 23]. This is because the cost of incremental computation is often determined by the size of the area affected by ΔG [20, 23, 39], not by $|G|$. In the real world, updates are typically small and diverse. For instance, various user operations on e-commerce platform, *e.g.,* item clicking, buying and refunding trigger millions of edge insertions and deletions everyday [53] on transaction graphs of billion scale [54].

(3) Incremental computation is a critical step of some graph systems, *e.g.,* the intermediate consequence operator in GRAPE [8, 24]. It effectively reduces the cost of iterative computation.

Challenges. No matter how desirable, incremental graph algorithms are hard to write and analyze. There is no systematic method for developing incremental algorithms yet. Moreover, the complexity analysis of incremental algorithms departs from its batch algorithm counterpart. It is more sensitive to the size of the areas affected by the updates [20, 39], not to the size $|G|$ of possibly big graph G.

In light of these challenges, while a large number of batch graph algorithms have been developed, few incremental graph algorithms are in place, and even fewer can provably guarantee that they outperform their batch counterparts when ΔG are small [20].

Is it possible to have a systematic method for developing incremental graph algorithms with performance guarantees?

Incrementalization. This paper studies the incrementalization of batch graph-centric algorithms.

Problem	Time (seconds)		
	Batch $\mathcal{A}$	Competitor	Deduced $\mathcal{A}_\Delta$
SSSP	4.57s (Dijkstra [25])	1.56s (DynDij [17])	0.88s
Sim	4.86s ($\mathsf{Sim}_{\mathsf{fp}}$ [26])	1.03s (IncMatch [23])	0.98s
LCC	78.1s ($\mathsf{LCC}_{\mathsf{fp}}$ [42])	18.6s (DynLCC [19])	12.0s

Table 1: Performance of incrementalized algorithms

(1) A systematic method. We propose to incrementalize existing batch graph algorithms. For a query class $\mathcal{Q}$, we pick a batch algorithm $\mathcal{A}$ that has been verified effective after years of practice. We deduce an incremental algorithm $\mathcal{A}_\Delta$ from $\mathcal{A}$, by *reusing the original logic and data structures of* $\mathcal{A}$ as much as possible. That is, we propose to deduce an incremental $\mathcal{A}_\Delta$ from a well-studied batch $\mathcal{A}$, rather than to design $\mathcal{A}_\Delta$ ad hoc starting from scratch.

Moreover, we deduce $\mathcal{A}_\Delta$ with the following guarantees.

(2) Correctness. For any query $Q \in \mathcal{Q}$, graph G and updates ΔG to G,

$$Q(G \oplus \Delta G) = \mathcal{A}(Q, G) \oplus \mathcal{A}_\Delta(Q, G, Q(G), \Delta G),$$

where $Q(G) = \mathcal{A}(Q, G)$ and $\mathcal{A}_\Delta(Q, G, Q(G), \Delta G)$ computes changes to $Q(G)$ in response to ΔG. That is, the deduced $\mathcal{A}_\Delta$ yields the same output as re-running $\mathcal{A}$ starting from scratch.

(3) Boundedness. We ensure that $\mathcal{A}_\Delta$ is *bounded relative to* $\mathcal{A}$ [20]. That is, the size of the data inspected by $\mathcal{A}_\Delta$ is a function in the sizes $|Q|$, $|\Delta G|$ and $|\mathsf{AFF}|$, not in (possibly big) $|G|$, where AFF denotes *the affected area* that is necessarily checked by $\mathcal{A}$ in response to ΔG. Intuitively, $|\Delta G|$ and $|\mathsf{AFF}|$ are the inherent updating cost of incrementalizing $\mathcal{A}$. When $|\Delta G|$ is small, $|\mathsf{AFF}|$ is often small as well, and hence $\mathcal{A}_\Delta$ is often more efficient than $\mathcal{A}$.

(4) Effectiveness. As proof of concept (PoC), it has been shown that effective incremental graph partitioners can be deduced from existing partitioning algorithms [21]. Moreover, compared with existing incremental algorithms in [17, 19, 23], on graphs with 73.7 million nodes and edges and 4% updates, Table 1 shows the performance of deduced $\mathcal{A}_\Delta$ for (a) SSSP; (b) graph simulation (Sim) that finds the maximum similarity relation between a graph pattern and a graph; and (c) local clustering coefficient (LCC) that measures the degree to which the neighbors of each node in a graph form a clique. One can see that the deduced incremental algorithms perform even better than existing fine-tuned competitors (see Section 6 for more).

Contributions & organization. We propose an approach to developing incremental graph algorithms.

(1) Incrementalization (Section 3). We identify a class of *incrementalizable* graph algorithms that can be modeled as fixpoint computations. We show how to deduce an incremental algorithm $\mathcal{A}_\Delta$ from such a batch algorithm $\mathcal{A}$, by *reusing the original logic and data structures of* $\mathcal{A}$ as much as possible. Moreover, we show that algorithm $\mathcal{A}_\Delta$ deduced from each $\mathcal{A}$ is correct, even without the use of auxiliary data structures, referred to as *deducible* from $\mathcal{A}$.

(2) Performance guarantees (Section 4). We identify generic conditions under which an incremental algorithm $\mathcal{A}_\Delta$ is *weakly deducible* from $\mathcal{A}$, *i.e.,* the deduced $\mathcal{A}_\Delta$ uses at most timestamp as auxiliary structures, and guarantees to be bounded relative to $\mathcal{A}$. We also provide guidelines for such incrementalization. These allow us to incrementalize existing batch algorithms with provable bounds.

(3) PoC (Section 5). We show that a variety of batch algorithms are incrementalizable, *e.g.,* algorithms for SSSP [25], Sim [26], LCC [42], and graph connectivity (CC) [12] that finds all connected components of a given graph, and depth-first search (DFS) [43] that traverses a given graph in a depth-first manner. We show that their incremental algorithms are either deducible (SSSP, DFS, LCC) or weakly deducible (CC, Sim). Moreover, all the deduced incremental algorithms are bounded relative to their batch counterparts.

(4) Empirical study (Section 6). Using real-life and synthetic graphs, we experimentally verify the following. On average, (a) our incrementalized $\mathcal{A}_\Delta$ for SSSP (resp. CC, Sim, DFS, LCC) is 20.1 (resp. 7.7, 39.6, 3.1, 18.7) times faster than its batch counterpart $\mathcal{A}$ when $|\Delta G| = 1\%|G|$. (b) When processing batch updates of edge insertions and deletions, these deduced $\mathcal{A}_\Delta$'s consistently outperform existing fine-tuned incremental methods, *e.g.,* by 1.8, 38.9, 1.2, 4.4, 2.7 times, respectively, when $|\Delta G|=1\%|G|$. (c) Their space costs are comparable to those of $\mathcal{A}$'s and competitors. (d) They scale well with $|G|$, *e.g.,* the one for SSSP takes 12.44s when $|G| = 2.2$ billion and $|\Delta G| = 1\%|G|$, as opposed to 99.9s by Dijkstra's algorithm.

Notation	Definition
$\mathcal{Q}$, Q	a class of graph queries, query $Q \in \mathcal{Q}$
G, ΔG	a directed or undirected graph, updates to graph
$\mathcal{A}$, $\mathcal{A}_\Delta$	fixpoint algorithm and its incremental algorithm
Φ	the class of fixpoint algorithms
$f_\mathcal{A}$ (resp. $f_{\mathcal{A}_\Delta}$)	step function of algorithm $\mathcal{A}$ (resp. $\mathcal{A}_\Delta$)
h	initial scope function
$D_\mathcal{A}, H_\mathcal{A}, S_\mathcal{A}$ (resp. $D_{\mathcal{A}_\Delta}, H_{\mathcal{A}_\Delta}, S_{\mathcal{A}_\Delta}$)	status, scope and data structure of $\mathcal{A}$ (resp. $\mathcal{A}_\Delta$)
x_i, σ_{x_i}	a status variable and its logical statement
f_{x_i}, Y_{x_i}	update function for variable x_i and its input set
$\sigma_\mathcal{A}$	invariant of algorithm $\mathcal{A}$
C_{x_i}, $<_C$	anchor set for x_i, an order of status variables

Table 2: Summary of notations

2 PRELIMINARIES

We first review basic notations of incremental algorithms.

Graphs. We consider graphs $G = (V, E, L)$, directed or undirected, where (1) V is a finite set of nodes; (2) $E \subseteq V \times V$ is a set of edges; and (3) each node v in V (resp. edge $e \in E$) carries a label $L(v)$ (resp. $L(e)$), indicating its content as in social networks and property graphs.

Graph algorithms. Consider a class $\mathcal{Q}$ of graph queries, *e.g.,* SSSP.

Batch algorithms. A *batch algorithm* $\mathcal{A}$ for $\mathcal{Q}$ takes as input a query $Q \in \mathcal{Q}$ and a graph G. It computes answers $Q(G)$ to Q in G, *i.e.,* $Q(G) = \mathcal{A}(Q, G)$. Its (combined) complexity is measured by a function in the sizes $|Q|$ of query Q and $|G|$ of the entire graph G [9].

Incremental algorithms. To simplify the discussion, we consider edge updates, *i.e.,* edge insertions and deletions. Vertex updates are a dual of edge updates [29], and will be elaborated in Section 4.

A *unit update* is either an edge insertion or an edge deletion. Previous work [39, 40] considers unit updates ΔG to study what an incremental algorithm has to do for different types of ΔG. As will be seen in Section 4, our correctness and boundedness results hold on *batch updates* ΔG, *i.e.,* sequences of unit updates.

In contrast to batch algorithms, an incremental algorithm $\mathcal{A}_\Delta$ for $\mathcal{Q}$ takes as input a query $Q \in \mathcal{Q}$, a graph G, old output $Q(G)$ and updates ΔG to G. It computes changes to $Q(G)$ such that

$$Q(G \oplus \Delta G) = Q(G) \oplus \mathcal{A}_\Delta(Q, G, Q(G), \Delta G).$$

We refer to this equation as *the correctness of* $\mathcal{A}_\Delta$.

Relative boundedness. Consider a batch algorithm $\mathcal{A}$ for a query class $\mathcal{Q}$. For a query Q in $\mathcal{Q}$ and a graph G, denote by $G_{(\mathcal{A},Q)}$ the data accessed by $\mathcal{A}$ for computing $Q(G)$, including the auxiliary structure used by $\mathcal{A}$. For updates ΔG to G, denote by AFF the difference between $(G\oplus\Delta G)_{(\mathcal{A},Q)}$ and $G_{(\mathcal{A},Q)}$, *i.e.,* the difference in the data inspected by $\mathcal{A}$ for computing $\mathcal{A}(Q, G\oplus\Delta G)$ and $\mathcal{A}(Q, G)$.

An incremental algorithm $\mathcal{A}_\Delta$ for $\mathcal{Q}$ is *bounded relative* to $\mathcal{A}$ [20] if for any query Q in $\mathcal{Q}$, graph G and updates ΔG to G, the size of the data checked by $\mathcal{A}_\Delta$ can be expressed as a function of the sizes $|Q|$, $|\Delta G|$ and $|\mathsf{AFF}|$. Here AFF includes changes ΔO to $Q(G)$.

Intuitively, $|\mathsf{AFF}|$ indicates the affected area by ΔG relative to $\mathcal{A}$, *i.e.,* the necessary cost for any possible incrementalization of $\mathcal{A}$.

Another criterion for measuring the effectiveness of incremental algorithms was proposed in [39, 44]. We say that $\mathcal{A}_\Delta$ is *bounded* if its cost can be expressed as a function of $|Q|$ and $|\mathsf{CHANGED}| = |\Delta G| + |\Delta O|$ [39, 44], *i.e.,* the size of changes to the input graph and the output. Unfortunately, this standard of boundedness is too strong, and few graph algorithms in practice are bounded [20].

Notations of this paper are summarized in Table 2.

3 INCREMENTALIZABLE ALGORITHMS

In this section, we first identify a class of batch graph algorithms that are "incrementalizable". We then show that we can deduce correct incremental algorithms from such batch algorithms.

A fixpoint model. For a query class $\mathcal{Q}$, a batch algorithm $\mathcal{A}$ often works as follows. Given a query $Q \in \mathcal{Q}$ and a graph G as input, (a) it builds data structures $S_\mathcal{A}$, which associates status variables with the nodes and edges of G. Denote by $\Psi_\mathcal{A}$ the set of all status variables adopted in $\mathcal{A}$. (b) For each status variable $x_i \in \Psi_\mathcal{A}$, algorithm $\mathcal{A}$ applies an *update function* f_{x_i} to decide the value of x_i, *i.e.,* $x_i = f_{x_i}(Y_{x_i})$, where Y_{x_i} is a set of status variables in $\Psi_\mathcal{A}$. Moreover, (c) it employs a logical statement σ_{x_i} defined on status variables such that σ_{x_i} is *guaranteed to be true* right after every invocation of $f_{x_i}(Y_{x_i})$. We denote by $\sigma_\mathcal{A}$ the conjunction of the logical statements σ_{x_i} for all x_i's in $\Psi_\mathcal{A}$, referred to as *the invariant of* $\mathcal{A}$.

Algorithm $\mathcal{A}$ often computes $Q(G)$ by iteratively operating on G and $S_\mathcal{A}$, and produces partial results $R_\mathcal{A}$, *i.e.,* the values of the status variables in $\Psi_\mathcal{A}$ in each round. We use $D_\mathcal{A}$ to denote $(S_\mathcal{A}, R_\mathcal{A})$, referred to as *status*, which keeps track of the computation.

We say that $\mathcal{A}$ is a *fixpoint algorithm* if it is expressible as

$$(D^{t+1}_\mathcal{A},\ H^{t+1}_\mathcal{A}) = f_\mathcal{A}(D^t_\mathcal{A},\ Q,\ G,\ H^t_\mathcal{A}),\ \text{ where} \qquad (1)$$

(1) $D^t_\mathcal{A}$ denotes the status $D_\mathcal{A}$ after $t-1$ rounds of iterations, and $D^0_\mathcal{A}$ includes the initial values for all status variables in $\Psi_\mathcal{A}$;

(2) $H^t_\mathcal{A}$ denotes the set $H_\mathcal{A}$ of status variables collected before the start of round t, referred to as *the scope of round t*; some status variables in $H^t_\mathcal{A}$ are to be updated in round t; initially $H^0_\mathcal{A}$ contains variables x_i that may have false statements σ_{x_i} for round 0; and

(3) $f_\mathcal{A}$ is the intermediate consequence operator of the fixpoint, called the *step function* of algorithm $\mathcal{A}$. It selects status variables from the scope $H^t_\mathcal{A}$ and performs update $f_{x_i}(Y_{x_i})$ on each selected x_i to compute status $D^{t+1}_\mathcal{A}$. Moreover, $f_\mathcal{A}$ returns the scope $H^{t+1}_\mathcal{A}$ that updates $H^t_\mathcal{A}$ with *affected* status variables of round t, *i.e.,* those x_i's when the value of some variable in Y_{x_i} is *changed* in round t.

We denote by Φ *the class of all such fixpoint algorithms.*

Intuitively, a fixpoint algorithm $\mathcal{A}$ is essentially "update-based". It computes $Q(G)$ by applying its step function $f_\mathcal{A}$ in rounds, *guided by* the invariant $\sigma_\mathcal{A}$. In round t, $f_\mathcal{A}$ propagates the changes from the last round $t-1$ to the scope $H^t_\mathcal{A}$ and corresponding parts of $D^t_\mathcal{A}$, and identifies the scope $H^{t+1}_\mathcal{A}$ for the next round. The process proceeds until it reaches a fixpoint r such that $D^{r+1}_\mathcal{A} = D^r_\mathcal{A}$ and $H^{r+1} = \emptyset$, *i.e.,* when no more changes can be made. All logical statements in invariant $\sigma_\mathcal{A}$ hold when the process terminates.

```
Input: Graph $G = (V, E, L)$, source vertex $s$.
Output: The shortest distance $x_v$ for each $v$ in $G$.
1.  $x_s \leftarrow 0$;  for each $v \neq s$ do $x_v \leftarrow \infty$;
2.  initialize priority queue que; /* scope $H_\mathcal{A}$: neighbors of nodes in que */
3.  que.addOrAdjust($s$, $x_s$);
4.  while que is not empty do                /* step function $f_\mathcal{A}$ */
5.      $v \leftarrow$ que.pop();
6.      for each $u \in$ out_nbr ($v$) do
7.          $alt \leftarrow x_v + L(v, u)$;
8.          if $alt < x_u$ then              /* apply update function $f_{x_u}$ */
9.              $x_u \leftarrow alt$;  que.addOrAdjust($u$, $x_u$);   /* adjust scope */
10. return $\{x_v \mid v \in V\}$;
```

Figure 1: Batch algorithm for SSSP

A variety of graph query classes have fixpoint algorithms, including SSSP [25], CC [12], Sim [26], DFS [43], LCC [42], and bi-connectivity (BC) [43], just to name a few.

Example 1: Dijkstra's algorithm for SSSP is a fixpoint algorithm, as shown in Fig. 1. Consider a graph $G = (V, E, L)$, where $L(u, v)$ is the length of edge (u, v). As data structure in $S_\mathcal{A}$, it associates each node v with a status variable x_v, recording the shortest distance from s, initialized as ∞ for $v{\neq}s$ (line 1). Apart from $S_\mathcal{A}$, $D_\mathcal{A}$ includes a priority queue que, and the scope $H_\mathcal{A}$ includes all outgoing neighbors of the nodes in que (line 2). Initially, que only has x_s. Its step function $f_\mathcal{A}$ is defined in lines 5-9. Each time $f_\mathcal{A}$ pops a node v from que, and when logical statement σ_{x_u} (*i.e.,* $x_u{=}f_{x_u}(Y_{x_u})$) does not hold for v's outgoing neighbor u, it applies update function f_{x_u} to x_u, setting it to $\min_{x_v \in Y_{x_u}} \{x_v + L(v, u)\}$ (lines 6-9). Here Y_{x_u} includes status variables of u's incoming neighbors. Function $f_\mathcal{A}$ also adjusts que accordingly, and the changes will be propagated to the next iteration. The process terminates when que and the scope become empty. At this time, the invariant $\sigma_\mathcal{A}$ holds. □

Incrementalization. We next show how to deduce an incremental algorithm $\mathcal{A}_\Delta$ from a fixpoint algorithm $\mathcal{A} \in \Phi$. Suppose that given a graph G and a query $Q \in \mathcal{Q}$, batch algorithm $\mathcal{A}$ computes $Q(G)$ and ends up with a fixpoint $D^r_\mathcal{A}$. Incremental algorithm $\mathcal{A}_\Delta$ starts from $D^r_\mathcal{A}$. It additionally takes updates ΔG as input, and possibly extends $S_\mathcal{A}$ to $S_{\mathcal{A}_\Delta}$ with auxiliary structures. It employs $D^t_{\mathcal{A}_\Delta} = (S^t_{\mathcal{A}_\Delta}, R^t_{\mathcal{A}_\Delta})$ and $H^t_{\mathcal{A}_\Delta}$ analogous to their counterparts of $\mathcal{A}$. It is dominated by $f_{\mathcal{A}_\Delta}$, which is (a mild extension of) the step function $f_\mathcal{A}$ of $\mathcal{A}$ to cope with new auxiliary structures in $S_{\mathcal{A}_\Delta}$.

Along the same lines as $\mathcal{A}$, it iterates in rounds to identify scope $H_{\mathcal{A}_\Delta}$ and compute new status $D_{\mathcal{A}_\Delta}$ as follows:

$$(D^{t+1}_{\mathcal{A}_\Delta},\ H^{t+1}_{\mathcal{A}_\Delta}) \;=\; f_{\mathcal{A}_\Delta}(D^t_{\mathcal{A}_\Delta},\ Q,\ G\oplus\Delta G,\ H^t_{\mathcal{A}_\Delta}), \qquad (2)$$

$$(D^0_{\mathcal{A}_\Delta}, H^0_{\mathcal{A}_\Delta}) \;=\; h(D^r_\mathcal{A}, \Delta G). \qquad (3)$$

Here h is an *initial scope function* that identifies scope $H^0_{\mathcal{A}_\Delta}$ for $\mathcal{A}_\Delta$. It is derived from the old fixpoint $D^r_\mathcal{A}$ and updates ΔG. It may initialize auxiliary structures of $S_{\mathcal{A}_\Delta}$ and changes $D^r_\mathcal{A}$ to status $D^0_{\mathcal{A}_\Delta}$.

Incremental algorithm $\mathcal{A}_\Delta$ works along the same lines as batch algorithm $\mathcal{A}$. It starts from $D^0_{\mathcal{A}_\Delta}$ and $H^0_{\mathcal{A}_\Delta}$. Moreover, it employs step function $f_{\mathcal{A}_\Delta}$ to identify scope $H^{t+1}_{\mathcal{A}_\Delta}$ and update status to $D^{t+1}_{\mathcal{A}_\Delta}$ in round t. The process iterates until it reaches a fixpoint.

Comparing $\mathcal{A}$ and $\mathcal{A}_\Delta$, one can see that both batch and incremental algorithms are dominated by step functions. The step function $f_{\mathcal{A}_\Delta}$ (resp. status $D_{\mathcal{A}_\Delta}$) of $\mathcal{A}_\Delta$ extends $f_{\mathcal{A}}$ (resp. $D_{\mathcal{A}}$) of $\mathcal{A}$ only to cope with newly added auxiliary structure in $S_{\mathcal{A}_\Delta}$. As will be seen shortly, the auxiliary structures are mostly for speeding up incremental computation. That is, incremental algorithm $\mathcal{A}_\Delta$ essentially adopts the same logic and data structures of $\mathcal{A}$. It differs from $\mathcal{A}$ mostly in the use of initial scope function h.

Scope function. The main objective of initial scope function h is to identify and adjust status variables whose corresponding logical statements no longer hold due to the updates ΔG. For instance, the distance value of some variable x_v may become invalid in SSSP if node v's adjacent edges evolve. As such, a practical initial scope function h (1) analyzes the dependencies among status variables that are implied in the invariant $\sigma_{\mathcal{A}}$, and finds all variables affected by ΔG, which should have new values; and (2) tunes affected variables to their "feasible" status, from where the new correct result can be obtained by applying the logic analogous to that of $\mathcal{A}$, *i.e.,* resuming $\mathcal{A}$'s iterative computation.

Deducible. We say that incremental algorithm $\mathcal{A}_\Delta$ is *deducible* from batch $\mathcal{A}$ if its $f_{\mathcal{A}_\Delta}$, $D_{\mathcal{A}_\Delta}$ and $H_{\mathcal{A}_\Delta}$ remain the same as their batch counterparts $f_{\mathcal{A}}$, $D_{\mathcal{A}}$ and $H_{\mathcal{A}}$, respectively. That is, it uses the same step function $f_{\mathcal{A}}$ and adds no auxiliary structure to $S_{\mathcal{A}}$.

Intuitively, $\mathcal{A}_\Delta$ adopts exactly the same logic and data structures of $\mathcal{A}$. It differs from $\mathcal{A}$ *only* in its use of initial scope function h, to cope with updates ΔG. In fact, by retracing the change propagation caused by $\mathcal{A}$, a conservative policy can eliminate all the previous effects enforced on *potential affected* (PE) variables. Here each PE variable x_i may have a new final value due to ΔG and the changed input set of update function f_{x_i}. Then we have a correct incremental algorithm $\mathcal{A}_\Delta$ deducible from $\mathcal{A}$ when PE variables and their input sets are firstly "reset" to (default) initial values, followed by executing the step function of $\mathcal{A}$ directly on the resulting status.

Theorem 1: *From every fixpoint algorithm $\mathcal{A} \in \Phi$, a correct incremental algorithm $\mathcal{A}_\Delta$ is deducible.* □

The example below illustrates the intuition behind Theorem 1, whose proof is a special case of that of Theorem 3 (see Section 4).

Example 2: Given an undirected graph $G = (V, E, L)$, where each node carries a node id, CC is to compute the id of the component to which each node $v \in V$ belongs. A fixpoint algorithm for CC, denoted as CC_{fp}, defines a status variable x_v for each $v \in V$, to store v's component id, initialized as its node id. Initially, the scope $H_{\mathcal{A}}$ includes all status variables. In each round, its step function $f_{\mathcal{A}}$ removes one x_v from $H_{\mathcal{A}}$ and computes $f_{x_v}(Y_{x_v}) = \min(\{x_v\} \cup Y_{x_v})$, *i.e.,* x_v takes the smallest value among its neighbors, and Y_{x_v} covers the neighbors of v. If the value of x_v changes, $f_{\mathcal{A}}$ adds to scope $H_{\mathcal{A}}$ all status variables in Y_{x_v}. The process terminates when $H_{\mathcal{A}}$ becomes empty.

An incremental algorithm for CC is deducible from CC_{fp}, in which the initial scope function h first marks each x_v as a PE variable if node v is covered by updates ΔG. Then it iteratively expands the set of PE variables by including all x_u if there exist other variables x_w in the input sets Y_{x_u} that are already marked PE; it updates each PE variable x_v to v's node id and returns PE variables found as $H^0_{\mathcal{A}_\Delta}$. After that, the incremental algorithm applies step function $f_{\mathcal{A}}$ of CC_{fp} to compute new values for PE variables.

In short, h cancels previous effects on PE variables via a "change propagation" analogous to that in CC_{fp}, *i.e.,* propagated along edges; and adjusts them to "feasible" status, *i.e.,* node id's. Other variables not touched by h remain stable and will not be accessed by $f_{\mathcal{A}}$. □

Remark. We remark the following. (1) There exist graph algorithms that are not expressible as fixpoint computation as above. These include, *e.g.,* REC [33] for graph sketching and METIS [28] for graph partitioning. We defer the study of incrementalizing such algorithms to future work. (2) The work is a step toward incrementalizing batch algorithms with performance guarantees. Nonetheless, it is not yet a fully automated method. It still requires domain knowledge to deduce initial scope function h (see Section 4).

4 BOUNDED INCREMENTALIZATION

Not all incremental algorithms deducible from fixpoint algorithms $\mathcal{A}$ are efficient, *e.g.,* relatively bounded. For instance, the incremental algorithm in Example 2 may recompute the component id's for the entire set of nodes when given a unit edge deletion to a graph having a single connected component. In light of this, in this section we identify generic conditions under which the incremental $\mathcal{A}_\Delta$ warrants the boundedness relative to $\mathcal{A}$. We also provide guidelines for how to deduce $\mathcal{A}_\Delta$ from $\mathcal{A}$ with performance guarantees.

Guaranteeing relative boundedness. To identify conditions for achieving correct and relatively bounded incrementalization, we start with a monotone characterization for fixpoint algorithms.

Monotonicity. Let $\preceq$ be a partial order on the domain of status variables in $\Psi_{\mathcal{A}}$. We define (i) $D^1_{\mathcal{A}} \preceq D^2_{\mathcal{A}}$ if $x^1_i \preceq x^2_i$ for each status variable $x_i \in \Psi_{\mathcal{A}}$, where x^1_i and x^2_i are copies of x_i in status $D^1_{\mathcal{A}}$ and $D^2_{\mathcal{A}}$, respectively; and (ii) $Y^1_{x_i} \preceq Y^2_{x_i}$ if $y^1 \preceq y^2$ for every variable y in the set Y_{x_i}, where y^1 and y^2 are copies of y in $Y^1_{x_i}$ and $Y^2_{x_i}$, respectively.

We say that algorithm $\mathcal{A}$ is *contracting* if the status variables in $\Psi_{\mathcal{A}}$ are updated following the partial order $\preceq$; and that $\mathcal{A}$ is *monotonic* if for each $x_i \in \Psi_{\mathcal{A}}$, the update function f_{x_i} is monotonic, *i.e.,* $Y^1_{x_i} \preceq Y^2_{x_i}$ implies that $f_{x_i}(Y^1_{x_i}) \preceq f_{x_i}(Y^2_{x_i})$. In the rest of this section, we study contracting and monotonic algorithms only.

Consider the computation of a contracting and monotonic algorithm $\mathcal{A} \in \Phi$. Denote by $x^\perp_i$ and x^*_i the initial and final value of each status variable x_i when running $\mathcal{A}$ on graph G, respectively; and by $D^\perp_{\mathcal{A}}$ and $D^*_{\mathcal{A}}$ the initial and final status of $\mathcal{A}$ on G.

We say that status variable x_i is *feasible for graph G* if $x^*_i \preceq x_i \preceq x^\perp_i$, *i.e.,* x_i carries a feasible value between x^*_i and $x^\perp_i$. Similarly, status $D_{\mathcal{A}}$ is *feasible for G* if each x_i of $D_{\mathcal{A}}$ is feasible for G.

When $\mathcal{A}$ is both contracting and monotonic, all status variables are necessarily feasible for graph G. More specifically,

$$D^*_{\mathcal{A}} \preceq \cdots \preceq D^{t+1}_{\mathcal{A}} \preceq D^t_{\mathcal{A}} \preceq \cdots \preceq D^0_{\mathcal{A}} = D^\perp_{\mathcal{A}}. \quad (4)$$

That is, each round of $\mathcal{A}$ essentially updates a feasible status $D^t_{\mathcal{A}}$ to another feasible status $D^{t+1}_{\mathcal{A}}$, following the order $\preceq$. We say a scope $H_{\mathcal{A}}$ is *valid w.r.t.* $D_{\mathcal{A}}$ if every status variables x_i of $D_{\mathcal{A}}$ that violates condition σ_{x_i} is in $H_{\mathcal{A}}$. Note that in each round, $H^t_{\mathcal{A}}$ is valid *w.r.t.* $D^t_{\mathcal{A}}$, since the computation of $\mathcal{A}$ is guided by invariant $\sigma_{\mathcal{A}}$.

Church-Rosser. If $\mathcal{A}$ is both contracting and monotonic, then the computation of $\mathcal{A}$ is *Church-Rosser* [9], *i.e.,* it is guaranteed to converge at the same result no matter how it runs.

Lemma 2: *Let $D_{\mathcal{A}}$ be a feasible status for graph G and $H_{\mathcal{A}}$ be a valid scope* w.r.t. *$D_{\mathcal{A}}$. If $\mathcal{A}$ is contracting and monotonic, then the computation of $\mathcal{A}$ starting from $(D_{\mathcal{A}}, H_{\mathcal{A}})$ converges at $(D^*_{\mathcal{A}}, \emptyset)$.* □

Proof sketch: Suppose that the batch run starting from $(D^0_{\mathcal{A}}, H^0_{\mathcal{A}})$ terminates after k rounds with $D^k_{\mathcal{A}} = D^*_{\mathcal{A}}$. Consider an arbitrary run ρ of $\mathcal{A}$ from $(D_{\mathcal{A}}, H_{\mathcal{A}})$. Denote by $(D^t_{\mathcal{A},\rho}, H^t_{\mathcal{A},\rho})$ the status and scope after the t-th round. Suppose that ρ terminates with $(D^*_{\mathcal{A},\rho}, \emptyset)$ after ℓ rounds for some $\ell > 0$. By induction on i and the assumed properties of $\mathcal{A}$, we can show that (1) $D^*_{\mathcal{A},\rho} \preceq D^i_{\mathcal{A}}$ for $0 \le i \le k$; and (2) $D^*_{\mathcal{A}} \preceq D^i_{\mathcal{A},\rho}$ for $0 \le i \le \ell$. It follows that $D^*_{\mathcal{A},\rho} = D^*_{\mathcal{A}}$. □

Recall that in the incrementalized algorithm $\mathcal{A}_\Delta$, $(D^0_{\mathcal{A}_\Delta}, H^0_{\mathcal{A}_\Delta}) = h(D^r_{\mathcal{A}}, \Delta G)$ and $f_{\mathcal{A}_\Delta}$ (*i.e.,* the mild extension of step function $f_{\mathcal{A}}$) conducts the fixpoint computation over the updated graph $G \oplus \Delta G$, starting from $(D^0_{\mathcal{A}_\Delta}, H^0_{\mathcal{A}_\Delta})$. Given Lemma 2, one can see that it suffices to develop a *correct* scope function h for $\mathcal{A}_\Delta$ such that $D^0_{\mathcal{A}_\Delta}$ is feasible for $G \oplus \Delta G$ and $H^0_{\mathcal{A}_\Delta}$ is valid *w.r.t.* $D^0_{\mathcal{A}_\Delta}$.

A brute-force approach to designing a correct h function is to reset all (PE) variables to initial values and compute the corresponding $H^0_{\mathcal{A}_\Delta}$, as in Example 2. This is too costly. To bound the cost of $\mathcal{A}_\Delta$, we have to bound the initial scope $H^0_{\mathcal{A}_\Delta}$ identified by h.

We say that the initial scope function h is *bounded* if the scope $H^0_{\mathcal{A}_\Delta}$ identified by h satisfies that $H^0_{\mathcal{A}_\Delta} \subseteq \mathsf{AFF}$.

When utilizing a correct and bounded initial scope function h, the incremental algorithm $\mathcal{A}_\Delta$ may require some auxiliary structures, *e.g.,* timestamps, and extend step function accordingly.

Weakly deducible. An incremental algorithm $\mathcal{A}_\Delta$ is *weakly deducible* from $\mathcal{A}$ if (a) the data structure $S_{\mathcal{A}_\Delta}$ extends $S_{\mathcal{A}}$ by associating timestamp with (some of) its status variables x_i to record the time of the last change to x_i; (b) the step function $f_{\mathcal{A}_\Delta}$ is the same as $f_{\mathcal{A}}$ except that it updates the timestamp of x_i when x_i is updated; and (c) scope $H_{\mathcal{A}_\Delta}$ extends $H_{\mathcal{A}}$ similarly.

Intuitively, $\mathcal{A}_\Delta$ still adopts the same logic and data structures of $\mathcal{A}$, except that it records the timestamps of status variables for identifying scope function h. The step function $f_{\mathcal{A}_\Delta}$ retains the same complexity as $f_{\mathcal{A}}$, since $f_{\mathcal{A}_\Delta}$ visits timestamps (auxiliary structures) only when $f_{\mathcal{A}}$ visits their corresponding status variables in a run.

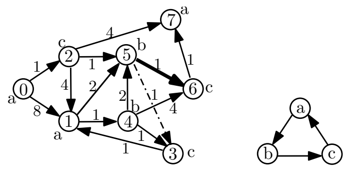
Relative boundedness conditions. We now identify conditions under which relatively bounded incrementalization is warranted.

(C1) The initial scope function h of $\mathcal{A}_\Delta$ is correct and bounded.

(C2) The batch algorithm $\mathcal{A}$ is contracting and monotonic.

Theorem 3: *For every fixpoint algorithm $\mathcal{A} \in \Phi$, under conditions* (**C1**) *and* (**C2**)*, there exists a weakly deducible incremental algorithm $\mathcal{A}_\Delta$ that is both correct and bounded relative to $\mathcal{A}$.* □

That is, for a contracting and monotonic algorithm $\mathcal{A}$, to get an incremental algorithm $\mathcal{A}_\Delta$ bounded relative to $\mathcal{A}$, one mainly needs to implement a correct and bounded initial scope function h.



(a) G (b) Q

Figure 2: Example graph and pattern

Proof sketch: When h is correct and bounded (**C1**), status $D^0_{\mathcal{A}_\Delta}$ produced by h is feasible for $G \oplus \Delta G$ and $H^0_{\mathcal{A}_\Delta}$ is valid *w.r.t.* $G \oplus \Delta G$. Thus $\mathcal{A}_\Delta$ is correct by Lemma 2. In particular, there always exists a correct h that retrieves all PE variables by analyzing the invariant $\sigma_{\mathcal{A}}$ and the change propagation of $\mathcal{A}$ only and resets PE variables to initial values, without using timestamps. This proves Theorem 1.

It remains to verify the relative boundedness of incremental algorithm $\mathcal{A}_\Delta$. By induction on the number t of rounds, we first prove that $H^t_{\mathcal{A}_\Delta} \subseteq \mathsf{AFF}$ for $t \geq 0$. Indeed, AFF contains a status variable x_i when (i) x_i's value changes on the updated graph $G \oplus \Delta G$, or (ii) the input set Y_{x_i} evolves even if x_i has already reached its final value. Such x_i's exactly match the variables that $\mathcal{A}_\Delta$ visits, by the semantics of h and the contracting property inherited from $\mathcal{A}$.

Since $H^t_{\mathcal{A}_\Delta} \subseteq \mathsf{AFF}$ and step function $f_{\mathcal{A}_\Delta}$ updates the timestamps only if the status variables in $H^t_{\mathcal{A}_\Delta}$ have changed, the size of the data accessed by $\mathcal{A}_\Delta$ is bounded by a function of $|\mathsf{AFF}|$, $|\Delta G|$ and $|Q|$. Therefore, $\mathcal{A}_\Delta$ is bounded relative to $\mathcal{A}$. □

Note that Theorem 1 and Theorem 3 hold for batch updates, since in the proof above, ΔG is not restricted to unit updates.

Deducing scope function. We next provide a guideline for deducing initial scope function h that is bounded and correct, *i.e.,* to satisfy condition (**C1**). We will see that when h is in place, relatively bounded $\mathcal{A}_\Delta$ can be readily deduced. Given a fixpoint $D^r_{\mathcal{A}}$ and input updates ΔG, h produces an initial scope $H^0_{\mathcal{A}_\Delta}$ and a feasible status $D^0_{\mathcal{A}_\Delta}$ for $G \oplus \Delta G$. We define h by refining PE variables to capture "essential" change propagation incurred by algorithm $\mathcal{A}$ on graphs, using at most timestamps as an additional auxiliary structure.

Capturing change propagation. Recall that batch algorithm $\mathcal{A}$ determines each status variable x_i by using the update function f_{x_i} with input set Y_{x_i}. However, it is common that only a subset of Y_{x_i} affects the final value x^*_i of x_i for graph G. To capture this, for each variable $x_i \in \Psi_{\mathcal{A}}$, we define its *anchor set* C_{x_i} as the subset of Y_{x_i} such that (a) both the value of $f_{x_i}(Y_{x_i})$ remains stable and the logical statement σ_{x_i} holds when every status variable y_i in C_{x_i} carries its final value y^*_i, while all other status variables in $Y_{x_i} \setminus C_{x_i}$ can have any feasible values; and (b) the final value y^*_i is determined ahead of x^*_i by batch algorithm $\mathcal{A}$ for each $y_i \in C_{x_i}$.

We say that status variable y_i is a *contributor* of x_i if it is included in one anchor set of x_i. Intuitively, contributors characterize which status variables in Y_{x_i} have essential impacts on the final value of x_i, *i.e.,* changes propagated *from* y_i to x_i are essential.

With contributors, we can build directed acyclic graphs (DAGs) in which the nodes are all status variables in $\Psi_{\mathcal{A}}$, and there exists an edge from x_j to x_i if and only if x_j is a contributor of x_i. Denote by $<_C$ the topological order on the variables in the DAGs, which can also be easily deduced from the timestamps. The DAGs represent

v	G		$G \oplus \Delta G$	
	x_v	C_{x_v}	x_v	C_{x_v}
0	0	$\emptyset$	0	$\emptyset$
1	5	$\{2\}$	4	$\{3\}$
2	1	$\{0\}$	1	$\{0\}$
3	7	$\{4\}$	3	$\{5\}$
4	6	$\{1\}$	5	$\{1\}$
5	2	$\{2\}$	2	$\{2\}$
6	3	$\{5\}$	9	$\{4\}$
7	4	$\{6\}$	5	$\{2\}$

(a) SSSP

v	G		$G \oplus \Delta G$	
	t	$C_{x[v,L(v)]}$	t	$C_{x[v,L(v)]}$
0	0	$\emptyset$	0	$\emptyset$
1	∞	$\emptyset$	∞	$\emptyset$
2	∞	$\emptyset$	∞	$\emptyset$
3	∞	$\emptyset$	∞	$\emptyset$
4	∞	$\emptyset$	∞	$\emptyset$
5	2	$\{6\}$	∞	$\emptyset$
6	1	$\{7\}$	1	$\{7\}$
7	0	$\emptyset$	0	$\emptyset$

(b) Sim

v	G		$G \oplus \Delta G$	
	x_v	C_{x_v}	x_v	C_{x_v}
0	[0, 15]	$\{r\}$	[0, 15]	$\{r\}$
1	[1, 12]	$\{0\}$	[1, 12]	$\{0\}$
2	[13, 14]	$\{0\}$	[13, 14]	$\{0\}$
3	[9, 10]	$\{4\}$	[3, 4]	$\{5\}$
4	[8, 11]	$\{1\}$	[8, 11]	$\{1\}$
5	[2, 7]	$\{1\}$	[2, 7]	$\{1\}$
6	[3, 6]	$\{5\}$	[10.1, 10.4]	$\{4\}$
7	[4, 5]	$\{6\}$	[10.2, 10.3]	$\{6\}$

(c) DFS

v	G		$G \oplus \Delta G$	
	d_v	λ_v	d_v	λ_v
0	2	1	2	1
1	5	4	5	5
2	4	2	4	2
3	2	1	3	3
4	4	3	4	3
5	4	3	4	4
6	3	1	2	0
7	2	0	2	0

(d) LCC

Figure 3: Status variables and anchor sets

the dependency of the essential computation, *i.e.,* essential change propagation of algorithm $\mathcal{A}$ when running on G.

Example 3: Consider the invocation of Dijkstra's algorithm for SSSP over the graph G shown in Figure 2(a) (excluding the dotted edge $(5, 3)$), where node 0 is the source. The final values, *i.e.,* the shortest distance of each status variable x_v together with the anchor set C_{x_v}, are given in Fig. 3(a) in the column annotated G. Here C_{x_v} is $\{x_u \in Y_{x_v} \mid x_u + L(u, v) = x_v\}$. Intuitively, if $x_u \in C_{x_v}$, then u lies on the shortest path from node 0 to v. The order $<_C$ is directly deduced from the final values, as the order of the latest update time of status variables, *i.e.,* $x_u <_C x_v$ if and only if $x_u < x_v$. □

<u>*Adjusting fixpoint.*</u> Figure 4 outlines how the initial scope function h works. It iteratively identifies infeasible values in the previous fixpoint status $D^r_{\mathcal{A}}$, and assigns feasible values to them for the updated graph $G \oplus \Delta G$, following the topological order $<_C$.

It first includes in scope $H^0_{\mathcal{A}_\Delta}$ all status variables whose update functions have changed input sets (line 1). Then h initializes a priority queue que recording status variables that may carry infeasible values (line 2). In each iteration of revising infeasible values, h pops a variable x_i from que with the smallest order *w.r.t.* $<_C$, generates the new feasible input set $\overline{Y}_{x_i}$, and compares a new value $f_{x_i}(\overline{Y}_{x_i})$ with x_i (lines 4-7). If x_i carries an old value that may be infeasible for $G \oplus \Delta G$, *i.e.,* $x_i \prec f_{x_i}(\overline{Y}_{x_i})$, it expands the scope $H^0_{\mathcal{A}_\Delta}$, updates x_i with the new value in status $D^0_{\mathcal{A}_\Delta}$ and adjusts que by including potential infeasible variables because of the infeasible contributor x_i (lines 8-9). The process terminates when que becomes empty.

To see that h is correct, note that the old value of x_i is infeasible *only if* (a) there exists at least one contributor y of x_i (*i.e.,* a status variable ahead of x_i in the topological order $<_C$) has been confirmed infeasible for $G \oplus \Delta G$; or (b) the input set Y_{x_i} of f_{x_i} changes due to ΔG. By induction on the rounds of the iteration in h, we can verify that all infeasible status variables are collected in que. Moreover, since h processes infeasible ones following $<_C$, the computed new values are guaranteed to be feasible for $G \oplus \Delta G$.

One can also verify that such initial scope function h is bounded since all status variables that are included in que are covered by AFF. They have different values between the two runs of $\mathcal{A}$, since either their input sets of update functions or contributors evolve.

```
Input: Previous fixpoint status $D^r_{\mathcal{A}}$ and input updates $\Delta G$.
Output: An initial scope $H^0_{\mathcal{A}_\Delta}$ and a feasible status $D^0_{\mathcal{A}_\Delta}$ for $G \oplus \Delta G$.
1.  collect into $H^0_{\mathcal{A}_\Delta}$ the status variables with evolved input sets of
      update functions due to $\Delta G$;  $D^0_{\mathcal{A}_\Delta} \leftarrow D^r_{\mathcal{A}}$;
2.  initialize a priority queue que via $H^0_{\mathcal{A}_\Delta}$ according to order $<_C$;
3.  while que is not empty do
4.      $x_i \leftarrow$ que.pop();  $\overline{Y}_{x_i} \leftarrow$ convert($Y_{x_i}, \Delta G$);
5.      for each $y \in \overline{Y}_{x_i}$ do      // make $\overline{Y}_{x_i}$ feasible
6.          if $x_i <_C y$ then $y \leftarrow y^\perp$;
7.      if $x_i \prec f_{x_i}(\overline{Y}_{x_i})$ then     // $x_i$ is potentially infeasible
8.          $H^0_{\mathcal{A}_\Delta} \leftarrow H^0_{\mathcal{A}_\Delta} \cup \{x_i\}$;   $D^0_{\mathcal{A}_\Delta}(x_i) \leftarrow f_{x_i}(\overline{Y}_{x_i})$;
9.          for each $z$ with $x \in C_z$ do que.insert (z);
10. return $H^0_{\mathcal{A}_\Delta}$ and $D^0_{\mathcal{A}_\Delta}$;
```

Figure 4: An implementation of initial scope function h

Example 4: We show how a correct and bounded scope function h works for SSSP. Continuing with Example 3, consider updates ΔG that delete bold edge $(5, 6)$ and insert dotted edge $(5, 3)$. Given ΔG and the final values of status variables in Example 3 (*i.e.,* $D^r_{\mathcal{A}}$), h produces feasible status $D^0_{\mathcal{A}_\Delta}$ and initial scope $H^0_{\mathcal{A}_\Delta}$ as follows.

(1) Initially, $H^0_{\mathcal{A}_\Delta} = \{x_3, x_6\}$ since only the input sets *w.r.t.* variables for destination nodes of the edges in ΔG evolve. Hence the queue que starts with $\{x_3, x_6\}$, and h pops x_6 first. Since $x_6 <_C x_4$, *i.e.,* node 6 has a shorter distance than node 4 from the source, h generates a new feasible input set $\overline{Y}_{x_6}$ as $\{x_4{=}\infty\}$. Here x_4 is reset to initial value, which must be feasible for $G \oplus \Delta G$. Observe that $x_6 = 3 \prec \infty = f_{x_6}(\overline{Y}_{x_6})$; thus x_6 is assigned ∞. In addition, h includes x_7 in the queue for further inspection because $x_6 \in C_{x_7}$.

(2) Next, h pops x_7 from que and makes it feasible. Similar to the processing of x_6, h determines $\overline{Y}_{x_7}$ as $\{x_6 = \infty, x_2 = 1\}$, changes x_7 to 5 and adds it to $H^0_{\mathcal{A}_\Delta}$. No nodes are included in que in this step.

(3) At last h inspects x_3. Since $x_4 <_C x_3$ and $x_5 <_C x_3$, we have that $f_{x_3}(\overline{Y}_{x_3}) = 3 \prec x_3$. Hence x_3's old value is still feasible and no new variables are further pushed into the queue que.

At the end, h returns $\{x_3, x_6, x_7\}$ as $H^0_{\mathcal{A}_\Delta}$ and the revised distance values as $D^0_{\mathcal{A}_\Delta}$. Here $D^0_{\mathcal{A}_\Delta}$ differs from fixpoint status $D^r_{\mathcal{A}}$ only in x_6 (∞ vs. 3) and x_7 (5 vs. 4). Observe that x_6 and x_7 indeed change in $G \oplus \Delta G$ (see Figure 3(a)). Thus $H^0_{\mathcal{A}} \subseteq$ AFF and h is bounded. □

Having developed initial scope function h, we can easily deduce the incremental algorithm $\mathcal{A}_\Delta$ as its step function $f_{\mathcal{A}_\Delta}$ is almost the same as $f_{\mathcal{A}}$. For instance, Figure 5 gives the incremental algorithm for SSSP, which adopts scope function h of Example 4 and can compute the new distance values in $G \oplus \Delta G$, as shown in Fig. 3(a).

Observe that for SSSP, the fixpoint $D^r_{\mathcal{A}}$ already subsumes the anchor sets and tells us whether the final value of a variable is determined prior to another. Thus $\mathcal{A}_\Delta$ of Fig. 5 is *deducible* from Dijkstra's algorithm. However, some other $\mathcal{A}_\Delta$ may need to use timestamps when deciding the order $<_C$ and scope function h, although timestamps can be deduced from the batch runs of $\mathcal{A}$ as a byproduct. Under such case, $\mathcal{A}_\Delta$ is weakly deducible from $\mathcal{A}$.

Example 5: An incremental algorithm for CC, denoted as IncCC, is *weakly deducible* from CC_{fp} (Example 2). It adopts timestamps to determine the anchor sets and the order $<_C$ on status variables. More specifically, the anchor sets for each status variable x_w include

```
Input: Graph $G = (V, E, L)$, source $s$, updates $\Delta G$, previous fixpoint $D^r_{\mathcal{A}}$.
Output: The updated shortest distance $x_v$ for each $v$ in $G \oplus \Delta G$.
1.  $(D_{\mathcal{A}_\Delta}, H_{\mathcal{A}_\Delta}) \leftarrow h(D^r_{\mathcal{A}}, \Delta G)$;          /* apply initial scope function h */
2.  initialize a priority queue que;
3.  for each incoming neighbor $v$ of a node in $H_{\mathcal{A}_\Delta}$ do
4.      que.addOrAdjust($v$, $x_v$);
5.  the same lines 4-10 as in the batch SSSP algorithm from Figure 1;
```

Figure 5: Incremental algorithm for SSSP

$x_{w'}$ such that w' is a neighbor of w, and $x_{w'}$ has a timestamp smaller than that of x_w. With timestamps, the order $<_C$ is immediately derived, *i.e.,* $x_w <_C x_{w'}$ if and only if x_w has a smaller timestamp.

Based on these, IncCC's initial scope function h works following the guidelines in Figure 4. Then it proceeds to compute the new values for affected variables using the step function $f_{\mathcal{A}_\Delta}$ in a way similar to $f_{\mathcal{A}}$ of CC_{fp}, except that the corresponding timestamps are also updated accordingly. In fact, when deleting an edge e from a single connected component, only one endpoint of e with a larger timestamp may be truly affected and be processed by IncCC. This is in contrast to the deducible incremental algorithm given in Example 2, which treats both endpoints as PE variables. Hence the relative boundedness of IncCC can be confirmed by Theorem 3. □

Vertex updates. The methods can be readily extended to handle node insertions and deletions. Clearly, removing a node is the same as removing all its incident edges and hence can be treated as edge updates. When inserting a new node v together with its adjacent edges (assuming a dummy edge when no edge is explicitly added), scope function h first computes initial values for new status variables introduced by v, and revises the order $<_C$ accordingly. Then h proceeds along the same lines as Fig. 4. The correctness and relative boundedness are still warranted by Lemma 2 and Theorem 3.

5 PROOF OF CONCEPT

As a proof of concept, we deduce incremental algorithms for graph simulation, depth-first search and local clustering coefficient.

5.1 Graph Simulation

We start with graph pattern matching via *graph simulation* (Sim).

Given a data graph $G = (V, E, L)$ and a pattern graph $Q = (V_Q, E_Q, L_Q)$, a binary relation $R \subseteq V \times V_Q$ is a *simulation* if for each $\langle v, u\rangle \in R$, (a) $L(v) = L_Q(u)$, and (b) for each edge $(u, u') \in E_Q$, there exists an edge $(v, v') \in E$ such that $\langle v', u'\rangle \in R$. If G matches Q, there exists a *unique non-empty maximum* R, denoted as $Q(G)$ [26]. We say that v matches u if $\langle v, u\rangle \in Q(G)$.

Graph pattern matching via graph simulation is as follows.

- Input: A directed graph G and a pattern Q.
- Output: The unique maximum relation $Q(G)$.

Batch algorithm. There is a fixpoint algorithm Sim_{fp} for Sim [26]. It associates each pair $\langle v, u\rangle \in V \times V_Q$ of nodes with a Boolean variable $x[v, u]$, indicating whether v matches u. Initially $x[v, u]$ is true if $L(v) = L_Q(u)$; otherwise, $x[v, u]$=false. It also employs a counter $\text{cnt}(v, u')$ to record the number of nodes v' in $\text{out\_nbr}(v)$ with $x[v', u']$=true, where $\text{out\_nbr}(v)$ records v's outgoing neighbors. The pairs $\langle v, u'\rangle$ with $\text{cnt}(v, u')$=0 are maintained in a set P, and the scope $H_{\mathcal{A}}$ includes $x[v, u]$ if $\langle v, u'\rangle \in P$ and $u' \in \text{out\_nbr}(u)$. The counters help Sim_{fp} decide the direction of change propagation. Each time step function $f_{\mathcal{A}}$ of Sim_{fp} removes a pair $\langle v, u'\rangle$ from P and invokes $f_{x[v,u]}(Y_{x[v,u]})$ to check the simulation condition for each $\langle v, u\rangle$, where $u' \in \text{out\_nbr}(u)$ and $Y_{x[v,u]}$ is $\{x[v'', u''] \mid v'' \in \text{out\_nbr}(v) \wedge u'' \in \text{out\_nbr}(u)\}$. It updates $x[v, u]$ from true to false if the condition is violated; the counters, set P and scope $H_{\mathcal{A}}$ are also updated accordingly. It terminates when $H_{\mathcal{A}} = \emptyset$. The invariant of Sim_{fp} is that if node v matches u, then there exists a nonzero counter $\text{cnt}(v, u')$ for each outgoing neighbor u' of u.

Incremental algorithm. An incremental algorithm IncSim is *weakly deducible* from Sim_{fp}. Its step function $f_{\mathcal{A}_\Delta}$ is the same as $f_{\mathcal{A}}$ of Sim_{fp} except that it specifies a timestamp $x[v, u].t$ for each variable $x[v, u]$ to record the time when $x[v, u]$ turns to false from true. Initially, $x[v, u].t = -1$ if $x[v, u]$ = false and $x[v, u].t = \infty$ otherwise. The timestamps are used to determine the anchor sets, especially for cyclic patterns [23]. Intuitively, those status variables in $Y_{x[v,u]}$ with timestamps smaller than $x[v, u]$ *essentially* contribute to the invocation of $f_{x[v,u]}$, which also constitute the anchor sets $C_{x[v,u]}$ of $x[v, u]$. The topological order $<_C$ can be deduced accordingly, *i.e.,* $x[v, u] <_C x[v', u']$ if and only if $x[v, u].t < x[v', u'].t$.

We can verify that Sim_{fp} is both contracting and monotonic, by letting $\preceq$ be the partial order on status variables such that false $\preceq$ true. By Theorem 3, it suffices to design a correct and bounded scope function h. Employing the order $<_C$ defined as above, IncSim just adopts the function h shown in Figure 4, which recursively adjusts all unfeasible status variables from false to true to get a revised status $D^0_{\mathcal{A}_\Delta}$ and the scope $H^0_{\mathcal{A}_\Delta}$. After that, IncSim applies step function $f_{\mathcal{A}_\Delta}$ to get the updated result as in Sim_{fp}. It also updates timestamps. Thus IncSim is weakly deducible from Sim_{fp}.

Example 6: Consider the graph pattern depicted in Figure 2(b) and the graph G described in Example 3, where each node in G is labeled 'a', 'b' or 'c'. Algorithm Sim_{fp} finds that G matches Q, and Figure 3(b) lists the resulting timestamps (t-column) and anchor sets. As each node v in G carries a single label $L(v)$, Figure 3(b) only shows information of variables $x[v, L(v)]$; the other variables are false and have empty anchor sets. Here we do not explicitly show the final value of each variable since it is true (resp. false) if the corresponding timestamp is ∞ (resp. an integer).

Given the updates ΔG of Example 4, the scope function h of incremental algorithm IncSim first initializes the scope $H^0_{\mathcal{A}_\Delta}$ as $\{x[5, b]\}$. Since the only outgoing neighbor of node 5 after the update is node 3, and $x[3, c] >_C x[5, b]$ (by $x[3, c].t = \infty$, $x[5, b].t = 2$), h recognizes a new feasible input set as $\overline{Y}_{x[5,b]} = \{x[3, c] = \text{true}\}$. As a result, h changes $x[5, b]$ from false to true. Then the iterative computation of h terminates since $x[5, b]$ is not in any anchor sets. It returns $\{x[5, b]\}$ as $H^0_{\mathcal{A}_\Delta}$, and the feasible status $D^0_{\mathcal{A}_\Delta}$ that differs from previous fixpoint only in $x[5, b]$.

Then IncSim proceeds along the same lines of function $f_{\mathcal{A}}$ of Sim_{fp} to compute the new values for $G \oplus \Delta G$ indicated in Figure 3(b). In particular, the timestamp of $x[5, b]$ is also changed to ∞. □

Analyses. One can verify that $D^0_{\mathcal{A}_\Delta}$ is feasible for $G \oplus \Delta G$, by induction on the order $<_C$ (see Section 4); from this the correctness of IncSim follows. Note that each status variable included in $H^0_{\mathcal{A}_\Delta}$ either has a new value in $G \oplus \Delta G$ or has to be inspected necessarily. Thus $H^0_{\mathcal{A}_\Delta} \subseteq$ AFF and IncSim is bounded relative to Sim_{fp}.

5.2 Depth-First Search

We next consider *depth-first search* (DFS) on directed graphs [43].

- Input: A directed graph $G = (V, E, L)$.
- Output: A DFS tree, *i.e.,* an ordered spanning tree $T_G = (V, E_T)$ generated by depth-first search on G.

Here each node v in the DFS tree is associated with v.first and v.last, indicating v's positions in the preorder and the postorder on the vertices of V induced by the DFS traversal, respectively. In practice, they also record the (relative) timestamps when v is visited for the first time and the last time in the DFS traversal [50], respectively. To simplify the discussion, we assume that there is a virtual root r connected to every node in G and the traversal starts from r.

Batch algorithm. There exists a fixpoint algorithm for DFS that is incrementalizable, denoted by $\mathsf{DFS_{fp}}$. For each node $v \in V \cup \{r\}$, it declares a status variable, *i.e.,* timestamp interval $x_v = [v.\mathsf{first}, v.\mathsf{last}]$. Initially all x_v's are assigned $[\infty, \infty]$ except $x_r = [0, \beta]$, where $\beta > 0$. The update function f_{x_v} adjusts x_v; its input set Y_{x_v} includes $x_{v'}$ for all incoming neighbors v' of v, in which one variable in Y_{x_v} is marked as the *parent* of x_v. The logical statement σ_{x_v} is defined as that there exists no $x_{v'} \in Y_{x_v}$ such that $v'.\mathsf{last} < v.\mathsf{first}$. This is consistent with the invariant of DFS that no forward-cross edge exists [50]. Therefore, initially the scope $H_{\mathcal{A}}$ contains variables x_v's associated with outgoing neighbors of virtual root r, and their parents are x_r. Moreover, x_w.last of x_v's parent x_w is regarded as the rank for each variable $x_v \in H_{\mathcal{A}}$.

In each round, the step function $f_{\mathcal{A}}$ of $\mathsf{DFS_{fp}}$ selects and removes one variable x_v from the scope $H_{\mathcal{A}}$ that has the smallest rank. It changes x_v to a new strict subinterval of $[w.\mathsf{first}, w.\mathsf{last}]$ by calling update function f_{x_v} and links w to v in the DFS tree, where x_w is the parent of x_v. Here the new subinterval does not overlap with previous ones computed from $[w.\mathsf{first}, w.\mathsf{last}]$ and has the largest first value. In addition, for each outgoing neighbor u of v with $v.\mathsf{last} < u.\mathsf{first}$, it adds x_u to the scope $H_{\mathcal{A}}$ and marks x_v as the new parent of x_u. The process iterates until $H_{\mathcal{A}}$ becomes empty.

Incremental algorithm. By Theorem 3, a relatively bounded incremental algorithm, denoted as IncDFS, is *deducible* from $\mathsf{DFS_{fp}}$.

To do this, we first show that $\mathsf{DFS_{fp}}$ is both contracting and monotonic. Recall that the status variables in $\mathsf{DFS_{fp}}$ have the form of closed intervals $[v.\mathsf{first}, v.\mathsf{last}]$. We define a partial order $\preceq$ such that $x_v \preceq x_u$ if v.last is no larger than u.first. One can see that update function f_{x_v} changes x_v to a subinterval of its parent x_w, and w.last is the minimum among all v's incoming neighbors. Therefore, $\mathsf{DFS_{fp}}$ is both contracting and monotonic *w.r.t.* $\preceq$.

It remains to develop a correct and bounded initial scope function h for IncDFS following the guidelines provided in Section 4. By the semantics of update functions in $\mathsf{DFS_{fp}}$, obviously only the (single) parent for status variable x_v is in the anchor set C_{x_v} of x_v. In addition, the order $<_C$ can also be derived from the final values of the status variables. That is, $x_v <_C x_u$ if and only if $v.\mathsf{first} < u.\mathsf{first}$. Using anchor sets (*i.e.,* contributors) and the order $<_C$, h can be deduced following Figure 4 for finding scope $H^0_{\mathcal{A}_\Delta}$ and new status $D^0_{\mathcal{A}_\Delta}$. In particular, when the expansion of scope $H^0_{\mathcal{A}_\Delta}$ stops in h, each status variable x_v in $H^0_{\mathcal{A}_\Delta}$ is given a new parent x_w and a corresponding rank w.last analogous to that in $\mathsf{DFS_{fp}}$, *i.e.,* w.last is the minimum among the incoming neighbors of v.

Having $H^0_{\mathcal{A}_\Delta}$ as the new initial scope, IncDFS reuses the same step function $f_{\mathcal{A}}$ of $\mathsf{DFS_{fp}}$ to compute the new fixpoint. The algorithm does not need to add timestamps, and is deducible from $\mathsf{DFS_{fp}}$.

Example 7: Recall graph G and input updates ΔG from Example 6. Running of $\mathsf{DFS_{fp}}$ over G produces the intervals, *i.e.,* status variables and their anchor sets shown in Figure 3(c), where the anchor sets also indicate the parent of each node in the DFS tree.

When G is updated by ΔG, the initial scope function h includes x_3 and x_6 in the scope $H^0_{\mathcal{A}_\Delta}$ and the queue because of the changed input sets. Observe that $x_6 <_C x_3$; hence h handles x_6 first. As the the only edge to node 6 is from node 4 after the update, *i.e.,* the old parent x_5 of x_6 no longer exists in the new feasible input set $\overline{Y}_{x_6} = \{x_4 = [\infty, \infty]\}$, h resets x_6 to the initial value $[\infty, \infty]$. It also puts x_7 into the queue since $x_6 \in C_{x_7}$. Similarly, x_7 is reset to $[\infty, \infty]$ by h. Function h finally processes x_3. Its feasible input set $\overline{Y}_{x_3}$ is $\{x_4 = [8, 11], x_5 = [2, 7]\}$. Thus $x_3 = [9, 10] \not\preceq f_{x_3}(\overline{Y}_{x_3}) = [3, 4]$ and no other nodes are pushed into the queue for further inspection.

Function h returns scope $H^0_{\mathcal{A}_\Delta}$ that consists of status variables x_3, x_6 and x_7. Their new parents are x_5, x_4 and x_6, respectively. Using the step function $f_{\mathcal{A}}$ of $\mathsf{Sim_{fp}}$, IncSim derives the new intervals and parents, *i.e.,* anchor sets in $G \oplus \Delta G$, also shown in Fig. 3(c). □

Analyses. The correctness of IncDFS is assured by the contracting and monotonic computation of $\mathsf{DFS_{fp}}$, and the correctness of scope function h (see Section 4). IncDFS is bounded relative to $\mathsf{DFS_{fp}}$ since for each x_v in $H^0_{\mathcal{A}_\Delta}$, either it has new value or its update function has evolved input sets, *i.e.,* $H^0_{\mathcal{A}_\Delta} \subseteq \mathsf{AFF}$ (see the proof of Theorem 3).

5.3 Local Clustering Coefficient

Finally, we study *local clustering coefficient* (LCC) [47].

- Input: An undirected graph $G = (V, E, L)$.
- Output: The local clustering coefficient γ_v for each vertex v in V, where γ_v is defined as

$$\gamma_v = \frac{2|\{(u, w) \in E \mid u, w \in \mathsf{nbr}(v)\}|}{d_v(d_v - 1)}.$$

Here d_v is the degree of v and $\mathsf{nbr}(v)$ is the neighbor set of v. Intuitively, γ_v measures how close v and its neighbors are to forming a clique, *e.g.,* when $\gamma_v = 1$, v and its neighbors form a clique. Let λ_v be the number of triangles including v. Then $\gamma_v = 2\lambda_v / d_v(d_v - 1)$.

Batch Algorithm. A fixpoint algorithm [41, 42] for LCC, denoted as $\mathsf{LCC_{fp}}$, works as follows. It associates each node $v \in V$ with two status variables d_v and λ_v. The update function f_{d_v} (resp. f_{λ_v}) computes their values based on the definitions above; their logical statements are defined accordingly. Initially the scope $H_{\mathcal{A}}$ includes all variables. The step function $f_{\mathcal{A}}$ of $\mathsf{LCC_{fp}}$ simply removes each d_v and λ_v from $H_{\mathcal{A}}$ and determines their values using update functions. Then coefficients γ_v can be readily obtained.

Incremental algorithm. A relatively bounded incremental algorithm IncLCC is *deducible* from $\mathsf{LCC_{fp}}$, without using timestamps. Given updates ΔG, it first identifies PE variables that have different values in the two runs of the batch $\mathsf{LCC_{fp}}$ (see Section 4). For each edge (u, v) involved in ΔG, it marks d_u, d_v and $\lambda_{v'}$ as PE variables, where v' ranges over all nodes that are within one-hop from u or v. All PE variables are included in the new scope $H_{\mathcal{A}_\Delta}$. Thereafter, IncLCC applies the original step function $f_{\mathcal{A}}$ of LCC_{fp} to determine the new values for the status variables in the scope $H_{\mathcal{A}_\Delta}$.

Example 8: Consider applying LCC_{fp} on graph G of Figure 2(a). The status variables d_v and λ_v for each node is shown in Figure 3(d) under the G-column. When processing input updates ΔG of Example 4, the incremental algorithm IncLCC derives $H_{\mathcal{A}_\Delta}$ as $\{d_3, d_5, d_6\} \cup \{\lambda_i \mid i \in [1,7]\}$. That is, $H_{\mathcal{A}_\Delta}$ includes variables d_v for v covered by ΔG, and λ_v for v within one-hop from ΔG. IncLCC next enforces step function $f_{\mathcal{A}}$ to update the variables in $H_{\mathcal{A}_\Delta}$ to the values listed under the $G \oplus \Delta G$-column of Figure 3(d). □

Analyses. Algorithm IncLCC is correct by Theorem 1. The status variable d_v (resp. λ_v) in LCC_{fp} is affected by updates ΔG, *i.e.,* $d_v \in$ AFF (resp. $\lambda_v \in$ AFF) if and only if there exist changes to the adjacent edges of v (resp. v and its neighbors). Note that IncLCC only collects such status variables into scope $H_{\mathcal{A}_\Delta}$ as PE variables; from this it follows that it is bounded relative to LCC_{fp}.

6 EXPERIMENTAL STUDY

We empirically evaluated our incrementalized algorithms of SSSP, CC, Sim, DFS and LCC, for their efficiency and effectiveness when processing (1) unit updates and (2) batch updates, (3) the scalability with larger synthetic graphs, and (4) memory cost.

Experiment setting. We start with graphs and updates.

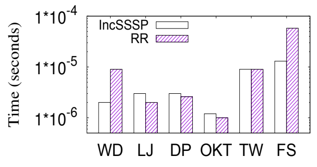
Datasets. We used six real-life graphs of different types, including (a) LiveJournal (LJ) [3, 11, 31], a social network with 4.8 million users and 68.9 million relationships, (b) DBPedia (DP) [6], a knowledge base with 4.9 million entities and 54 million edges, (c) Orkut (OKT) [1, 37], a social network of 3.1 million users and 117 million connections between users, (d) Twitter-2010 (TW) [4, 13, 14], a social network with 41.6 million nodes and 1.4 billion links, (e) Friendster (FS) [2, 51], a gaming network with 65.6 million users and 1.8 billion edges, and (f) Wiki-DE (WD) [5, 30], a real-life temporal graph with 2.1 million nodes and 86.3 million links, which record the evolution of hyperlinks between articles in German Wikipedia. The edges in the temporal WD are labeled with timestamps indicating when they were last added or removed.

To test the scalability, we also designed a generator to produce larger synthetic graphs $G = (V, E, L)$, controlled by the number $|V|$ of nodes (up to 135 million) and the number $|E|$ edges (up to 2.1 billion) with L drawn from an alphabet of 5 labels.

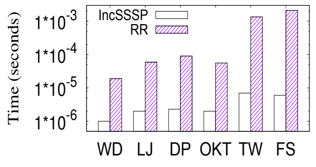
Updates. For the temporal graph WD, we constructed updates ΔG from real timestamped changes by limiting certain time intervals. For the other graphs, we generated random updates controlled by the size $|\Delta G|$. The random updates were comprised of equal amount of edge insertions and deletions, unless stated otherwise.

Queries. We sampled 20 source nodes from each graph to create SSSP queries. For Sim, we constructed 5 patterns $Q=(V_Q, E_Q, L_Q)$ on each graph with labels drawn from the data graphs.

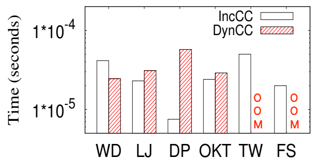
Implementation. We implemented the following, all in C++: (1) the incrementalized algorithms given in Sections 3–5, *i.e.,* IncSSSP, IncCC, IncSim, IncDFS and IncLCC; (2) their variants IncSSSP_n, IncCC_n, IncSim_n, IncDFS_n, and IncLCC_n, which process unit updates in given batch updates ΔG one by one using the corresponding



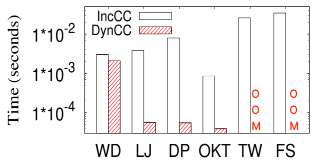
(a) Unit insertions: SSSP



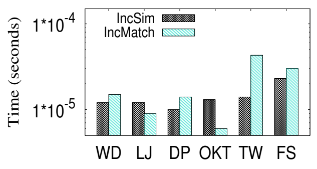
(b) Unit deletions: SSSP



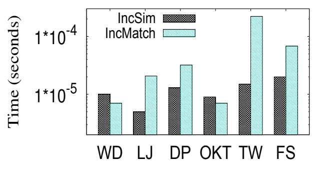
(c) Unit insertions: CC



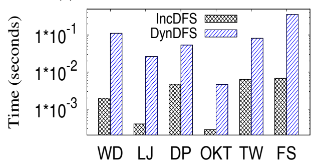
(d) Unit deletions: CC



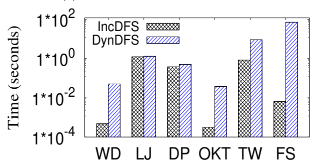
(e) Unit insertions: Sim



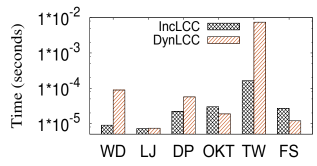
(f) Unit deletions: Sim



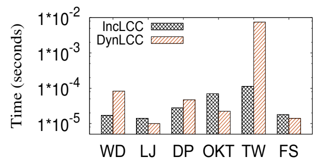
(g) Unit insertions: DFS



(h) Unit deletions: DFS



(i) Unit insertions: LCC



(j) Unit deletions: LCC

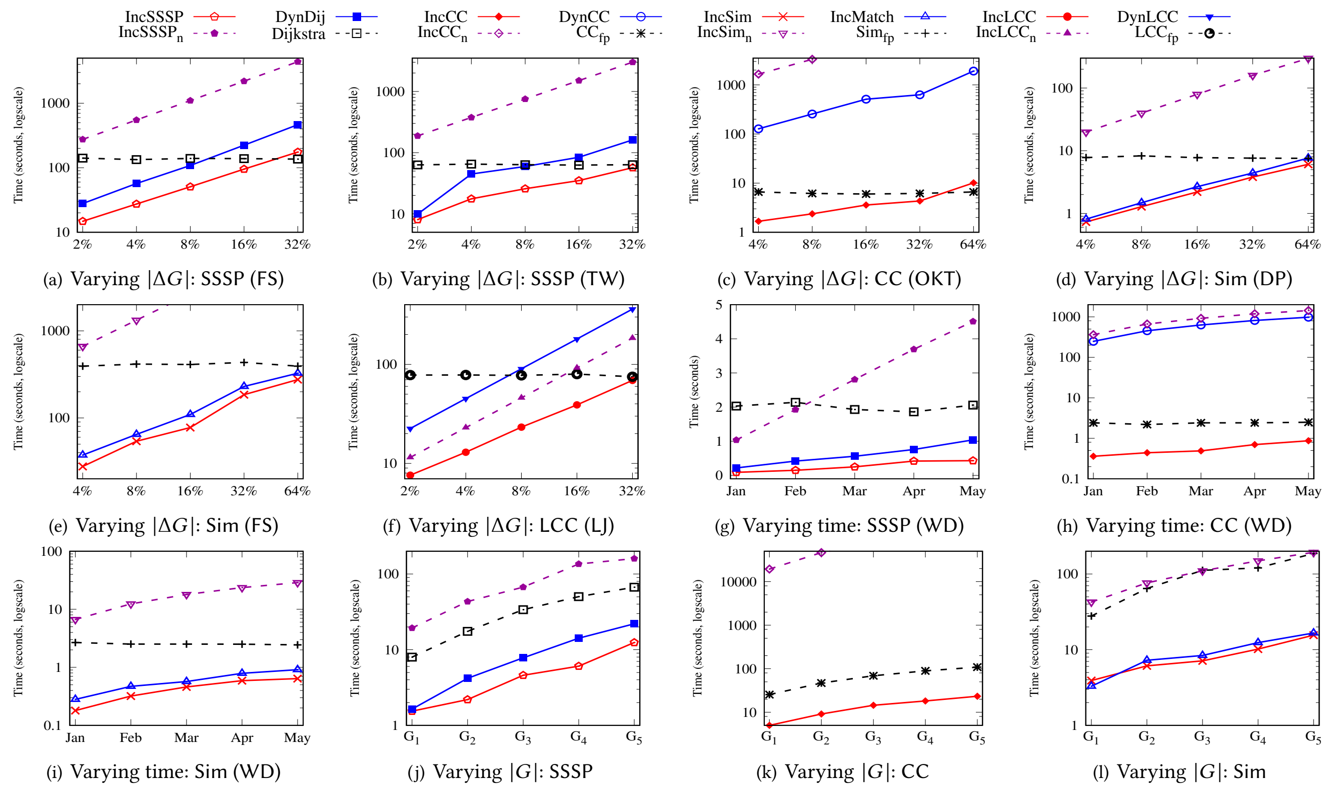
Figure 6: Efficiency in processing unit updates

techniques developed in this work; (3) their batch counterparts, *i.e.,* fixpoint algorithms Dijkstra, CC_{fp}, Sim_{fp}, DFS_{fp} and LCC_{fp}, which are also described in Sections 3–5; and (4) existing incremental (dynamic) algorithms for different graph query classes, including (a) the dynamic SSSP algorithms in [39] and [17] for processing unit updates and batch updates, denoted as RR and DynDij, respectively; (b) the fully dynamic algorithm of [27] for CC, denoted as DynCC (we used the implementation provided at [7]); (c) IncMatch [23] for Sim; (d) the fully dynamic algorithm DynDFS for DFS [50]; and (e) the streaming LCC algorithm in [19], denoted as DynLCC.

All experiments were conducted on one single processor of Intel Xeon 2.20 GHz CPU, with 128 GB memory, running Red Hat Enterprise Linux Server 7.3. All the implementations are single threaded. Each experiment was repeated 5 times. The average is reported here.

Experimental results. We next report our findings. When testing Sim, we fixed $|Q| = (4, 6)$, *i.e.,* patterns with 4 nodes and 6 edges.

Exp-1: Efficiency. We first evaluated the efficiency of our incrementalized algorithms in processing unit updates, compared with existing dynamic methods for various query classes. This is to study the impact of different types of ΔG on the performance of incremental algorithms. We used all the six real-life graphs and sampled



(a) Varying $|\Delta G|$: SSSP (FS) (b) Varying $|\Delta G|$: SSSP (TW) (c) Varying $|\Delta G|$: CC (OKT) (d) Varying $|\Delta G|$: Sim (DP)

(e) Varying $|\Delta G|$: Sim (FS) (f) Varying $|\Delta G|$: LCC (LJ) (g) Varying time: SSSP (WD) (h) Varying time: CC (WD)

(i) Varying time: Sim (WD) (j) Varying $|G|$: SSSP (k) Varying $|G|$: CC (l) Varying $|G|$: Sim

Figure 7: Performance evaluation in processing batch updates

10000 edge insertions (resp. deletions) for each graph as unit updates ΔG. We applied unit updates one by one and recorded the average runtime for handling each edge insertion (resp. deletion).

(1) Edge insertion. Figures 6(a), 6(c), 6(e), 6(g) and 6(i) report the performance of different incremental methods for SSSP, CC, Sim, DFS and LCC, respectively. From the results we can find the following.

(a) In most cases, our incrementalized algorithms outperform the existing dynamic methods for edge insertions, despite that they are systematically deduced from batch algorithms, while the baselines are fine-tuned individually with possibly sophisticated techniques for specific queries. On average IncSSSP (resp. IncCC, IncSim, IncDFS, IncLCC) is 2.6 (resp. 1.3, 1.4, 31.0, 29.4) times faster than RR (reps. DynCC, IncMatch, DynDFS, DynLCC).

(b) The incrementalized methods are efficient. Over the largest real-life graph FS, the average response time of all our incremental methods is less than 7ms for unit edge insertions.

(c) The size |AFF| of affected area for unit insertions is rather small. For instance, over OKT, |AFF| accounts for only 1.8×10^{-6}%, 4.2×10^{-5}%, 1.7×10^{-6}%, 2.0×10^{-6}% and 2.6×10^{-3}% of the total size of the auxiliary structures on average, for IncSSSP, IncCC, IncSim, IncDFS and IncLCC, respectively (not shown).

(2) Edge deletion. As reported in Figures 6(b), 6(d), 6(f), 6(h) and 6(j) for the five query classes, respectively, (a) in most cases the incrementalized algorithms perform better than the existing dynamic methods, *e.g.,* IncLCC is 1.7 times faster than DynLCC over DP. (b) While DynCC works better than IncCC for unit edge deletions on WD, LJ, DP and OKT, it ran out of memory (OOM) on larger real-life graphs TW and FS for its excessive memory usage, no matter whether for insertions or deletions. (c) The size |AFF| for unit deletions is also small, *e.g.,* over OKT, it is 1.8×10^{-6}%, 2.4×10^{-3}%, 1.7×10^{-6}%, 5.9×10^{-5}% and 2.5×10^{-5}% of the total size of auxiliary structures, for IncSSSP, IncCC, IncSim, IncDFS and IncLCC, respectively. Note that the |AFF| for unit deletions of IncCC and IncDFS is much larger than that for unit insertions, which explains why they take longer when dealing with deletions.

These show that the incrementalization approach suffices to deduce efficient incremental algorithms from batch algorithms.

Exp-2: Effectiveness in processing batch updates. We further evaluated the performance of the deduced algorithms against their batch counterparts and competitors. We studied the impact of the size and the type of updates, respectively. In all the cases tested, we find that the incrementalized algorithms consistently perform better than their variants that handle unit updates one by one.

(1) Varying $|\Delta G|$. We first tested (a) SSSP with real-life graphs FS and TW, (b) CC with OKT and LJ, (c) Sim with DP and FS, (d) LCC with LJ and OKT, and (e) DFS with OKT, based on their applications in social networks, knowledge graphs and community detection.

(a) SSSP. Varying $|\Delta G|$ from 2% to 32% over FS and TW, we report the results in Figures 7(a) and 7(b), which tell us the following.

(i) On average, the incrementalized algorithm IncSSSP is from 20.7 to 31.0 (resp. 1.5 to 2.7) times faster than IncSSSP$_n$ (resp. DynDij) when $|\Delta G|$ varies from 2% to 32% of $|G|$ on FS and TW. IncSSSP$_n$ is even slower than batch Dijkstra when $|\Delta G| \geq 2\%$. This is because it processes batch updates as a sequence of unit updates one by one, which is inefficient. In addition, the gap between the runtime of IncSSSP and DynDij gets larger when $|\Delta G|$ increases.

(ii) IncSSSP outperforms batch counterpart Dijkstra from 9.5 (resp. 7.8) to 1.5 (resp. 1.8) times when $|\Delta G|$ varies from 2% to 16% on FS (resp. TW). It is faster than Dijkstra even when $|\Delta G| = 32\%|G|$.

(iii) All the incremental algorithms IncSSSP, IncSSSP_n and DynDij take less time on smaller updates, as expected, while Dijkstra is insensitive to $|\Delta G|$, since the sizes of the graphs remain stable.

(b) CC. Figure 7(c) reports the performance of different algorithms for CC on OKT. We can see that (i) incremental IncCC consistently outperforms batch counterpart CC_fp when $|\Delta G|$ is up to 32% of $|G|$; the improvement is 4.0 times when $|\Delta G| = 4\%|G|$. (ii) On average, IncCC beats DynCC by 132 times when $|\Delta G|$ varies from 4% to 64% of $|G|$, as opposed to the case of unit edge deletions. (iii) Surprisingly, DynCC is even slower than batch CC_fp that recomputes answers starting from scratch. This is because DynCC processes unit updates in batch updates one by one, instead of treating them as a whole. The results on LJ are consistent (not shown).

(c) Sim. As reported in Figures 7(d) and 7(e) over DP and FS, respectively, (i) incremental algorithms IncSim and IncMatch consistently beat batch Sim_fp when $|\Delta G| \leq 64\%|G|$; IncSim is 10.7 (resp. 14.2) times faster than Sim_fp, on DP (resp. FS) with 4% of updates. (ii) IncSim and IncMatch scale better with $|\Delta G|$ than IncSim_n, while Sim_fp is insensitive to $|\Delta G|$, as expected. (iii) IncSim is on average 18% and 28% faster than the fine-tuned IncMatch on DP and FS, respectively. (iv) The two take 3.8s and 4.4s when $|\Delta G|$ is up to $32\%|G|$ on DP, respectively, while the batch algorithm takes 7.6s.

(d) LCC. As shown in Fig. 7(f) over LJ, when $|\Delta G|$ is varied from 2% to 32% of $|G|$, (i) on average IncLCC is 4.5 and 2.1 times faster than LCC_fp and IncLCC_n, respectively; IncLCC outperforms batch LCC_fp even when $|\Delta G|$ is up to 32% of $|G|$. (ii) IncLCC consistently outperforms DynLCC, *e.g.*, IncLCC is 2.7 times faster than DynLCC when $|\Delta G| = 2\%|G|$. The gap gets larger when $|\Delta G|$ increase. The results on OKT are consistent (not shown).

(e) DFS. Over OKT, IncDFS performs much better than its batch counterpart DFS_fp when $|\Delta G| \leq 1\%|G|$, *i.e.*, on average 0.53s versus 1.64s (not shown). IncDFS is also 4.4 times faster than DynDFS when $|\Delta G| = 1\%|G|$. However, it takes longer than DFS_fp when $|\Delta G| > 4\%|G|$. In fact, compared to other graph query classes such as Sim, small input updates easily affect a larger percentage of the prior computation of DFS, *e.g.*, the traversal order of the nodes.

These results verify the effectiveness of incrementalized algorithms, and are consistent with their relative boundedness.

(2) Types of updates. Besides the random mixed updates adopted above, we further evaluated the impact of the real-life updates on the performance of the incremental algorithms. We extracted real-life updates for the temporal graph WD by inspecting its status over 5 months in 2011. Here the updates within a month on average account for 1.9% of $|G|$, in which 81% of updates are edge insertions and 19% are edge deletions. We find the following from the results in Figures 7(g)–7(i) for SSSP, CC and Sim, respectively.

(a) The results on real-life updates are consistent with the results on randomly generated updates: IncSSSP, IncCC and IncSim are

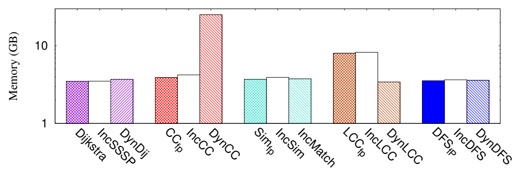


Figure 8: Memory usage (OKT)

substantially faster than their batch counterparts; the improvement is on average 10.7, 4.6 and 7.2 times, respectively.

(b) IncSSSP, IncCC and IncSim are 2.2, 1093 and 1.4 times faster than DynDij, DynCC and IncMatch on average, respectively. As remarked in Exp-2 (1), DynCC does not handle batch updates well.

(c) The incrementalized algorithms handle batch updates much better than the methods that process unit insertions and deletions one by one in a batch update. Over 5 months of real-life updates to WD, IncSSSP (resp. IncCC, IncSim) takes 0.43s (resp. 0.87s, 0.64s), while none of IncSSSP_n, IncCC_n and IncSim_n terminates in 4.6s.

(d) The scope function h on average takes 47% (resp. 92% and 83%) of the total cost of IncSSSP (resp. IncCC and IncSim; not shown) on WD. It dominates the computation in IncCC because only less than 5.6% of the affected area is outside ΔG. This is also due to the power-law node degree distribution of WD, which easily results in stable connected components. Therefore, only a small fraction of previous fixpoint is updated by the step function of IncCC, while scope function h has to scan the entire input updates.

The results for DFS and LCC are consistent and are not shown.

Exp-3: Scalability. We also evaluated the scalability of the incrementalized algorithms over larger synthetic graphs. Fixing $|\Delta G| = 1\%|G|$, we tested SSSP, CC and Sim on synthetic graphs by varying $|G| = |V|+|E|$ from 0.5 billion to 2.2 billion, respectively. As reported in Figures 7(j) to 7(l), (a) our deduced algorithms scale with $|G|$ as well as their batch counterparts. (b) They are also efficient: when G consists of 2.2 billions of nodes and edges, IncSSSP (resp. IncCC, IncSim) takes 12.44s (resp. 23.37s, 15.55s), as opposed to 99.9s (resp. 109.1s, 168.7s) by batch algorithm Dijkstra (resp. CC_fp, Sim_fp). (c) IncSSSP and IncSim are on average 1.8 and 1.1 times faster than DynDij and IncMatch, respectively. For CC, DynCC cannot handle large graphs. This is consistent with the results in Exp-1 and Exp-2. The results on IncDFS and IncLCC are consistent (not shown).

Exp-4: Space cost. Fixing $|\Delta G| = 1\%|G|$, Figure 8 shows the memory usage of different algorithms in processing batch updates over OKT. Observe that (1) the deducible incremental algorithms IncSSSP, IncDFS and IncLCC require no more space than their batch counterparts Dijkstra, DFS_fp and LCC_fp, respectively, as expected (Section 3). (2) The space cost of weakly deducible IncCC and IncSim is comparable to their batch counterparts; *e.g.*, IncSim needs only 6% more memory than that of Sim_fp. (3) Our incrementalized algorithms require space that is comparable to or less than the existing dynamic methods, except DynLCC; in practice, most existing dynamic methods trade off space for runtime, but DynLCC is a stream algorithm that trades runtime for space.

Summary. We find the following. (1) Our incrementalized algorithms consistently outperform their batch counterparts in response to unit updates and small batch updates. On average, IncSSSP (resp.

IncCC, IncSim, IncLCC) is 4.3 (resp. 4.0, 12.4, 6.0) times faster than its batch counterpart when $|\Delta G| = 4\%|G|$. (2) They scale well with $|G|$, *e.g.,* IncSSSP completes in 12.44s on a graph of size 2.2 billion with 1% updates, while Dijkstra takes 99.9s. (3) When processing batch updates, IncSSSP (resp. IncCC, IncSim, IncDFS, IncLCC) outperforms existing fine-tuned DynDij (resp. DynCC, IncMatch, DynDFS, DynLCC), *e.g.,* by 1.8 (resp. 38.9, 1.2, 4.4, 2.7) times when $|\Delta G| = 1\%|G|$. (4) The deduced algorithms require space comparable to both their batch counterparts and existing dynamic algorithms.

7 RELATED WORK

Incrementalization. There has been work on incrementalizing programs, *e.g.,* [10, 15, 32, 36], often at the instruction level. Self-adjusting computation [10] memoizes intermediate results and tracks the dependencies of the computation. It handles updates via a change propagation algorithm. A static method was proposed in [32] to transform programs written in a first-order functional language into their incremental versions at compile-time. The transformation reuses previous intermediate results and employs complicated reasoning techniques. Another static approach is to map the changes in the program's input directly to the changes in the output, using derivatives of the program [15]. As implemented in Naiad [38], differential dataflow [36] extends incremental computation to support nested iterations, and allows the state of the computation to vary based on a partial order of versions. Apart from these, PowerLog [46] establishes fundamental results for checking whether a recursive program can be executed incrementally.

There has also been work on incrementalizing vertex-centric graph algorithms. A new message passing policy is presented in [52] to exchange meaningful results via Δ-messages. While it reduces messages, changes to input graphs are not considered. GraphInc [16] and HBSP model [48] apply memoization to save and reuse previous computations. There are also dependency-driven streaming frameworks, *e.g.,* [35, 45]. KickStarter [45] studies monotonic algorithms and computes safe approximation results upon edge deletions, to fix approximation errors via iterative computation. GraphBolt [35] tracks dependencies using memoized aggregation values. When input updates arrive, it refines the dependencies iteration-by-iteration to do incremental computation.

This work differs from the prior work in the following. (1) We target graph-centric algorithms, beyond the vertex-centric ones inspected in *e.g.,* [16, 45, 48, 52]. (2) We deduce incremental graph algorithm $\mathcal{A}_\Delta$ by reusing the same logic of its batch counterpart $\mathcal{A}$, not by costly memoization [16, 35, 48]. Our proposed method is also beyond the instruction-level considered in [10, 15, 32]; all of our components involved, including the initial scope function (Section 3), can be deduced from the batch algorithm without complicated analyses of instructions. (3) In addition to correctness, we provide performance guarantee for the incrementalization method in efficiency, which is not studied in most previous work. (4) The dependencies of the computation in our incrementalized algorithms are not limited to trees as required by KickStarter [45].

Closer to this work are [20, 21, 23]. While [20] studies practical measures for evaluating the effectiveness of incremental graph algorithms, [21] and [23] focus on graph partitioner incrementalization and incremental graph simulation, respectively. They substantially differ from this work in the following. (1) Proposing the notion of relative boundedness, [20] defines *what* incremental algorithms are effective and shows the existence of such algorithms for certain problems that are unbounded [39]. In contrast, this work answers *how* to incrementalize algorithms with relative boundedness, by proposing a systematic method. (2) We also identify what graph algorithms are incrementalizable, and provide generic conditions under which a deduced algorithm guarantees to be both correct and relatively bounded. No such conditions are studied in [20, 21, 23]. (3) We consider generic graph algorithms, way beyond (heuristic) partitioners [21] and graph simulation [23]. (4) Our method uses timestamps at most as auxiliary structure. In contrast, various types of additional data structures are needed to implement the incremental algorithms of [20, 21, 23]. (5) Unlike [20, 21, 23] that handle edge insertions and deletions separately by different methods, our approach proposes an initial scope function to uniformly process both types of updates. This relieves the users' burden since they only need to implement a single initial scope function.

Effectiveness measures of incremental algorithms. The costs of incremental algorithms are mostly measured by averaged operation time in response to a sequence of updates using amortized analysis (see [18] for a survey). In contrast to the conventional measure, [39] introduces a notion of boundedness to decide whether the cost of an incremental algorithm can be expressed as a function of the size of the changes to the input and output. However, the criterion is too strong and most incremental algorithms are unbounded. Two weaker standards, *i.e.,* semi-boundedness [23] and relative boundedness [20], characterize incremental algorithms by verifying whether they only access the data that is necessarily checked by any incremental algorithm or they visit the difference in the data inspected by batch counterpart algorithms during two runs alone. Heuristic boundedness of [21] classifies incremental non-deterministic partitioners. This work adopts the relative boundedness of [20] as the effectiveness criterion and aims to provide a systematic method for developing incremental algorithms while satisfying this criterion.

8 CONCLUSION

We have proposed to incrementalize batch graph algorithms $\mathcal{A}$ by reusing the same logic and data structures of $\mathcal{A}$. We have identified a class of fixpoint algorithms that are incrementalizable. We have developed conditions under which the deduced algorithms $\mathcal{A}_\Delta$ are guaranteed correct and relatively bounded. We have shown how to deduce relatively bounded $\mathcal{A}_\Delta$ from $\mathcal{A}$, using either no additional auxiliary structures or at most timestamps. Our experiments have verified that the deduced $\mathcal{A}_\Delta$'s perform better than manually fine-tuned incremental baselines in efficiency and comparably in space.

The work is only a step towards incrementalizing graph algorithms. As a topic for future work, we are currently developing a tool for assisting users to develop initial scope function h and revise step function $f_{\mathcal{A}_\Delta}$. We are also extending the class Φ of fixpoint algorithms to make more graph algorithms incrementalizable.

Acknowledgments. Fan is supported in part by ERC 652976 and Royal Society Wolfson Research Merit Award WRM/R1/180014. This work is also supported in part by Alibaba Group through Alibaba Innovative Research Program.

REFERENCES

[1] 2007. Orkut. *http://konect.uni-koblenz.de/networks/orkut-links.*

[2] 2012. Friendster. *https://snap.stanford.edu/data/com-Friendster.html.*

[3] 2012. LiveJournal. *http://snap.stanford.edu/data/com-LiveJournal.html.*

[4] 2012. Twitter-2010. *http://law.di.unimi.it/webdata/twitter-2010/.*

[5] 2012. Wiki-de. *http://konect.uni-koblenz.de/networks/link-dynamic-dewiki.*

[6] 2014. DBpedia. *http://wiki.dbpedia.org/Downloads2014.*

[7] 2020. *https://github.com/yogesh1q2w/Dynamic-Graph-Algorithms.*

[8] 2020. GRAPE. *https://github.com/alibaba/libgrape-lite.git.*

[9] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley.

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

[11] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: Membership, growth, and evolution. In *SIGKDD.*

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

[13] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *WWW.*

[14] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *WWW.*

[15] 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.*

[16] Zhuhua Cai, Dionysios Logothetis, and Georgos Siganos. 2012. Facilitating real-time graph mining. In *CloudDB.*

[17] Edward P. F. Chan and Yaya Yang. 2009. Shortest Path Tree Computation in Dynamic Graphs. *IEEE Trans. Computers* 58, 4 (2009), 541–557.

[18] Camil Demetrescu, David Eppstein, Zvi Galil, and Giuseppe F Italiano. 2010. Dynamic graph algorithms. In *Algorithms and theory of computation handbook.*

[19] David Ediger, Karl Jiang, E. Jason Riedy, and David A. Bader. 2010. Massive streaming data analytics: A case study with clustering coefficients. In *IPDPS Workshop.*

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

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

[22] Wenfei Fan, Xueli Liu, Ping Lu, and Chao Tian. 2018. Catching Numeric Inconsistencies in Graphs. In *SIGMOD.*

[23] Wenfei Fan, Xin Wang, and Yinghui Wu. 2013. Incremental graph pattern matching. *ACM Trans. Database Syst.* 38, 3 (2013), 18:1–18:47.

[24] Wenfei Fan, Wenyuan Yu, Jingbo Xu, Jingren Zhou, Xiaojian Luo, Qiang Yin, Ping Lu, Yang Cao, and Ruiqi Xu. 2018. Parallelizing Sequential Graph Computations. *ACM Trans. Database Syst.* 43, 4 (2018), 18:1–18:39.

[25] M. L. Fredman and R. E. Tarjan. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *J.ACM* 34 (1987), 596–615.

[26] M. R. Henzinger, T. Henzinger, and P. Kopke. 1995. Computing Simulations on Finite and Infinite Graphs. In *FOCS.*

[27] 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.

[28] George Karypis and Vipin Kumar. 1998. Multilevel k-way Partitioning Scheme for Irregular Graphs. *J. Parallel Distributed Comput.* 48, 1 (1998), 96–129.

[29] W. G. Kropatsch. 1995. Building irregular pyramids by dual-graph contraction. *IEE Proceedings - Vision, Image and Signal Processing* 142, 6 (1995), 366–374.

[30] Jérôme Kunegis. 2013. KONECT: the Koblenz network collection. In *WWW.*

[31] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. 2009. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics* 6, 1 (2009), 29–123.

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

[33] Andreas Loukas and Pierre Vandergheynst. 2018. Spectrally Approximating Large Graphs with Smaller Graphs. In *ICML.*

[34] 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.*

[35] Mugilan Mariappan and Keval Vora. 2019. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *EuroSys.*

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

[37] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and Analysis of Online Social Networks. In *IMC.*

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

[39] G. Ramalingam and Thomas Reps. 1996. On the Computational Complexity of Dynamic Graph Problems. *Theor. Comput. Sci.* 158, 1-2 (1996).

[40] G. Ramalingam and Thomas W. Reps. 1996. An Incremental Algorithm for a Generalization of the Shortest-Path Problem. *J. Algorithms* 21, 2 (1996), 267–305.

[41] Thomas Schank and Dorothea Wagner. 2005. Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Study. In *WEA.*

[42] Siddharth Suri and Sergei Vassilvitskii. 2011. Counting triangles and the curse of the last reducer. In *WWW.*

[43] Robert Endre Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160.

[44] Tim Teitelbaum and Thomas W. Reps. 1981. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. *Commun. ACM* 24, 9 (1981), 563–573.

[45] Keval Vora, Rajiv Gupta, and Guoqing (Harry) Xu. 2017. KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. In *ASPLOS.*

[46] 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.*

[47] Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of 'small-world' networks. *Nature* 393, 6684 (1998), 440–442.

[48] Charith Wickramaarachchi, Charalampos Chelmis, and Viktor K. Prasanna. 2015. Empowering Fast Incremental Computation over Large Scale Dynamic Graphs. In *IPDPS.*

[49] Lingkun Wu, Xiaokui Xiao, Dingxiong Deng, Gao Cong, Andy Diwen Zhu, and Shuigeng Zhou. 2012. Shortest Path and Distance Queries on Road Networks: An Experimental Evaluation. *PVLDB* 5, 5 (2012), 406–417.

[50] Bohua Yang, Dong Wen, Lu Qin, Ying Zhang, Xubo Wang, and Xuemin Lin. 2019. Fully Dynamic Depth-First Search in Directed Graphs. *PVLDB* 13, 2 (2019), 142–154.

[51] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems* 42, 1 (2015), 181–213.

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

[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] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: A Comprehensive Graph Neural Network Platform. *PVLDB* 12, 12 (2019), 2094–2105.