Data Science and Engineering (2021) 6:339–357
https://doi.org/10.1007/s41019-021-00158-0

Maximizing Influence Over Streaming Graphs with Query Sequence

Yuying Zhao[1] · Yunfei Hu[1] · Pingpeng Yuan[1] · Hai Jin[1]

Received: 6 January 2021 / Revised: 24 March 2021 / Accepted: 17 April 2021 / Published online: 29 May 2021

Abstract
Now, with the prevalence of social media, such as Facebook, Weibo, how to maximize influence of individuals, products, actions in new media is of practical significance. Generally, maximizing influence first needs to identify the most influential individuals since they can spread their influence to most of others in the social media. Many studies on influence maximization aimed to select a subset of nodes in static graphs once. Actually, real graphs are evolving. So, influential individuals are also changing. In these scenarios, people tend to select influential individuals multiple times instead of once. Namely, selections are raised sequentially, forming a sequence (query sequence). It raises several new challenges due to changing influential individuals. In this paper, we explore the problem of Influence Maximization over Streaming Graph (SGIM). Then, we design a compact solution for storing and indexing streaming graphs and influential nodes that eliminates the redundant computation. The solution includes Influence-Increment-Index along with two sketch-centralized indices called Influence-Index and Reverse-Influence-Index. Computing influence set of nodes will incur a large number of redundant computations. So, these indices are designed to keep track of the nodes' influence in sketches. Finally, with the indexing scheme, we present the algorithm to answer SGIM queries. Extensive experiments on several real-world datasets demonstrate that our method is competitive in terms of both efficiency and effectiveness owing to the design of index.

Keywords Influence maximization · Network diffusion · Dynamic · Sketch · Index

1 Introduction

With the prevalence of social networks, more and more people are engaged in online activities where they interact with each other and produce an unprecedented amount of content. By making better use of these data, we can improve our comprehension of information diffusion which plays a significant role in a variety of practical applications including rumor control [1, 2], social recommendation [3], and business performance optimization [4]. One of the most extensively studied problems of social networks is Influence Maximization (*IM*) [5] which originates from viral marketing [6]. It aims to select a subset of individuals to adopt a new product and trigger a large cascade of further adoptions. This problem has attracted researchers from different fields ever since its formulation. Many effective approaches have been proposed over the last decade.

Most of them are under the assumption of static networks. However, the social networks, rapidly evolving in the real world [7], are innately dynamic. The fact that the classical *IM* fails to capture the dynamics of these networks motivates many researchers to pursue the answers in dynamic scenarios from different perspectives. Some extend the original problem from static scenarios to dynamic ones [8–10]. They try to reuse some intermediate data to avoid recomputing from scratch, so that the target set can be calculated efficiently when the network changes. And some others propose new models in consideration of different factors like data recency [11] and user distinction [12]. While researchers have explored the dynamic aspect of network to some extent, most of them ignore the fact that not only the graph

✉ Pingpeng Yuan
ppyuan@hust.edu.cn

Yuying Zhao
yyzhao@hust.edu.cn

Yunfei Hu
yunfeihu@hust.edu.cn

Hai Jin
hjin@hust.edu.cn

[1] National Engineering Research Center for Big Data Technology and System, Cluster and Grid Computing Lab, Services Computing Technology and System Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China

is evolving but also the requirement of user selections (queries) are raised sequentially which form a query sequence. From a more practical standpoint, queries are prevalent in real life. There are many rounds of promotions to advertise the products rather than merely one round. Every time when promoters launch viral marketing campaigns for promotion purpose, a set of users are selected. Previous works regard different queries as isolated events. In this way, when analysis is carried out in a static angle, the users who were selected previously are highly likely to be chosen repetitively due to their centrality. As a result, selections that have large "overlaps" will be generated. This will definitely dampen the cascade effect of further adoptions, since users who have been influenced might experience a period that is similar to *refractory period* in physiology. During *refractory period*, a body organ or cell is incapable of repeating a particular action or reacting to a repetitive stimulation. The effect of each selection can be viewed as a stimulate to evoke users' desire to shop. Like the viscera, influenced users tend to be hard to be influenced again, since they are still "activated" or they show a dwindling appetite for similar stimulus in short terms. Therefore, selections with large overlapping cannot guarantee the effectiveness among several queries.

As mentioned before, regarding each query as independent one will be disadvantageous to further adoptions. Therefore, we investigate a new problem called **Influence Maximization over Streaming Graphs (*SGIM*)** which aims to find the seed set that has the maximum influence when considering previous query sequence. One of the major challenges of the problem formulation is its inborn uncertainty. Due to complex relationships among nodes, the possibility with which a node influence another node cannot be deduced from the uncertain graph and whether an individual is truly influenced cannot be observed. Thus, it is infeasible to directly remove the influenced individuals from selection candidates. To address this issue, we use sketch-based methods to transform an uncertain graph to many deterministic sketches which can be viewed as many possible worlds. Then, the results in each possible world are aggregated to estimate the expectation of influenced users that have not been influenced before. Additionally, users who have been activated will experience "*refractory period*" and they can recover from this state and become capable of being influenced again after some time. Consequently, we adopt the *sliding window model* [13] to manage the nodes in the *"refractory period"*.

To solve the *SGIM* problem, the difficulty lies in the dynamic features of the problem. In real world, social networks are highly dynamic, evolving rapidly. When network evolves, the obtained answer soon becomes outdated. In the meantime, queries are raised successively and form a query sequence. These two evolving factors both raise the question of how to efficiently identify the users with maximum expectation of newly influenced users at any time. We find that directly extending the existing algorithms in static scenarios to this new problem will incur inefficiency in terms of time and space. And the cause is redundant computation among different sketches and queries. For each sketch, the search algorithm will be performed to acquire the average of node influence. We construct Influence-Increment-Index to avoid running from scratch when answering queries along with two sketch-centralized indices called Influence-Index and Reverse-Influence-Index to aid in the maintenance of Influence-Increment-Index. We also design update algorithms and its optimized version to prune unnecessary operations during index maintenance. Besides, we maintain the results of query sequence with sliding window model. Two extra indices are designed to facilitate the update process. With the aid of these components, *SGIM* problem can be solved expeditiously in a steaming manner.

In summary, our main contributions are as follows:

- Motivated by practical factors, we formalize a new problem called Influence Maximization over Streaming Graphs (*SGIM*) which takes the influence of the most recent θ queries into consideration.
- We construct Influence-Increment-Index along with Influence-Index and Reverse-Influence-Index to save the intermediate results and devise corresponding update algorithms. We introduce prune technique to speed up the update process.
- We maintain the results of evolving query sequence with sliding window and design two extra indices to expedite its update.
- We experimentally demonstrate the efficiency and effectiveness of our method on several real-world graphs.

2 Related Work

2.1 Influence Maximization in Static Networks

The last decades have witnessed the booming of influence maximization approaches which can be classified into three categories: the simulation-based algorithms [5, 14–16], the heuristic-based algorithms [17, 18] and the sketch-based algorithms [19–24].

The simulation-based methods repeatedly simulate the diffusion process to obtain an approximation of *influence spread*. Some optimization approaches have been proposed to alleviate the pain of expensive computation caused by *Monte Carlo (MC) simulation* [5]. They either decrease the number of simulations [14, 15] or reduce the complexity of *MC simulation* [16].

The heuristic-based methods estimate the influence of a node according to some metrics (such as using degree [17]

or using local arborescence structures of each node [18]) rather than running heavy *MC simulations*. These methods are more scalable and efficient than the simulation-based methods, but they often lack theoretical guarantee, thus generating poor quality seeds in some datasets.

The sketch-based methods provide efficient solutions with a good guarantee. They rely on sampling-based exploration and estimate the influence with the aid of generated sketches, such as forward influence sketch [19–21] and reverse reachable sketch [22–24]. Rather than repeatedly running simulations, they generate sketches that capture the diffusion process through simulations. As the names suggest, the forward influence sketch method generates sketches by conducting forward simulations while reverse reachable sketch method builds sketches by performing reverse simulations.

These methods are proposed in static scenarios and directly applying them to dynamic scenarios is computationally expensive.

Recently, there is another type of work called adaptive IM [25–29], which has attracted many researchers' attention. These works assume that the feedback in the real-world is available. In this way, k seeds can be selected in batches rather than at once since researchers could use the feedback to choose more high-quality seeds in the following selections. Han et. al. [25] extends the reverse reachable sketch work to select the most influential nodes in a batch and then rule out the influenced nodes with the observation of real-world data. In the next batch, the selection is based on the new graph that the influenced nodes have been removed. This process continues until all seeds are selected. This line of work emphasizes that different batches of selections are not independent with previous ones. However, the actual feedback sometimes cannot be observed in real life. Without the observation of feedback, it will be hard to extend these works to dynamic scenarios.

2.2 Influence Maximization in Dynamic Networks

There are several recent studies of *IM* problem in dynamic networks [8–10, 30–32]. The majority of the literature focus on solving the problem extended directly from static networks which aim to acquire seeds efficiently when the networks change. Song et al. [8] devised an Upper Bound Interchange Greedy (UBI) approach which started from previously found seed set with node replacement instead of constructing the set from scratch. Ohsaka et al. [9] and Yang et al. [10] extended the sketch-based method [22]. They maintained a sample of random RR sets and devised incremental algorithms for updating the sets when networks change. The problem they tried to solve is a special case of our proposed problem.

There are also some variants of *IM* in dynamic networks considering specific conditions. Zhao et al. [11] is concerned about data recency which means that older user interactions are less significant than more recent ones when evaluating influence. Thus, they proposed a general time-decaying dynamic interaction network (TDN) model to smoothly discard outdated data and designed three efficient algorithms based on this model. Huang et al. [12] modeled evolving network as a sequence of snapshots and proposed a new problem called DIM which aims to find a fixed seed set of k target users to maximize the *influence spread* over distinct users in an evolving social network. They integrated all the snapshots for one selection and devised two different strategies (HCS and VCS) to solve the DIM problem. They think influencing a large number of different people is more important. This motivates us to pay attention to the value of user distinction, but instead of focusing on one query, we are more concerned about the query sequence.

Table 1 Notations

Notation	Description
SG_i	The i-th sketch of graph G
$\mathcal{IS}_{SG_i}(S)$	The influenced node set of S in the i-th sketch
$\sigma(S)$	The *influence spread* of node set S
BS_i	The blocked nodes maintained in the sliding window in the i-th sketch
$\mathcal{INC}_G(S)$	The *influence spread increment* of node set S in graph G

3 Preliminaries and Problem Formulation

In this section, we extends classical *IM* problem in static scenarios to dynamic ones. Several notations are listed in Table 1.

3.1 Preliminaries

A static graph is composed of fixed nodes and edges. While in dynamic scenarios, a graph is always evolving, which can be defined as a sequence of streaming edges.

Definition 1 (Streaming Graph) A streaming graph consists of an infinite stream of edges that arrive chronologically. Each edge in the stream is a 5-tuple $(u, v, +, p, t)$ where u is the source node, v is the destination node, $+/-$ is the notation which indicates whether the edge is inserted or removed, p is the probability with which u can influence v, and t is the timestamp when the edge arrives.

In this paper, we only consider the edge insertions. But edge deletions can also be integrated into the solution in a similar way. Edge insertions and deletions are both frequent actions in the social networks. For example, the "retweet"

actions on Twitter, the "reply" actions on Stackoverflow, and the "comment" actions on Facebook are all common activities in the real world. These actions insert edges when they are created, and the edge is deleted when that action is canceled.

Definition 2 (Snapshot) A snapshot at a specific timestamp t is a graph that consists of nodes and edges appearing before timestamp t.

To capture the state of streaming graph at timestamp t, a snapshot can be constructed. For each snapshot, influence of nodes and node sets can be evaluated as *influence spread*. Our problem is based on a widely adopted information diffusion model called Independent Cascading model (*IC model*) [5]. In this model, each edge $e = (u, v)$ has a weight p_{uv} representing the probability with which u can activate v. Given an initial activated set (*seeds*), the independent cascading process unfolds as follows. The active nodes will activate their dormant neighbors with corresponding probabilities. This process is iterated recursively until no more activation happens.

The *influence spread* of *seeds* S, denoted by $\sigma(S)$, is the expected number of activated nodes after the cascading process. Since $\sigma(S)$ cannot be derived analytically sometimes, simulation-based methods [33] can be applied to simulate the activation process for many times. Due to the heavy computational cost of simulation process, sketch-based methods [33] are proposed to approximate the expectation, which have already been exploited in many previous researches [19, 20]. One of the sketch-based methods is called forward influence sketch approach. This approach transforms a probabilistic graph to scores of deterministic sketches with coin flip technique and performs reachability test on each of these sketches to estimate *influence spread*.

Definition 3 (Sketch) A sketch is an instance induced by the diffusion process. Given a graph (snapshot) G, a sketch (denoted by SG_i) is constructed by removing each edge $e = (u, v)$ with probability $1 - p_{uv}$ from G.

Definition 4 (Influence Set) The **I**nfluence **S**et of a seed set S in a sketch SG_i (denoted by $\mathcal{IS}_{SG_i}(S)$) is the set of nodes reachable from S in sketch SG_i.

Given R sketches $SG_1, SG_2, \ldots, SG_R$ and a set S, $\sigma(S)$ is estimated by averaging the number of nodes that are reachable from S on different sketches.

$$\sigma(S) = \frac{\sum_{i=1}^{R} \left|\mathcal{IS}_{SG_i}(S)\right|}{R} \tag{1}$$

The influence maximization problem (*IM*) is to find a seed set S^* of k nodes that satisfies:

$$S^* = argmax_{S \subseteq V \text{ and } |S|=k} \sigma(S) \tag{2}$$

Although Kempe et al. [5] proved that this problem is NP-hard, owing to the property of monotonity and submodularity of function $\sigma(S)$, a greedy hill-climbing strategy (Algorithm 1) can achieve an approximation ratio of $(1 - 1/e)$ to the optimum solution. This greedy strategy iteratively adds the node v with a largest marginal gain $(\sigma(S \cup \{v\} - \sigma(S))$ until k nodes have been selected.

Algorithm 1: greedy hill-climbing

```
S = {};
for i = 1 to k do
    v_i = argmax_{v∈V\S} σ(S ∪ {v}) − σ(S);
    S = S ∪ {v_i};
end
return S;
```

3.2 Influence Maximization over Streaming Graphs (SGIM)

3.2.1 Problem Formulation

Due to the observation that graph evolves and large-overlapping selections will reduce the effectiveness, we propose a new problem. We focus on dynamic graph rather than a static one. Besides, we take into account mutual interaction of different queries. As shown in Fig. 1, the graph evolves with the arrival of new edges. When a query is raised, the corresponding snapshot at that time can be obtained to capture the latest state of streaming graph. There are two points about mutual interaction when answering a query. On one hand, the results of previous queries will definitely have a strong impact on the effectiveness of current selection. On the other hand, the effect of past selections is not permanent. A user who has been influenced in the past will still have the chance to be influenced again after some time. Therefore, the query is then answered by considering both the current snapshot and certain number of previous selections. Specifically, we adopt the *sliding window model* [13] and utilize the *active window* to control the range of the effect. Given an infinite stream of queries, let current query be Q_t, the influence results of query $Q_{t-\theta}$, $Q_{t-\theta+1}$, ..., Q_t are in the *active window* of sliding window where θ is the window length. And the length of *active window* means how many number of rounds that needs to be considered. By focusing on the freshest data in the *active window* and ignoring the oldest data, nodes that have been recently influenced can be ruled out and blocked nodes can be activated again after

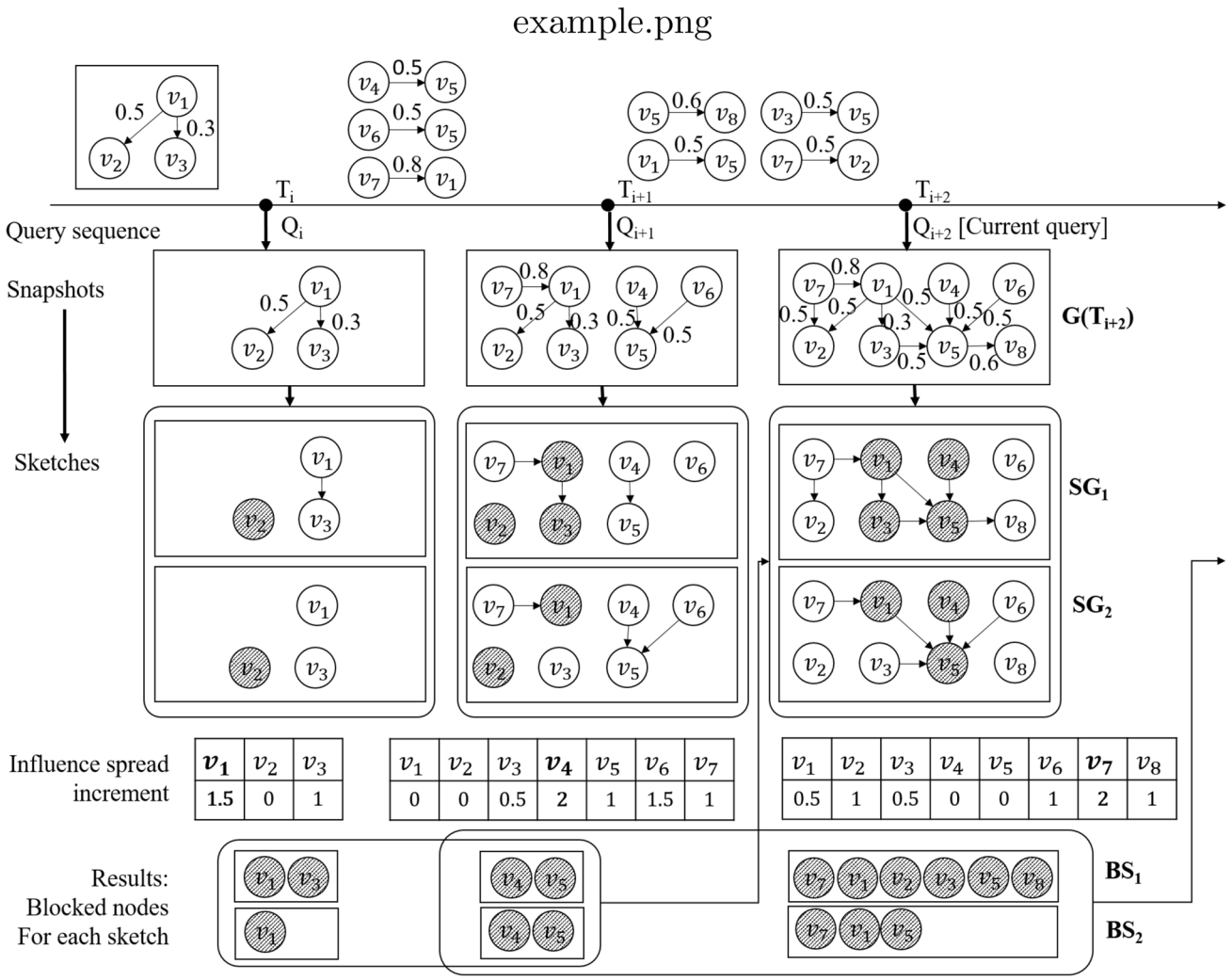


Fig. 1 SGIM model (θ=2, number of sketches=2). Current query Q_{i+2} takes into the account of the results of Q_i and Q_{i+1} when obtaining the *influence spread increment* for each node. After the query process completes, new nodes are blocked (shaded circle). Therefore, the window slides and the blocked nodes in the *active window* change correspondingly

some time. Each data element in the *active window* records the influence of that round of selection and expires exactly after θ rounds of queries, which means the influenced/activated nodes in the *active window* are **blocked** for following θ rounds and they can be influenced again after the release.

We generate several sketches for a snapshot to estimate the expectation of newly activated nodes. When a query is raised, those nodes that have been influenced in previous θ rounds of queries are correspondingly ruled out in each sketch. We denote the blocked nodes in the *active window* in the i-th generated sketch (SG_i) as Blocked Set BS_i. And then the expectation of newly activated nodes can be estimated by *influence spread increment*.

Definition 5 (Influence Spread Increment) Given R sketches SG_1, SG_2, ..., SG_R of snapshot G, the result of previous θ rounds of queries $BS_1, BS_2, \ldots, BS_R$, and a set S, influence spread **INC**rement of S equals the average of the number of the nodes which are activated in the most recent round of selection in each sketch. Therefore, *influence spread increment* of snapshot G is defined as

$$\mathcal{INC}_G(S) = \frac{\sum_{i=1}^{R} \left| \mathcal{IS}_{SG_i}(S) \backslash BS_i \right|}{R} \tag{3}$$

For example, in Fig. 1, when query Q_{i+2} is raised, nodes v_1, v_3, v_4, and v_5 are blocked in the first sketch, and node v_1, v_4, and v_5 are blocked in the second sketch. Therefore, node v_7 can newly influence node v_7, v_2, and v_8 in the first sketch while influencing node v_7 in the second sketch. Consequently, the *influence spread increment* of node v_7 equals *2*. Since our goal is to find the maximum *influence spread*

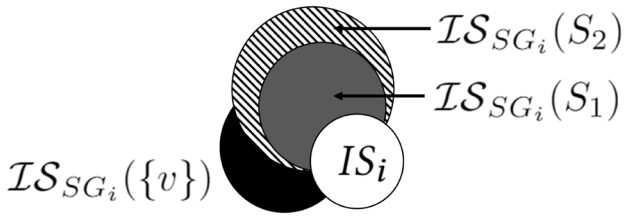


Fig. 2 Submodularity. The black circle denotes the influence set of node v in sketch SG_i. The white circle denotes the blocked nodes in sketch SG_i. The grey and shaded circle correspondingly denotes the influence set of node set S_1 and S_2 in sketch SG_i

increment instead of *influence spread*, we formulate *SGIM* problem as follows.

***SGIM* problem** *Given positive integer k, SGIM problem is to find the seed set S^* of k nodes in the latest snapshot of streaming graph G when a query happens that satisfies:*

$$S^* = argmax_{S\subseteq V\wedge|S|=k}\mathcal{INC}_G(S) \tag{4}$$

Since the classical *IM* problem is NP-hard and is a special case of *SGIM* problem when setting θ to 0, *SGIM* problem is also NP-hard.

3.2.2 Property

Now, we prove that the objective function $\mathcal{INC}_G(S)$ is monotonic and submodular under the *IC model*.

Proof 1 (Monotonicity) For each sketch, BS_i is fixed and adding a node v to set S guarantees $\mathcal{IS}_{SG_i}(S) \subseteq \mathcal{IS}_{SG_i}(S\cup\{v\})$, so $|\mathcal{IS}_{SG_i}(S)\backslash BS_i|$ will not decrease. Thus, $\mathcal{INC}_G(S\cup\{v\}) \geq \mathcal{INC}_G(S)$ is always satisfied. □

Proof 2 (Submodularity) A function F is submodular if $F(S_1\cup\{v\}) - F(S_1) \geq F(S_2\cup\{v\}) - F(S_2)$ holds for $\forall\, v$, $\forall\, (S_1, S_2)$ where $S_1 \subseteq S_2$.

Recap that $\mathcal{INC}_G(S) = \frac{\sum_{i=1}^{R}\left|\mathcal{IS}_{SG_i}(S)\backslash BS_i\right|}{R}$, we only need to prove that for each i, function $\left|\mathcal{IS}_{SG_i}(S)\backslash BS_i\right|$ is submodular, as the non-negative linear combination of submodular function is also submodular. Given SG_i and BS_i, let

$$H(S) = \left|\mathcal{IS}_{SG_i}(S\cup\{v\})\backslash BS_i\right| - \left|\mathcal{IS}_{SG_i}(S)\backslash BS_i\right|,$$

we only need to prove that for any $S_1 \subseteq S_2$, $H(S_1) \geq H(S_2)$. Instead of providing a theoretical analysis, we turn to a more intuitive way to show the property of submodularity (Fig. 2). As the equation illustrates, $H(S)$ equals the number of elements that are in $\mathcal{IS}_{SG_i}(\{v\})$ while not in $\mathcal{IS}_{G_i}(S)$ or BS_i ($|\mathcal{IS}_{SG_i}(\{v\})\backslash(\mathcal{IS}_{G_i}(S)\cup BS_i)|$). Therefore, $H(S_1)$ equals the number of elements in black area which is uncovered by grey circle and white circle, and $H(S_2)$ equals to the number of elements in black area which is uncovered by shaded circle and white circle. Since $S_1 \subseteq S_2$, $\mathcal{IS}_{SG_i}(S_1) \subseteq \mathcal{IS}_{SG_i}(S_2)$, which means that the area of the shaded circle is larger or equal to the grey circle. Conclusion can be drawn according to the uncovered area that $H(S_1) \geq H(S_2)$. □

3.2.3 Extension of IM Solutions to Solve SGIM Problem

Since our objective function satisfies monotonicity and submodularity at the same time, we can use greedy algorithm to approximate the optimum to within a factor of $(1-1/e)$(where e is the base of natural logarithm) [5]. The algorithm goes as follows. When a new query arrives, the blocked nodes can be ruled out in each sketch based on the memory of the most recent results. Then, the *influence spread increment* of a node v can be obtained by averaging the number of nodes that are reachable from node v while not in the blocked set in each sketch. The node with the maximum marginal gain is selected successively until the query finishes.

However, this approach may require large storage for sketches and induce redundant computation among queries. We design intermediate results to facilitate the query process with the aid of Influence-Increment-Index along with sketch-centralized Influence-Index and Reverse-Influence-Index.

4 Influence-Increment-Index

When a query is raised, running from scratch will incur a large amount of redundant computations. We analyze those inessential operations and design Influence-Increment-Index to facilitate generating answers efficiently. Besides, we design two types of indices to assist in the update of Influence-Increment-Index. Since the reliance on the sketch-separated principle is one of the major reasons of inefficiency, we carefully design them in a sketch-centralized manner.

4.1 Structure

The core to answer a query is obtaining the *influence spread increment* of each node. To obtain the value, two phases are typically required. The first phase is to get the influence set of the node. The second phase is to count the number of influenced nodes in each sketch that are not blocked. Specifically, for each node v, given blocked nodes $\mathcal{BI}$, its *influence spread increment* can be calculated by Algorithm 2[1,2].

[1] *count* operation of a bitset returns the number of *ones* in the bitset.

[2] *flip* operation of a bitset converts *zeros* into *ones* and *ones* into *zeros*.

Algorithm 2: Influence Spread Increment

```
Data: v, BI
increment = 0;
T = get influence set of v in each sketch by performing a search;
for (v_i, b_i) in T do
    if v_i not in BI then
        increment += b_i.count[1];
    end
    else
        increment += (b_i & BI[v_i].flip[2]).count;
    end
end
```

Since the influence set of nodes and blocked nodes change gradually in the dynamic scenarios, *influence spread increment* is also evolving correspondingly. Consequently, we can incrementally update *influence spread increment* rather than running from scratch. We maintain the value of *influence spread increment* in Influence-In**C**rement-**I**ndex ($\mathcal{CI}$) so that it can be directly used to efficiently generate seed set as the answer. Each node v can obtain its *influence spread increment* by getting the value of $\mathcal{CI}[v]$.

During the graph evolution, if the subgraph that contains a certain node has not changed, then the node's reachability would not change, either. In this case, there is no need to compute from scratch. On other occasions where its reachability has changed, in order to get the latest reachability, one way is to recompute. This will guarantee the accuracy of reachability, but it neglects the unchanged calculation that can be utilized to expedite the process of retrieving the influence set, resulting in computational overhead. So maintaining the intermediate results of the influence set would avoid repeating the same process when answering different queries. Furthermore, during the query process, the maintained results can also assist in answering *SGIM* queries since less update operations are needed when a node is selected. Therefore, we maintain the influence set of each node in Influence-Index. Motivated by work [12], we also use bitset to represent the reachability in different sketches.

Influence-**I**ndex of a node v (denoted by $v.\mathcal{II}$) records its influence in different sketches. $v.\mathcal{II}$ is composed of a series of tuples $\{(v_0, b_0^R), (v_1, b_1^R), \ldots, (v_i, b_i^R), \ldots\}$ which contains a node id v_i and a corresponding bitset b_i^R, while R stands for the number of generated sketches. Tuple (v_i, b_i^R) in $v.\mathcal{II}$ means *node v can reach* v_i in at least one of the R sketches, and b_i^R, which aggregates the reachability between these two nodes in R sketches, can be viewed as the weight of a virtual edge between v and v_i. If the i-th number in b_i^R equals *one*, it means node v can influence node v_i in the i-th sketch.

Equipped with Influence-Indices, we can compute the latest *influence spared increment* for any node v when its influence set or the blocked nodes have changed. However, the maintenance of Influence-Indices itself can also be time-consuming if only the information of influence is stored. Therefore, we also maintain the information of reverse influence in Reverse-Influence-Indices to facilitate the update of Influence-Index when an edge is inserted.

Reverse-**I**nfluence-Index of a node v (denoted by $v.\mathcal{RI}$) records its reverse influence in different sketches. $v.\mathcal{RI}$, which has a similar structure with $v.\mathcal{II}$, is composed of a series of tuples. Each tuple (v_i, b_i^R) in $v.\mathcal{RI}$ means *node* v_i *can reach* v in at least one of the R sketches, and b_i^R aggregates the reachability between these two nodes in R sketches. If the i-th number in b_i^R equals *one*, it means node v_i can influence node v in the i-th sketch.

4.2 Update Indices

When an edge is inserted, a part of nodes' reachable nodes will change, raising the requirement of updating maintained indices. Specifically speaking, when edge $(u, v, +, p, t)$ is inserted to the graph, the Influence-Index of u and its ancestors in the graph along with the Reverse-Influence-Index of node u and its children will change. Consequently, the *influence spread increment* of node u and its ancestors become different. As shown in Fig. 3, when edge $(v_3, v_5, +, 0.5, t)$ is inserted, Influence-Index of node v_3 and its ancestors v_1 and v_7 are different after the edge insertion. Reverse-Influence of node v_3 and its child v_8 also changes. Naturally, Influence-Increment-Index changes along with Influence-Index. We can use a reverse search from node u to visit the ancestors and update their maintained indices. Since the way reachable nodes transmits along the path matches Depth-First Search (*DFS*)'s nature of going in depth, *DFS* is more suitable than Breadth-First Search (*BFS*) in this scenario. Therefore, the algorithm of update indices goes as follows. When edge $(u, v, +, p, t)$ arrives, bitset uv is generated according to probability p where each bit represents whether the edge (u, v) is inserted to each sketch. After inserting the generated bitset to compressed graph,[3] the indices are updated during the reverse *DFS* process. A naive update algorithm is first introduced and then an improved version with pruning technique is illustrated afterward.

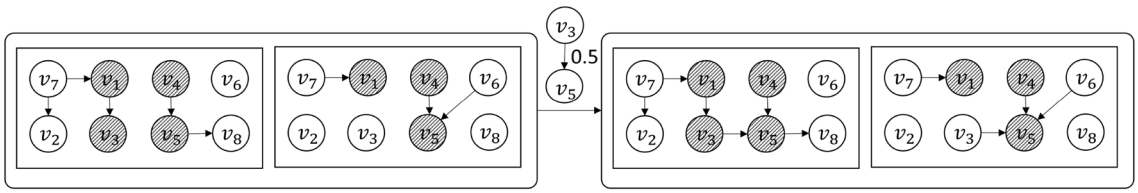
Algorithm 3: naive-UpdateOneNode

```
Data: u, uvEdge, updateIndex
for (v_t, b_t) in updateIndex do
    old_value = u.II[v_t];
    new_value = u.II[v_t] | (uvEdge & b_t);
    if old_value ∧ new_value ≠ 0 then
        u.II[v_t] = new_value;
        v_t.RI[u] = new_value;
        CI[u] += (new_value & QInf[v_t].flip).count-(old_value & QInf[v_t].flip).count;
    end
end
```

4.2.1 Naive Algorithm for Intermediate Results Update

The reachability between nodes in a compressed graph is represented by a bitset rather than a bool value. If the

[3] We adopt the same graph structure as [12] in which the edge in the graph is a bitset.



Before the edge insertion		After the edge insertion
Influence-Index		
v_3	(v_3, [11])	+ (v_5, [11]), (v_8, [10])
v_1	(v_1, [11]), (v_3, [10])	+ (v_5, [10]), (v_8, [10])
v_7	(v_1, [11]), (v_2, [10]), (v_3, [10]), (v_7, [11])	+ (v_5, [10]), (v_8, [10])
Reverse-Influence-Index		
v_5	(v_5, [11])	+ (v_3, [11]), (v_1, [10]), (v_7, [10])
v_8	(v_4, [11]), (v_5, [10]), (v_8, [11])	+ (v_3, [10]), (v_1, [10]), (v_7, [10])
Influence-Increment-Index		
v_3	0.5	1
v_1	0	0.5
v_7	1.5	2

Fig. 3 Influence-Index, Reverse-Influence-Index, and Influence-Increment-Index change when an edge is inserted

reachability from u to v in R sketches is b_{uv} and from v to t is b_{vt}, then it equals $b_{uv}\&b_{vt}$ from u to t. This property is used when updating the indices. In Algorithm 3, v_t denotes the nodes that could be reached from v. Given a bitset b_t which contains the reachability between v and v_t, the latest reachability R_{uv_t} between u and v_t can be calculated as shown in 3th line. Subsequently, node u's Influence-Index ($u.\mathcal{II}$), node v_t's Reverse-Influence-Index ($v_t.\mathcal{RI}$) and u's *influence spread increment* ($\mathcal{CI}[u]$) are updated with R_{uv_t}.

Algorithm 4: naive-UpdateNodes

```
   Data: u, uvEdge, updateIndex
1  if (visited[u].flip & uvEdge) == 0 then
2  |  return;
3  end
4  visited[u] |= uvEdge;
5  naive-UpdateOneNode(u, uvEdge, updateIndex);
6  for v_a in u's ingoing neighbors do
7  |  e_{v_a u} = get edge from v_a to u(a bitset);
8  |  naive-UpdateNodes(v_a, uvEdge&e_{v_a u}, updateIndex);
9  end
```

This operation of intermediate results update of one node is then applied to u and its ancestors during reverse *DFS* (Algorithm 4). We also use a bitset rather than a bool value to record the visit information of nodes, denoted as *visited[v]* in the compressed graph.

4.2.2 Pruned Algorithm for Intermediate Results Update

There are certain conditions when naive update algorithm will cause inefficiency. If node u can already reach v before the edge insertion, the intermediate results of node u and its ancestors do not need to be updated, because the insertion of this edge will not affect their reachability. As shown in Fig. 4, since the reachability from node v_1 to node v_5 has not changed, there is no need to update the indices of node v_1 and v_7.

Based on this observation, we propose pruned algorithm to filter out the invalid update operations in the naive algorithm. Specifically, the pruning rule is described as follows. In each sketch, if node v_t can reach node v before this edge insertion, then the intermediate results of v_t and its ancestors do not need to be updated, which is implemented by skipping node v_t of performing the reverse *DFS* process. This rule will not reduce the accuracy.

Having been armed with the pruning rule, an improved algorithm on compressed graph is illustrated in Algorithm 5 and 6. Rather than using the same piece of information ($v.\mathcal{II}$) to update all the ancestors in Algorithm 3, the pruned

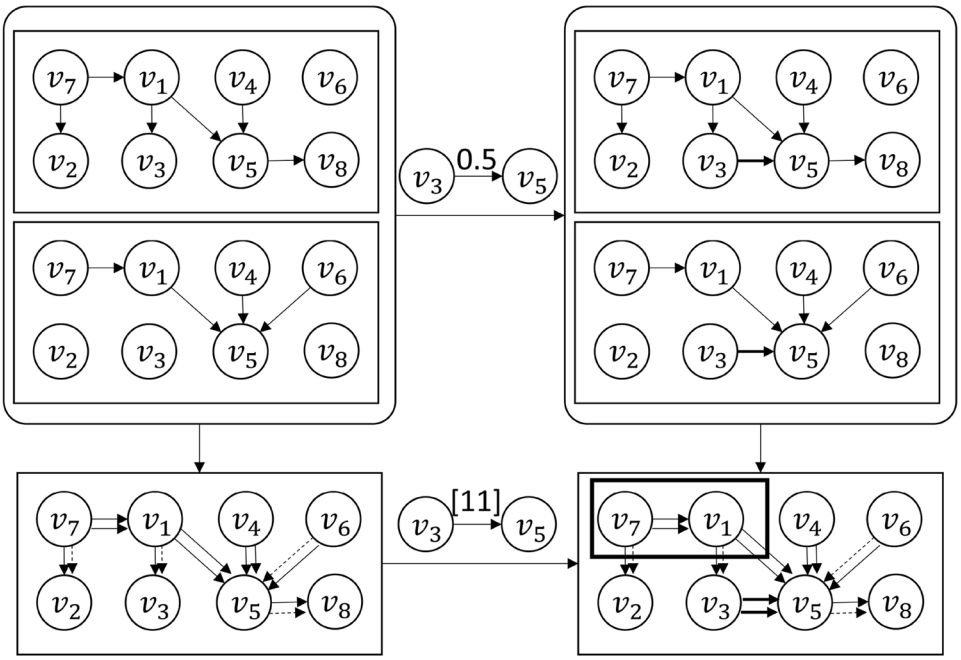


Fig. 4 When a new edge $(v_3, v_5, +, 0.5, t)$ arrives, v_1 can reach v_5 before this insertion, so the reverse *DFS* from v_1 can be halted

update algorithm uses different information for different ancestors. For each ancestor v_a, only necessary information that will influence v_a's reachability will be used. During the process of reverse *DFS*, the size of updateIndex will continue to decrease by screening off the unchanged nodes. Besides, when the updateIndex becomes empty for a node v_t, the algorithm stops updating the reachability of v_t's ancestors. Since the reachability of node v_t has not changed, its ancestor's reachability will not change, either. In a nutshell, the pruning technique can both prune the number of times that an ancestor is visited and the number of times that the node in node v's reachability set is visited while warranting the accuracy.

Algorithm 5: pruned-UpdateOneNode

```
   Data: u, v, updateIndex
 1 uvEdge=get edge from u to v(a bitset)
 2 for (v_t, b_t) in updateIndex do
 3 |   old_value = u.II[v_t];
 4 |   new_value = u.II[v_t] | (uvEdge & b_t);
 5 |   if old_value ∧ new_value ≠ 0 then
 6 |   |   u.II[v_t] = new_value;
 7 |   |   v_t.RI[u] = new_value;
 8 |   |   CI[u] += (new_value &
 9 |   |   QInf[v_t].flip).count-(old_value & QInf[v_t].flip).count;
10 |   |   updateIndex[v_t] = old_value ∧ new_value;
11 |   end
12 |   else
13 |   |   remove tuple (v_t, b_t) from updateIndex;
14 |   end
15 end
```

Algorithm 6: pruned-UpdateNodes

```
   Data: u, uvEdge, updateIndex
 1 if (visited[u].flip & uvEdge) == 0 then
 2 |   return;
 3 end
 4 visited[u] |= uvEdge;
 5 pruned-UpdateOneNode(u, v, updateIndex);
 6 if updateIndex is not empty then
 7 |   for v_a in u's ingoing neighbors do
 8 |   |   e_{v_a u} = get edge from v_a to u(a bitset);
 9 |   |   pruned-UpdateNodes(v_a, uvEdge&e_{v_a u}, updateIndex);
10 |   end
11 end
```

5 Solution

SGIM problem needs to consider the result of last θ rounds of queries. In this section, we will first introduce how to maintain the results of query sequence and then how to answer *SGIM* queries.

5.1 Maintaining the Results of Query Sequence

We use sliding window model to manage the results of queries. Its *active window* (denoted as Blocked Window) records the results of last θ rounds of queries and current query. Along with Blocked Window, we also save two extra indices to facilitate the query process. These three structures as shown in Fig. 5 store the results of previous query sequence in a more compact way.

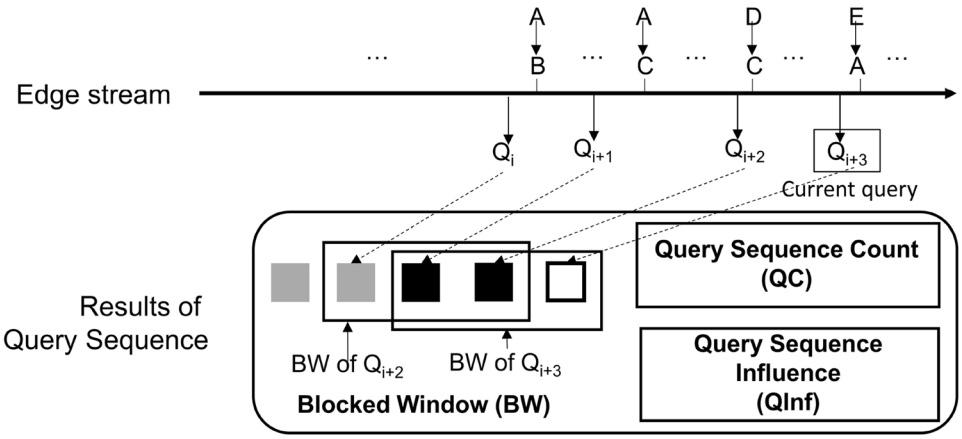


Fig. 5 Sketch-centralized results of query sequence ($\theta = 2$)

5.1.1 Blocked Window (BW)

Blocked Window is the *active window* of sliding window model. It contains $\theta + 1$ data elements (denoted as BI_θ, $BI_{\theta-1}$, ..., BI_0), and each one chronologically represents the result of each query from left to right, while the rightest blank in Fig. 5 corresponds to the result of current query, which is initially empty. To be more specific, each data element is a tuple sequence and each tuple (v, b_i^R) means node v has been influenced in that round of query and thus is blocked for following θ rounds of selections. The bitset b_i^R is the detailed information in different sketch where value *one* means node has been influenced in that sketch.

When a new data element arrives, the oldest element is discarded and is no longer in the *active window*. For example, as illustrated in Fig. 5, Blocked Window of Q_{i+2} maintains the query result of Q_i, Q_{i+1}, and Q_{i+2}. When new query Q_{i+3} arrives, Blocked Window slides one step. Therefore, current Blocked Window of Q_{i+4} stores the query result of Q_{i+1}, Q_{i+2}, and Q_{i+3}. The sliding window model guarantees that the last θ rounds of query sequences will be considered and the blocked nodes will have the chance to be activated again after θ rounds.

5.1.2 Query Sequence Influence and Count (QInf and QC)

Blocked Window maintains the results of previous query sequence; however, it is not convenient to directly use it to answer a query since the blocked nodes in each data element need to be ruled out successively. This will make some common nodes in different sketches be processed more than once, resulting in extra computation. Therefore, we further maintain two aggregation indices—Query Sequence Influence (QInf) and Query Sequence Count (QC).

Query Sequence Influence (QInf) represents whether a node is influenced in each sketch during any of the selections (including the current one). In this way, the blocked nodes in different query is aggregated together and each blocked node will be merely processed once instead of several times when being ruled out. To be more specific, QInf is a tuple sequence and each tuple (v_i, b_i^R) means that node v_i has been influenced and if the bit in b_i^R equals *one*, then node v_i has been influenced in that particular sketch. Its relationship with Blocked Window can be written as Eq. 5. The *union* operation compresses all indices into one by merging all bitsets for each node $v \in \cup_{i=1}^{k} I_i.V$[4] with *OR* operations.

$$QInf = union(BI_\theta, BI_{\theta-1}, \ldots, BI_0) \tag{5}$$

Every time when the window slides, QInf is updated simultaneously. An naive way to fulfill this aim is to recalculate Eq. 5. However, part of the redundant computation can be spared since the information from $BI_{\theta-1}$ to BI_1 remains the same after one step. Based on this observation, Query Sequence Count (QC) is designed. It records the number of times of a node being influenced in each sketch across all past selections. Similarly, each tuple (v_i, b_i^R) in QC represents the detailed information in each sketch. It means that node v_i has been influenced and it has been influenced $b_i^R[i]$ times in the i-th sketch. It can be formulated as Eq. 6. The *sum* operation transforms bitsets for each node $v \in \cup_{i=1}^{k} I_i.V$ into integer lists which are then added together. This operation aggregates scattered information of the blocked nodes so that these nodes can be processed unitedly when solving the *SGIM* problem.

$$QC = sum(BI_\theta, BI_{\theta-1}, \ldots, BI_1) \tag{6}$$

We adopt lazy update strategy, resulting in the difference that QInf integrates BI_0, while QC does not. Since the query process is dependent on QInf, whenever a node is picked as *seed*, its influence set needs to be updated to QInf immediately for subsequent selections. However, QC serves to

[4] $I.V$ denotes the set of all node ids in the index I.

simplify computation during window slide movement exclusively, so it can be updated after the selection of k seeds completes when the slide movement is ready and necessary to be performed.

5.1.3 Update

During the query process, BI_0 changes due to the selection of *seed*. After the query process completes, the sliding window slides one step. The outdated data are moved out, while new data element is moved in. Two corresponding algorithms (*moveIn* and *moveOut*) are illustrated in Algorithm 7 and 8[5].

Algorithm 7: moveIn

```
Data: data
union(BI_0, data);
union(QInf, data);
```

Algorithm 8: moveOut

```
change = an empty index;
for [id, count]: BI_θ do
    old_inf = QInf[id];
    for i = 1 to R do
        QC[id][i] += BI_0[id][i] − count[i];
    end
    QInf[id] = toBitset[5](QC[id]);
    if (old_inf ∧ QInf[id]).count != 0 then
        change[id] = old_inf ∧ QInf[id];
    end
    if QInf[id].count == 0 then
        remove id from QC and QInf;
    end
end
BW.remove(QS[0]);
BW.insert(empty data element);
return change;
```

moveIn(data) moves current blocked node into BI_0 and updates QInf by performing *union* operations. This operation integrates the information of the blocked nodes in current selection. According to our problem setting, the size of sliding window is fixed. Therefore, when a new data element arrives, the Blocked Window slides one step. *moveOut* operation moves out BI_θ and moves in an empty data element. It simultaneously updates QC and QInf. We bring in two operations called *addCount* and *minusCount* to update QC. They are supplementary to *sum* operation mentioned above. They aim to precisely modify the output of *sum* by integrating or disintegrating an index. Along with these two operations, Eq. 6 can be transformed to Eq. 7. In this way, instead of calculating the aggregation of Blocked Window from scratch, only incremental computation is needed.

$$QC = minusCount(addCount(QC, BI_0), BI_\theta) \tag{7}$$

QInf is updated by transforming each integer list in QC to a bitset. For each tuple in QInf, the *XOR* result of the bitset before and after the update is calculated and stored. This information is returned and will be further used to update intermediate result $\mathcal{CI}$.

5.2 Answering *SGIM* Queries

The solution (Algorithm 9) consists of five phases which will be introduced sequentially. Firstly, the node v_i with the maximum *influence spread increment* is added to the seed set. Secondly, the result of query sequence is updated by performing *moveIn* operation according to the selected seed. This step blocks the nodes that are influenced by the newly selected seed v_i, which guarantees the effectiveness of following selections. Thirdly, the intermediate result $\mathcal{CI}$ is updated due to the insertion of the blocked nodes. After these three phases are repeated until all k seeds are selected, *moveOut* operation is performed to discard outdated data. Lastly, the intermediate result $\mathcal{CI}$ is updated correspondingly due to the release of blocked nodes.

Algorithm 9: query

```
Data: k
S = {};
for i = 1 to k do
    v_i = argmax_{v∈V\S} CI[v];
    S = S ∪ {v_i};
    moveIn(v_i.II);
    updateCI(v_i.II, true);
end
changedData = moveOut();
updateCI(changedData, false);
return S;
```

The insertion and the release of blocked nodes both contribute to the change of the intermediate result $\mathcal{CI}$. We presume that v_i is a node which is removed due to the release of blocked nodes and b_i aggregates detailed information in different sketches. If i-th bit in b_i equals *one*, it means that node v_i was blocked before this removal and has been released after the movement in the i-th sketch. We suppose node v_j can reach node v_i in some sketches. Since node v_j can only influence node v_i in sketches where v_i is not blocked, the release of v_i in some sketches where it was blocked will enable v_j influence v_i in those sketches, leading to an increase in $\mathcal{CI}[v_j]$. A detailed illustration when sketch number equals 2 is presented in Fig. 6. The incremental contribution to the *influence spread increment* made by node v_2 to node v_7 in all sketches can be calculated as the number of *ones* in

[5] toBitset operation transforms a list to a bitset by converting *zero* to bit *zero* and other value to *one*.

the *AND* results. The update operation due to the insertion of blocked nodes goes similarly. The whole algorithm for update the intermediate result is presented in Algorithm 10. For each node that has been released in some of the sketches, its ancestors' values in $\mathcal{CI}$ are updated according to their reachability and condition of whether the update operation is due to a removal movement or an insertion one.

Algorithm 10: updateCI

```
   Data: data, flag
 1 for (v_i, b_i) in data do
 2     for (v_j, R_ji) in v_i.RI do
 3         if flag then
 4             CI[v_j] -= (b_i & R_ji).count;
 5         end
 6         else
 7             CI[v_j] += (b_i & R_ji).count;
 8         end
 9     end
10 end
```

5.3 Analysis

In this section, we analyze the time complexity of our proposed solution and also show the comparison with other methods. Additionally, we further provide the theoretical guarantee on the number of sketches.

5.3.1 Time Complexity Analysis

In algorithm 9, line 3 and line 4 take $O(1)$ time. The *moveIn* operation in line 5 can be regarded as two union operations of indices. Suppose that each bitwise operation takes $O(c)$ time and the number of nodes and edges in the current graph is denoted as n and m. Merging two indices can be implemented in $O(cn)$ time. The *updateCI* operation takes $O(cn^2)$ time to traverse the nodes and perform bitwise operations. The *moveOut* operation takes $O(nR)$ time for modifying the frequency value in each sketch. Therefore, the complexity for algorithm 9 is $O(ckn^2 + nR)$. The complexities of algorithms that are extended from previous researches to answer this query are as follows. The typical Monte Carlo simulation method (MC method) would run R rounds of simulations while excluding the already influenced nodes. The cost to compute the nodes that have been influenced/blocked in each round of previous queries is $O(\theta nR)$ for each node. The total complexity is $O(\theta Rkn^2 + mnRk)$. Since [20] provides several techniques to speed up the MC method but does not substantially improve the complexity, its complexity stays the same as $O(\theta Rkn^2 + mnRk)$. While using the same strategy with our method to maintain the query sequence in the extended version of [12], its complexity is $O(cmnRk + mR + nR)$. Owing to the strategy that we exploit to maintain the results of query sequence, the parameter θ is moved out from the time complexity equation. Furthermore, since real-life graphs usually have more edges than nodes, our method has better complexity than the method extended from [12]. As analyzed above, the bottleneck of our method is the update process. It will take a long time when the number of influenced nodes is large. However, it is unavoidable if we would like to keep track of the exact number of increments in each sketch. In the future, maybe heuristic methods with good guarantees can be proposed to seek an estimated value instead of an exact one.

5.3.2 Theoretical Guarantee on the Number of Sketches

We follow similar steps in [20] and theoretically analyze the error ratio of estimated influence spread increment. Firstly, we introduce Hoeffding's inequality.

Theorem 1 (Hoeffding's inequality) *Let $X_1, X_2, \ldots, X_n$ be independent random variables in* [0, 1]. *Let $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$. Then, $Pr[|\bar{X} - X| > t] \leq 2e^{-2nt^2}$.*

Lemma 1 *Let G be a graph and $\mathcal{S}$ be a family of node sets. Let $R = O(\frac{1}{\alpha^2} log \frac{2|\mathcal{S}|}{\delta})$. Then, with probability at least $1 - \delta$, $|\mathcal{INC}_G(S) - \mathcal{E}_G(S)| \leq \alpha n$ for every set $S \in \mathcal{S}$, where $\mathcal{INC}_G(S) = \frac{\sum_{i=1}^{R} \left|\mathcal{IS}_{SG_i}(S) \backslash BS_i\right|}{R}$ and $\mathcal{E}_G(S)$ denotes the expectation of the number of newly activated nodes.*

Proof 3 For brevity, we denote $\left|\mathcal{IS}_{SG_i}(S) \backslash BS_i\right|$ as $f_{SG_i}(S)$. Since each sketch is generated independently and $f_{SG_i}(S) \in [0, n]$, for any set $S \in \mathcal{S}$, by applying Hoeffding's inequality on $\frac{1}{n} f_{SG_1}(S)$, $\frac{1}{n} f_{SG_2}(S)$, ..., $\frac{1}{n} f_{SG_R}(S)$, we can acquire the relationship $Pr[|\mathcal{INC}_G(S) - \mathcal{E}_G(S)| > \alpha n] \leq 2e^{-2\alpha^2 R}$. Then, we apply the union bound over all sets in $\mathcal{S}$, the probability that the condition $|\mathcal{INC}_G(S) - \mathcal{E}_G(S)| \leq \alpha n$ is satisfied for every $S \in \mathcal{S}$ is at least $1 - 2|\mathcal{S}|e^{-2\alpha^2 R}$. We have the desired bound when we choose $R = O(\frac{1}{\alpha^2} log \frac{2|\mathcal{S}|}{\delta})$. □

Given a positive integer k, the number of node sets whose length equals k satisfies $|\mathcal{S}| \leq kn$. Thus, we have the above guarantee when we choose $R = O(\frac{1}{\alpha^2} log \frac{2kn}{\delta})$.

Lemma 2 *Let $\alpha = \frac{\epsilon OPT}{2nk}$, the greedy algorithm returns a $(1 - \frac{1}{e} - \epsilon)$-approximate solution with at least $1 - \delta$ probability.*

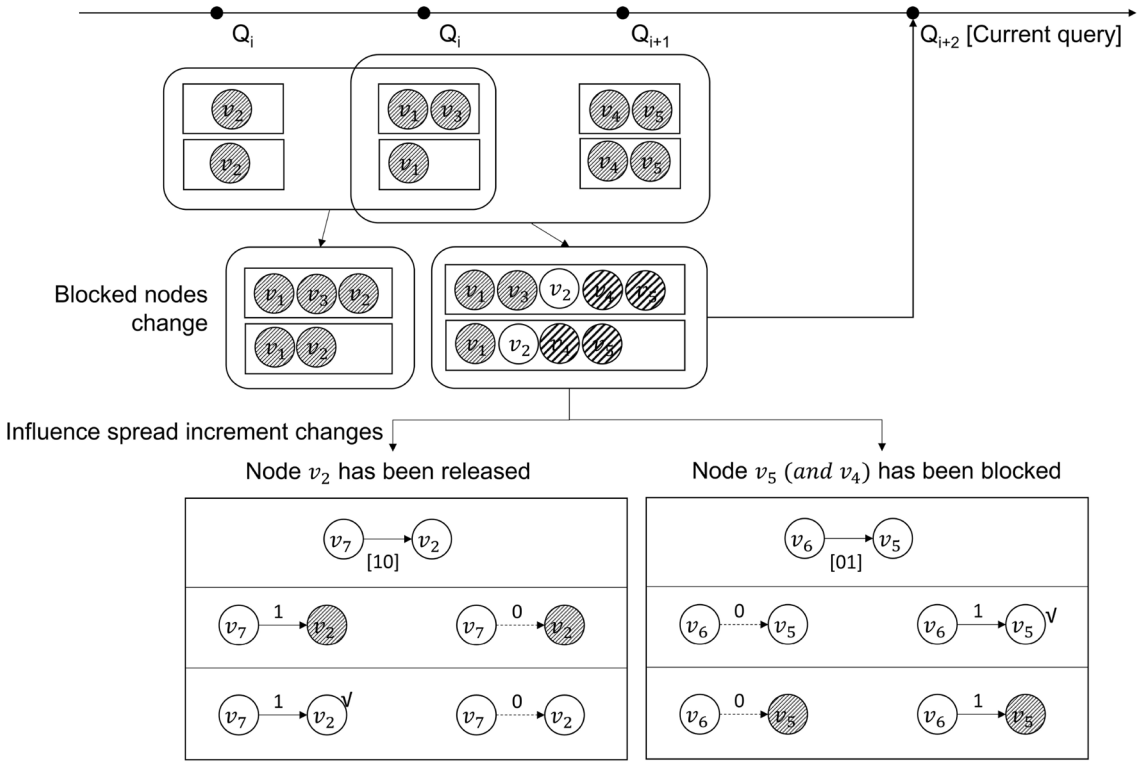


Fig. 6 Blocked nodes in *BW* change due to the change of data elements. Consequently, the *influence spread increment* of some nodes change correspondingly. For example, before the release operation, node v_2 is blocked in the first and second sketches (its $\mathcal{INC}$ equals *0*); after the release operation, it has been released in these two sketches (its $\mathcal{INC}$ equals *1*). The incremental equals $1 - 0 = 1$, which can also be calculated by ([10]&[11]).*count* = 1, where [10] is the reachability of edge (v_7, v_2) and [11] is the *XOR* result of the blocked bitset of node v_2 before and after the movement ([11] ∧ [00] = [11])

6 Experiments

In this section, we evaluate the effectiveness and efficiency of our proposed approach for *SGIM* problem on various real-world datasets.

6.1 Setup

6.1.1 Datasets

Our method is exhaustively tested on 5 real-world datasets. The summary of these datasets is listed in Table 2.

6.1.2 Environments

We conduct experiments on a Linux server with intel CPU (2.40GHz). All algorithms are implemented in C++ and compiled with -O3 option, and run in single thread.

6.1.3 Compared Approaches and Parameters

We compare our method with the following algorithms extended from state-of-the-art methods of solving *IM* problem in static graphs and a baseline algorithm. The number of generated sketches is set to 200 in all sketch-based methods.

- baseline: the naive algorithm that is directly extended from the sketch-based method for solving *IM* problem in static scenarios.
- *CSO*: due to the compact and streaming features, we denote our method as a *Compact Streaming Optimization* approach (abbreviated as *CSO*).
- VCS [12]: the algorithm extended from the VCS approach that uses compression techniques to manage graph.
- PMC [20]: the algorithm extended from PMC approach that exploits the existence of a hub to accelerate *BFS* for reachability tests.

6.1.4 Probabilistic Settings

We validate the performance of algorithms under a classical probabilistic model called *weighted cascade model* [5]. In this model , probability of each edge $e = (u, v)$ is set to $1/d^{-}(v)$. Since the graph is evolving, the in-degree of node

Table 2 Summary of datasets

dataset	# of nodes	# of edges
mathoverflow [34]	24,818	506,550
Twitter-Higgs[a] [35]	304,198	555,481
askubuntu [34]	159,316	964,437
superuser [34]	194,085	1,443,339
stackoverflow-c2q [34]	1,655,353	20,268,151

[a]The Higgs dataset consists of a mention network and a reply network denoted as higgs-MT and higgs-RE, respectively.

is changing, too. And the probability is generated based on the snapshot when the edge is inserted.

6.1.5 Experimental Method

For evaluating effectiveness and efficiency of different methods when solving *SGIM* problem, we set up experiment as follows to model the *SGIM* problem. Firstly, we generate a preprocessed file from the original dataset. This file consists of two types of lines: edge line and query line. An edge line is a tuple (u, v, p) where u is the source node, v is the destination node, and p is the probability with which u will influence v. A query line is composed of a character "Q" and a positive integer k, which means how many nodes are expected to be selected in this round.

We first order the temporal edges in each dataset by time and use the first *thr*% of edges as the base graph. Then, the remaining edges in the dataset are inserted chronologically to model the streaming process. For example, there are 90 edges as base graph and 10 edges as remaining graph, a random number r between (90, 100] is generated and the query is inserted exactly in the r-th line. In this experiment, $thr = 90$, $\theta = 3$, and 5 queries with $k = 10$ are inserted to each dataset.

In real life, graph is dynamic and queries can happen at any time. This can still be captured with the format of the preprocessed file. Then, given the preprocessed file, algorithms for experimental evaluation run as follows: when an edge arrives, the edge is inserted into the graph. And when a query is raised, the *SGIM* query is answered based on the current snapshot and results of previous θ rounds of queries.

6.2 Experimental Results for the *SGIM* Problem

We compare different methods on these datasets in terms of *Influence Spread Increment* and running time. We conduct experiments to measure the maintenance cost of index, and the memory cost of methods with and without compression. Besides, we compare the insertion time of our proposed approach with pruning technique and the one without pruning technique. The parameter θ can also be set to other values, and the effect of this parameter will be analyzed afterwards.

6.2.1 Influence Spread Increment

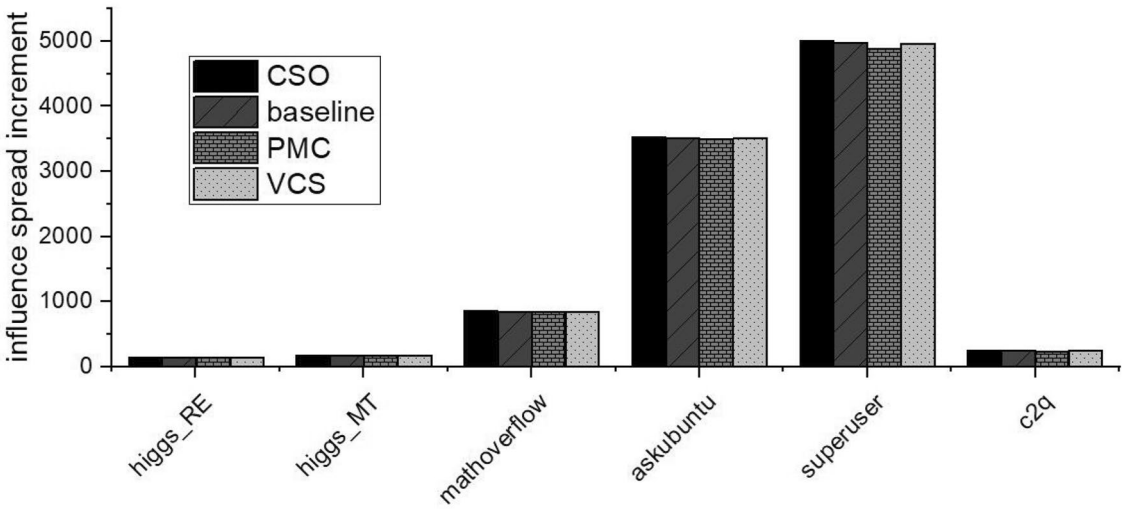
Figure 7a shows the *influence spread increments* that are calculated by different algorithms under *weighted cascade model*. *influence spread increment* of different algorithms are very close to each other. This means that seeds from these algorithms have close quality.

We also compare the *influence spread* and *influence spread increment* calculated by our proposed method. The result is shown in Fig. 7b. The value of *influence spread increment* is always smaller than *influence spread*, since those nodes influenced in previous selections are not included in the calculation. We further investigate the difference of these two concepts by a case study. In two consecutive rounds of queries when processing dataset higgs, the value of *influence spread* is 125 and 127, respectively, while the influence sets are highly overlapping with each other. However, the value of *influence spread increment* of these two same rounds of queries is 82 and 61, respectively, while the influence sets are different from each other. The result identifies that, following the greedy strategy of selecting nodes with maximum *influence spread increment*, the seeds can trigger larger cascade among different users than choosing nodes with maximum *influence spread*. Therefore, the advantage of *influence spread increment* is confirmed.

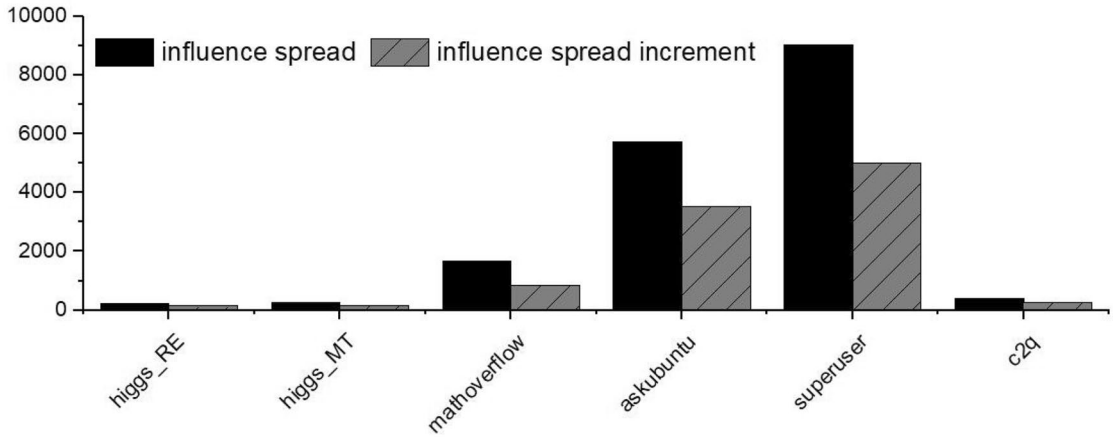
The largest dataset *stackoverflow-c2q* has a small *influence spread increment*. This is due to the scenario of the dataset. The dataset *stackOverflow-c2q* captures the relationship of people who ask questions and who comment on the question. Each edge (u, v) represents that user u's question has been commented by user v. This relationship is not as strong as the normal relationship among friends. The interactions among users who post questions and comments are not that frequent as the ones among friends. Therefore, the *influence spread increment* is relatively smaller than the *influence spread increment* in other datasets despite its large size.

6.2.2 Running Time

As depicted in Fig. 8, our proposed method *CSO* outperforms other compared methods, which shows that *CSO* is competitive in algorithm efficiency. Especially when graphs are large, the difference is much more significant, because running from scratch will result in more redundant computations in this case. Indices are designed to speed up the process of query; however, it will simultaneously bring computational overhead of maintenance. Our method greatly reduces query time compared with other methods; thus, the effectiveness of our index design is validated.



(a) Influence Spread Increment.



(b) Influence Spread vs Influence Spread Increment.

Fig. 7 Quality of seeds

6.2.3 Update Time of Index

Since the update operation is the most frequent operation when processing a streaming graph, we conduct experiment to measure the maintenance cost of index. The result is as follows (Fig. 9). When a graph is dense, the cost of update becomes higher. Since the number of related nodes, whose *influence spread increment* has changed, is larger, and the update cost for each of these nodes is also higher.

6.2.4 Memory Consumption

As shown in Fig. 10, our proposed method *CSO* could reduce the memory cost when compared with methods without compression.

6.2.5 Effect of Pruning Technique

The naive version of the insertion would incur large amount of unnecessary computations. And the prune version chips out the excessive calculation. We compare the average number of nodes that are visited during the insertion process. When an edge $e = (u, v, +, p, t)$ is inserted, as explained in the naive version, every ancestor of node u would update its intermediate results with the information of node v's reachable nodes ($v.II$). However, owing to the pruning technique, the ancestors are not necessarily visited and also the number of nodes that would make a difference to the ancestors' reachability is decreasing during the reverse *DFS* process. The average number of times that ancestors are visited is illustrated in Fig. 11a, and the average number of times that nodes in $v.II$ are visited is presented in Fig. 11b.

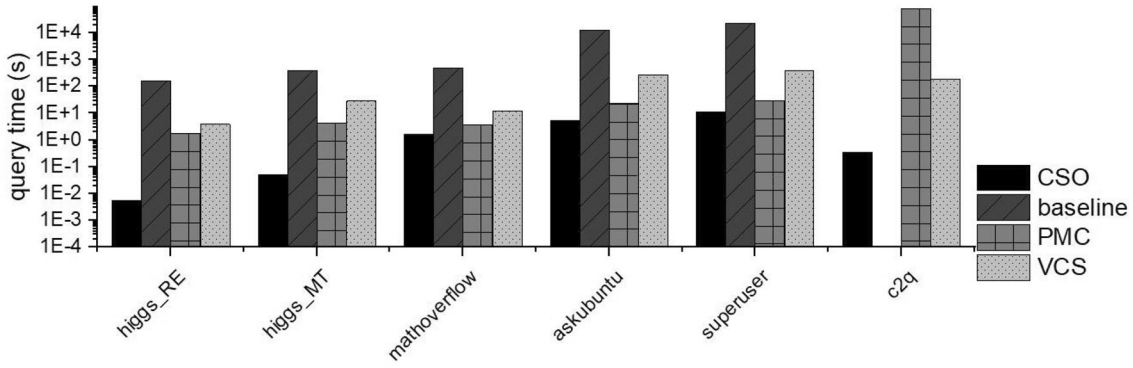


Fig. 8 Query time

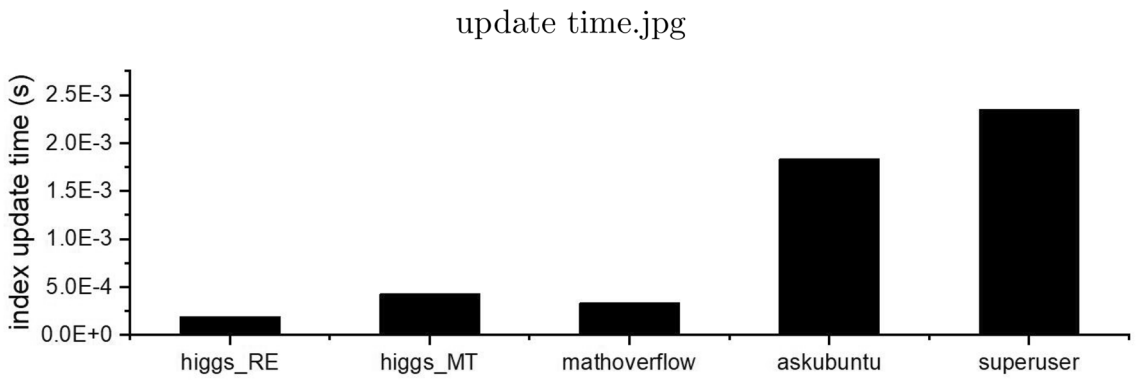


Fig. 9 The time of index update

As shown by the figures, the pruning technique reduces both the average number of times ancestor is visited and the average number of times node in $v.II$ is visited in all datasets. The decrease in the average number of times node in $v.II$ is visited is more significant.

6.2.6 Effect of Parameter θ

We evaluate the performance of *CSO* with different values of θ, including query time, insertion time, and *influence spread increment*. Owing to the transformation from equation $QC = sum(QS[\theta], QS[\theta - 1], \ldots, QS[1])$ to $QC = minusCount(addCount(QC, QS[0]), QS[\theta])$, the process of aggregating information across θ queries is simplified to one *addCount* and one *minusCount* operation. Thus, in Fig. 12a, the insertion time and query time are not increasing with θ. Interestingly, the query time is decreasing, which seems counterintuitive at first glance. However, this phenomenon is reasonable under further scrutiny. Under extreme circumstances, if θ is big enough, all information of previous selections is maintained. The number of nodes whose *influence spread increment* needs to be updated is small, since nearly all nodes are blocked. Consequently, the query time decreases. Additionally, we conduct experiment to measure *influence spread increment* of different θ. As shown in Fig. 12b, the *influence spread increment* decreases when θ increases. This is due to the fact that when θ becomes larger, the number of blocked nodes becomes larger. Therefore, generally, the *influence spread increment* becomes small.

7 Conclusion

In this paper, we propose and study the *SGIM* problem in which the graphs are modeled in a dynamic manner and the effect of query sequence is taken into account. While directly extending existing approach to address this issue will bring in non-negligible computational cost due to the redundant computation among sketches and queries, we design Influence-Increment-Index to avoid running from scratch along with two sketch-centralized indices called Influence-Index and Reverse-Influence-Index to facilitate the update process. We also design structure using sliding window and update algorithms to maintain evolving query sequence. By exploiting these components, we can answer the query at the latest

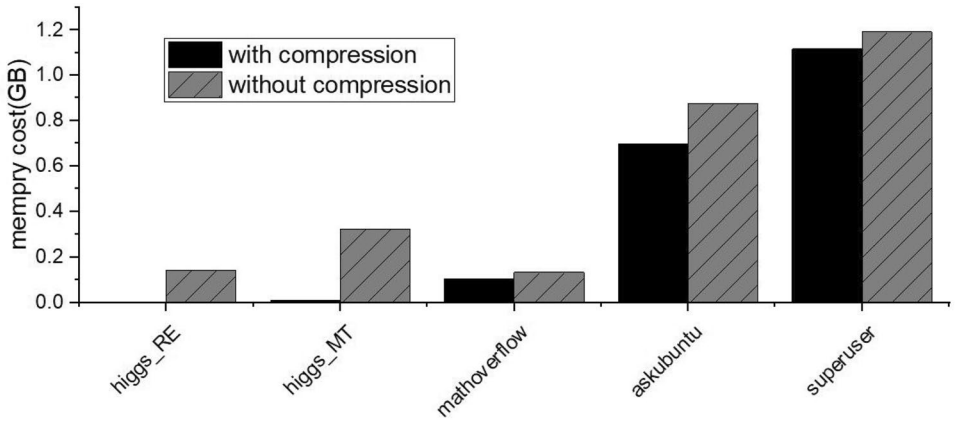
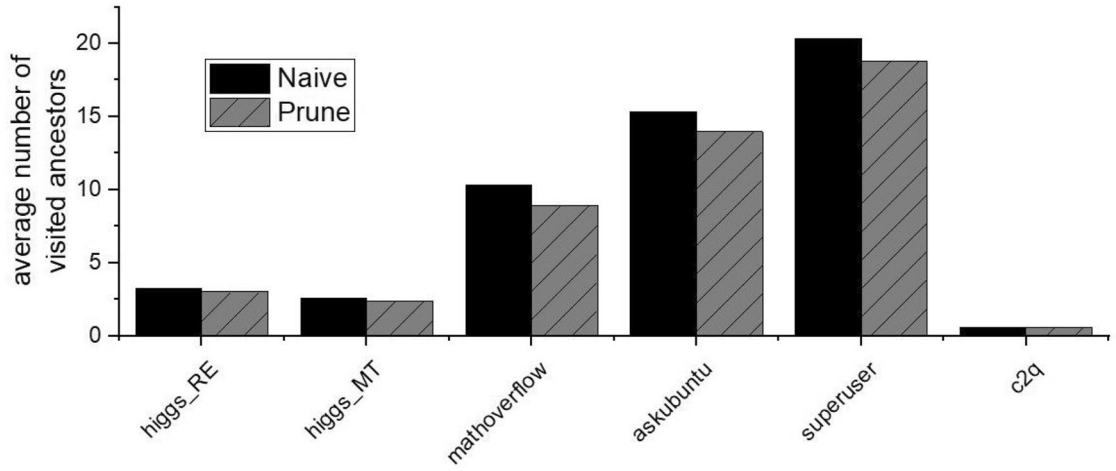
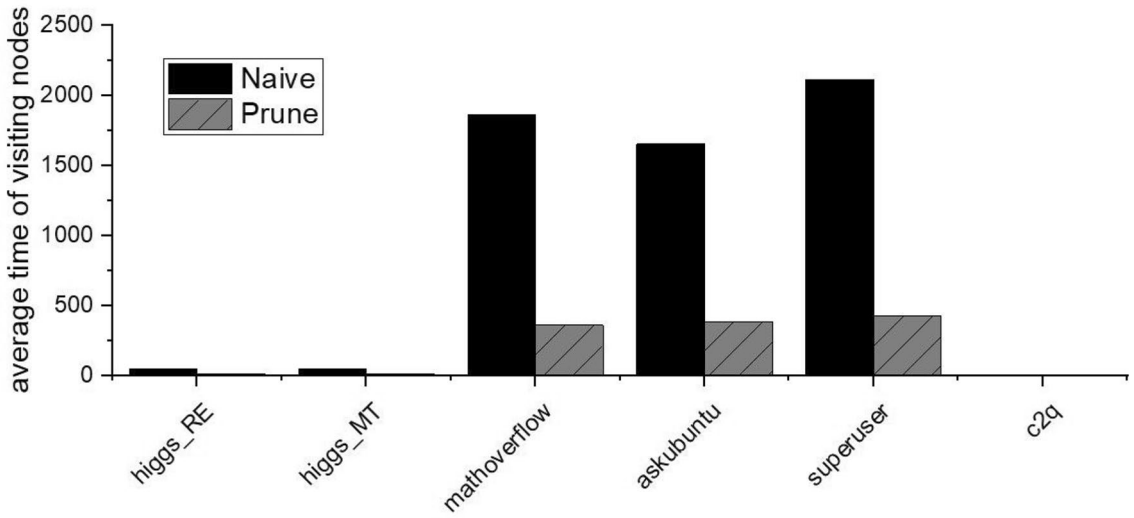


Fig. 10 The memory cost of intermediate results with and without compression



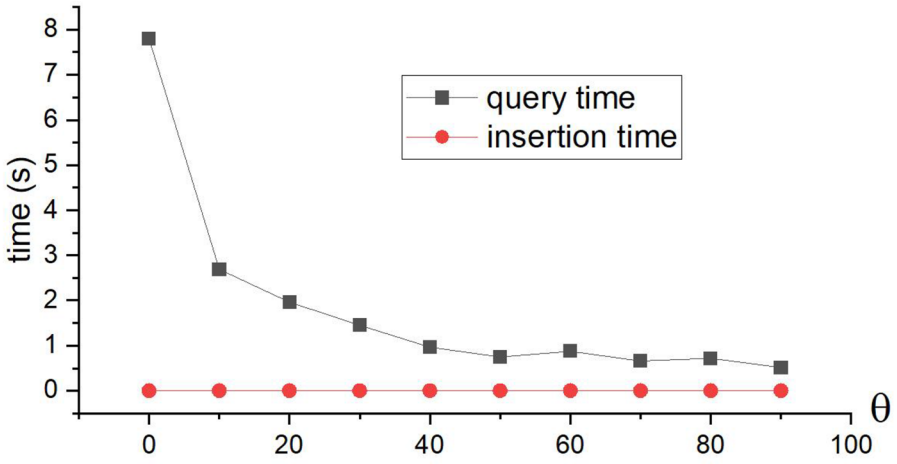
(a) The average number of times ancestor is visited.



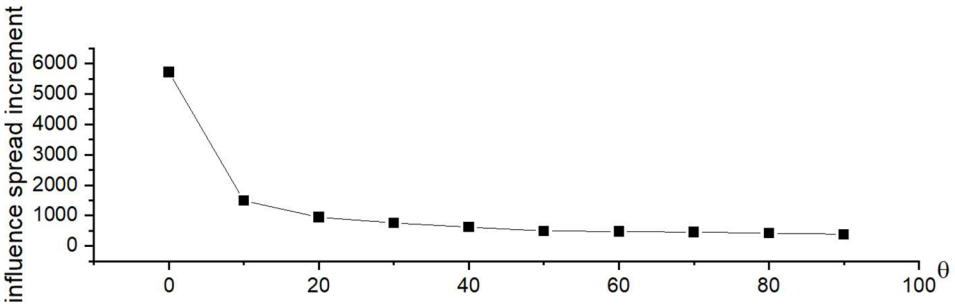
(b) The average number of times node in $v.II$ is visited.

Fig. 11 Number of visited nodes

Fig. 12 Effect of θ



(a) query time and insertion time of different θ.



(b) influence spread increment of different θ.

snapshot expeditiously. Extensive experiments on several real-world datasets have demonstrated that our method is competitive in terms of both efficiency and effectiveness owing to the indexing scheme.

In the future, we plan to further improve the scalability of the method and have a better performance on larger streaming graphs. One way to achieve this is to propose heuristic method to reduce the cost of updating *influence spread increment*. Additionally, the explore-exploit strategy, which is a trade-off between the value of activating some peripheral nodes versus giving the activated central nodes a second stimulation, will be applied to our method. Another future work is to extend reverse reachable sketch methods to solve this problem.

Acknowledgements The research is supported by The National Key Research and Development Program of China (No. 2018YFB1004002), NSFC (Nos. 62072205 and 61932004).

Author Contributions YZ and YH proposed the problem model, carried out the experiments, and contributed to the interpretation of the results. YZ designed the experiments with the guidance of PY. YZ and YH wrote the manuscript in consultation with PY and HJ. All authors provided critical feedback and helped shape the research, analysis and manuscript.

Availability of data and materials The datasets that are used in the experiment of this study are openly available in Stanford Large Network Dataset Collection. Mathoverflow dataset can be downloaded at https://snap.stanford.edu/data/sx-mathoverflow.html. Twitter-Higgs dataset can be downloaded at https://snap.stanford.edu/data/higgs-twitter.html. Askubuntu dataset can be downloaded at https://snap.stanford.edu/data/sx-askubuntu.html. Superuser dataset can be downloaded at https://snap.stanford.edu/data/sx-superuser.html. Stackoverflow-c2q dataset can be downloaded at https://snap.stanford.edu/data/sx-stackoverflow.html.

Declarations

Conflict of interest The authors declare that they have no competing interests.

References

1. Budak C, Agrawal D, Abbadi AE (2011) Limiting the spread of misinformation in social networks. In: WWW, pp 665–674
2. He X, Song G, Chen W, Jiang Q (2012) Influence blocking maximization in social networks under the competitive linear threshold model. In: SDM, pp 463–474
3. Ye M, Liu X, Lee W (2012) Exploring social influence for recommendation: a generative model approach. In: SIGIR, pp 671–680
4. Guille A, Hacid H, Favre C, Zighed DA (2013) Information diffusion in online social networks: a survey. SIGMOD Rec 42(2):17–28
5. Kempe D, Kleinberg J, Tardos É (2003) Maximizing the spread of influence through a social network. In: KDD, pp 137–146
6. Domingos P, Richardson M (2001) Mining the network value of customers. In: KDD, pp 57–66
7. Myers SA, Leskovec J (2014) The bursty dynamics of the twitter information network. In: WWW, pp 913–924
8. Song G, Li Y, Chen X, He X, Tang J (2017) Influential node tracking on dynamic social network: an interchange greedy approach. TKDE 29(2):359–372
9. Ohsaka N, Akiba T, Yoshida Y, Kawarabayashi KI (2016) Dynamic influence analysis in evolving networks. VLDB 9(12):1077–1088
10. Yang Y, Wang Z, Pei J, Chen E (2017) Tracking influential individuals in dynamic networks. TKDE 29(11):2615–2628
11. Zhao J, Shang S, Wang P, Lui JCS, Zhang X (2019) Tracking influential nodes in time-decaying dynamic interaction networks. In: ICDE, pp 1106–1117
12. Huang S, Bao Z, Culpepper JS, Zhang B (2019) Finding temporal influential users over evolving social networks. In: ICDE, pp 398–409
13. Datar M, Gionis A, Indyk P, Motwani R (2002) Maintaining stream statistics over sliding windows. SIAM J Comput 31(6):1794–1813
14. Leskovec J, Krause A, Guestrin C, Faloutsos C, VanBriesen JM, Glance NS (2007) Cost-effective outbreak detection in networks. In: KDD, pp 420–429
15. Goyal A, Lu W, Lakshmanan LVS (2011) CELF++: optimizing the greedy algorithm for influence maximization in social networks. In: WWW, pp 47–48
16. Wang Y, Cong G, Song G, Xie K (2010) Community-based greedy algorithm for mining top-k influential nodes in mobile social networks. In: KDD, pp 1039–1048
17. Chen W, Wang Y, Yang S (2009) Efficient influence maximization in social networks. In: KDD, pp 199–208
18. Chen W, Wang C, Wang Y (2010) Scalable influence maximization for prevalent viral marketing in large-scale social networks. In: KDD, pp 1029–1038
19. Cheng S, Shen H, Huang J, Zhang G, Cheng X (2013) Staticgreedy: solving the scalability-accuracy dilemma in influence maximization. In: CIKM, pp 509–518
20. Ohsaka N, Akiba T, Yoshida Y, Kawarabayashi K (2014) Fast and accurate influence maximization on large networks with pruned monte-carlo simulations. In: AAAI, pp 138–144
21. Cohen E, Delling D, Pajor T, Werneck RF (2014) Sketch-based influence maximization and computation: scaling up with guarantees. In: CIKM, pp 629–638
22. Borgs C, Brautbar M, Chayes JT, Lucier B (2014) Maximizing social influence in nearly optimal time. In: SODA, pp 946–957
23. Tang Y, Xiao X, Shi Y (2014) Influence maximization: near-optimal time complexity meets practical efficiency. In: SIGMOD, pp 75–86
24. Tang Y, Shi Y, Xiao X (2015) Influence maximization in near-linear time: a martingale approach. In: SIGMOD, pp 1539–1554
25. Han K, Huang K, Xiao X, Tang J, Sun A, Tang X (2018) Efficient algorithms for adaptive influence maximization. Proc VLDB Endow 11(9):1029–1040
26. Golovin D, Krause A (2011) Adaptive submodularity: theory and applications in active learning and stochastic optimization. J Artif Intell Res 42:427–486
27. Yuan J, Tang S (2017) No time to observe: adaptive influence maximization with partial feedback. In: Proceedings of the 26th international joint conference on artificial intelligence, pp 3908–3914
28. Tong G, Wu W, Tang S, Du D-Z (2016) Adaptive influence maximization in dynamic social networks. IEEE/ACM Trans Netw 25(1):112–125
29. Huang K, Tang J, Han K, Xiao X, Chen W, Sun A, Tang X, Lim A (2020) Efficient approximation algorithms for adaptive influence maximization. VLDB J 29:1–22
30. Wang Y, Fan Q, Li Y, Tan K (2017) Real-time influence maximization on dynamic social streams. Proc VLDB Endow 10(7):805–816
31. Aggarwal CC, Lin S, Yu PS (2012) On influential node discovery in dynamic social networks. In: SDM, pp 636–647
32. Zhuang H, Sun Y, Tang J, Zhang J, Sun X (2013) Influence maximization in dynamic social networks. In: ICDE, pp 1313–1318
33. Li Y, Fan J, Wang Y, Tan K (2018) Influence maximization on social graphs: a survey. TKDE 30(10):1852–1872
34. Paranjape A, Benson AR, Leskovec J (2017) Motifs in temporal networks. In: WSDM, pp 601–610
35. De Domenico M, Lima A, Mougel P, Musolesi M (2013) The anatomy of a scientific rumor. Sci Rep 3:2980