Distributed Near-Maximum Independent Set Maintenance over Large-scale Dynamic Graphs

Xubo Wang[†], Dong Wen[§], Wenjie Zhang[§], Ying Zhang[♮], Lu Qin[♮]

[†]*Zhejiang Laboratory, China;* [§]*University of New South Wales, Australia*
[♮]*AAII, University of Technology Sydney, Australia;*
[†]wangxb@zhejianglab.com; [§]{dong.wen, wenjie.zhang}@unsw.edu.au;
[♮]{ying.zhang, lu.qin}@uts.edu.au

***Abstract*—Computing the maximum independent set (MIS) in a graph is a fundamental NP-hard problem, which is widely adopted in many real-world applications. Extensive works have been done on computing an approximate MIS. While the highly dynamic property of real-world graphs calls for efficient MIS maintenance solutions, existing works for dynamic MIS computation in the literature mainly focus on the single-machine scenario. The assumption that a single machine can access the whole graph makes them difficult to be straightforwardly applied for large-scale graphs in distributed environment. Motivated by this, in this paper, we study the problem of maintaining approximate MIS over large-scale dynamic graphs in distributed environments. We propose a new vertex centric algorithm** OIMIS**. Compared with existing solutions,** OIMIS **avoids the strong order dependency in distributed computation, which makes it easy to handle dynamic graph updates.** OIMIS **computes and maintains MIS with high effectiveness and efficiency. In terms of high effectiveness,** OIMIS **maintains consistent MIS results with the state-of-the-art distributed algorithm to compute MIS in static graphs. In terms of high efficiency, each vertex in** OIMIS **only updates MIS status according to its neighbor attributes. Novel optimization techniques are also designed to reduce communication and computation cost. We conduct extensive experiments to prove the effectiveness and efficiency of our distributed algorithms.**

***Index Terms*—Maximum Independent Set, Dynamic Graph, Distributed Algorithm**

I. Introduction

Graphs have been widely used to represent relationships between entities in real-world applications such as social networks, web search, collaboration network, and biology. With the proliferation of graph applications, research efforts have been devoted to many fundamental problems in managing and analyzing graph data. Among them, the problem of computing the maximum independent set (MIS) is a classical NP-hard problem in graph theory and has attracted a lot of attention. Given a graph $G(V, E)$, a set I of vertices in G is an independent set if there is no edge between any pair of vertices in I. The MIS is an independent set of G that has the largest number of vertices. We show an example of MIS in Fig. 1. In graph G, $\{u_1, u_2, u_3\}$ is the maximum independent set. $\{u_2\}$ is also a maximal independent set of G, i.e., we cannot add any additional vertex, but it is not maximum.

Applications. Computing the MIS of a graph can be used in many real-world applications, such as collusion detection in voting pools [1], computing social network coverage and reach [2], automated labeling of maps [3], building index for processing shortest path/distance queries [4], [5], association rule mining [6] and information coding [7]. Moreover, MIS is closely related to other fundamental graph problems, such as minimum vertex cover, graph coloring, maximum common induced subgraphs, and maximum common edge subgraphs, etc. [8], [9].

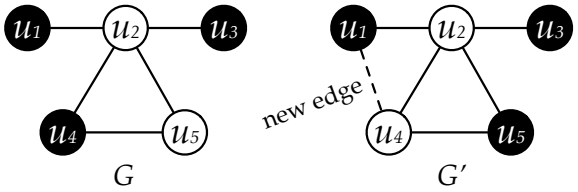


Fig. 1: An example of maximum independent set

Motivation. The scale of real-world graphs is large and grows exponentially. For example, the sub-domain of a web graph from EU countries contains 1.07 billion vertices and 91.79 billion edges. Computing MIS for such large graphs requires huge memory and computation resources that are expensive in a single machine. This makes the centralized algorithms designed for a single machine not scalable for real-world applications. To handle large-scale graphs, distributed computing clusters provide sufficient resources in a relatively easy and cheap way, and distributed MIS algorithms have been studied in the literature [10]–[12]. Although distributed MIS algorithms address the scalability problem, all of these existing algorithms focus on the static graphs while many real-world graphs are highly dynamic. To the best of our knowledge, there is not much efforts on solving maximum independent set problem on large dynamic graphs in distributed environments. In this paper, we aim to develop an efficient distributed algorithm to fill this research gap. Given that computing the MIS is NP-hard, it is easy to prove that maintaining MIS is also NP-hard [13]. We follow existing works [13], [14] to maintain a high-quality maximal (near maximum) independent set. In the rest, we simply use MIS to represent approximate maximum independent set when the context is clear. We will show the excellent result quality maintained by our algorithms theoretically and practically. An example can be found in Fig. 1. When inserting a new edge (u_1, u_4) into G, the MIS updates from $\{u_1, u_3, u_4\}$ to $\{u_1, u_3, u_5\}$.

Challenges. Most existing solutions on MIS maintenance focus on single machine scenarios. They usually employ sophisticated auxiliary data structures like a degeneracy graph in [15] and indexes in [14]. In distributed environments, constructing and synchronizing these sophisticated auxiliary information between different machines is costly and inefficient. We introduce more details for single-machine MIS maintenance in Section VIII. For distributed methods, the state-of-the-art solution for approximate MIS computation in static graphs, called DisMIS, runs in several rounds. Each round contains three supersteps. The first superstep adds several vertices into the result set. The second superstep marks several vertices not belonging to the result set, and the third superstep synchronizes information and gets prepared for the next round. The algorithm terminates when all vertices are either selected or deleted. We can see that the algorithm processes vertices in a certain order, and such order dependency makes it hard to be extended for edge updates. To maintain the result quality in dynamic settings, much information regarding the selection and deletion order needs to be kept. There are also some other distributed solutions based on probability models. However, their results are not deterministic so that the quality is not guaranteed. We target on generating the same quality result as DisMIS, as we observe that the result quality of DisMIS is excellent. In our experimental results over 16 small to large-scale real datasets from different domains, the independent set size is on average 98.2% of that of the state-of-the-art near-maximum independent set algorithm [16], which is developed in single-machine scenarios.

Main Idea. Our target is to design a distributed maintenance algorithm without sacrificing the result quality. To achieve this goal, we first propose a new distributed framework for independent set computation, called OIMIS. In OIMIS, we keep a boolean status for each vertex to indicate whether the vertex is in the result or not. We iteratively update their status until convergence. More specifically, in each iteration, each active vertex is removed from MIS if there is a higher-ranking neighbor belonging to MIS. The vertex ranks are defined based on degree. We prove that OIMIS guarantees the same result quality as DisMIS theoretically but relaxes the order dependency, which makes OIMIS easy to handle dynamic graph updates. More importantly, we prove that the maintained result is independent from the initial states of vertices and only dependent on the graph structure. This means our algorithm perfectly suits dynamic graphs and can generate consistent independent set regardless of the graph update order. We further improve efficiency by several optimization techniques. We design several rules to selectively activate vertices in our algorithm. The proposed activation strategies significantly reduce the number of active vertices in computation without breaking the correctness. We also show that the algorithm works well for batch updates.

Contributions. We summarize our main contributions in this paper as follows.

- *A simple distributed MIS framework for dynamic graphs.* We propose a simple distributed framework to efficiently compute a near maximum independent set in large dynamic graphs. Based on the vertex centric model, the framework is easy to be deployed and handles batch updates. As far as we know, this is the first work to study dynamic MIS maintenance in distributed environment.
- *Guaranteed result quality.* We prove the effectiveness of our framework for dynamic MIS maintenance. Our framework generates a consistent independent set result regardless of the order of edge updates. We also propose several optimization techniques to further improve its efficiency.
- *Extensive performance studies.* We conduct extensive experiments over 16 real graphs from different domains. The experimental results show the effectiveness and efficiency of our method.

Organization. The rest of this paper is organized as follows. Preliminaries are introduced in Section II. Section III reviews existing solutions. Our algorithms are presented in Section IV. Section V gives the optimization techniques. Section VI introduces the support of batch update by our algorithms. Section VII reports the performance studies. Section VIII introduces related works, and Section IX concludes the paper.

II. Preliminaries

Let $G = (V, E)$ be an undirected and unweighted graph, where $V(G)$ is the set of vertices and $E(G)$ is the set of edges, respectively. We denote the number of vertices as n and the number of edges as m, i.e., $n = |V(G)|$, $m = |E(G)|$. For a vertex $v \in V(G)$, we use $\mathsf{nbr}(v, G)$ to denote the set of neighbors of v. The degree of a vertex $v \in V(G)$, denoted by $\mathsf{deg}(v, G)$ is the number of neighbors of v, i.e., $\mathsf{deg}(v, G) = |\mathsf{nbr}(v, G)|$. For simplicity, we omit G in notations when the context is self-evident.

Definition 2.1: (Independent Set) Given a graph G, an independent set $I(G)$ of G is a subset of vertices of G such that $I(G) \subseteq V(G)$ and any pair of vertices in $I(G)$ is not connected in G, i.e., $\forall\ u, v \in I(G)$, then $(u, v) \notin E(G)$.

Definition 2.2: (Maximum Independent Set) Given a graph G, a maximum independent set (MIS) of G, denoted by $M(G)$, is an independent set with the largest possible size in G.

Example 2.1: An example of MIS is presented in Fig. 1. $\{u_1, u_3, u_4\}$ is the MIS of G. $\{u_2\}$ is also a maximal independent set of G, and we cannot add any additional vertex to the result. However, it is not maximum. □

Problem statement. In this paper, we study the problem of distributed dynamic MIS, which is defined as follows: Given a graph G, compute the MIS of G in distributed environments when G dynamically updates by insertion/deletion of edges.

Given that computing the MIS is an NP-Hard problem [17] and it is also NP-Hard to maintain an exact MIS over dynamic graphs [18], we resort to approximate (near maximum) solutions in this paper following existing works. When the context is clear, we use MIS for short to represent approximate Maximum Independent Set in the rest.

Distributed graph processing framework. We follow the vertex-centric framework and develop distributed algorithms in Pregel-like systems [19]–[21], which are commonly used in distributed graph processing. A distributed algorithm deployed in Pregel-like system runs in supersteps. In each superstep, each vertex computes and communicates with others based on functions defined by users. The algorithm terminates until no vertex is active.

III. Existing Solution for Static Graphs

Given that the distributed algorithm for MIS maintenance is less studied, we review the distributed algorithm for MIS computation in static graphs in this section. As far as we know, the state-of-the-art distributed solution for static graphs is DisMIS [22], which computes a maximal independent set with a high quality. Other related works will be introduced in Section VIII. DisMIS is based on Luby's classic parallel algorithm [23], where every vertex in the graph belongs to one of the following three status.

- In: The vertex is in MIS.
- NotIn: The vertex is not in MIS.
- Unknown: The vertex has yet confirmed in MIS or not.

Initially, the status of all vertices are Unknown. When the algorithm terminates, each vertex belongs to either In or NotIn. Note that a vertex u must connect to at least one neighbor belonging to MIS if u is in NotIn. Otherwise, we can just add u to the result MIS. DisMIS also requires a predefined vertex order. We use the degree order which is commonly utilized in many works for approximate MIS computation [24]–[26].

Definition 3.1: (Vertex Total Order $\prec$) Given a graph G and two vertices $u, v \in V(G)$, we have $u \prec v$ if

(i) $\mathsf{deg}(u) < \mathsf{deg}(v)$, or
(ii) $\mathsf{deg}(u) = \mathsf{deg}(v)$ and $\mathsf{id}(u) < \mathsf{id}(v)$.

For two vertices u and v, if $u \prec v$, we say u dominates v or u ranks higher than v. It is easy to say that $\prec$ defines a total order of all vertices in G, i.e., we have $u \prec w$ if $u \prec v$ and $v \prec w$. Compared with the random order, the degree order significantly enlarges the result MIS [24]. In our performance studies, we use vertex IDs from the original dataset to break the tie of the vertex degree. A different vertex ID assignment (e.g., a random assignment) would not affect the result quality too much, because degrees of two connected vertices are different in most cases.

Algorithm. Based on Definition 3.1, the pseudocode of DisMIS is presented in Algorithm 1. DisMIS maintains three sets of vertices during the computation. Accordingly, every vertex u is considered in the Unknown set at the beginning in line 2. u.info is set as the degree of u in line 3, which will be used to determine the $\prec$ relation in the following. At last, the message containing the id of u, the status of u, and the degree of u, namely $(u, u.\mathsf{status}, u.\mathsf{info})$, is broadcast to the neighbors of u in line 4.

After the initialization, DisMIS iteratively selects the vertices from the Unknown set into the In set (selection supersteps) and moves the vertices that are adjacent to the vertices

Algorithm 1: DisMIS

```
Input: A graph G = (V, E)
Output: MIS of G
   /* Initialization                                   */
1  if superstep() = 0 then
2      u.status ← Unknown;
3      u.info ← deg(u);
4      broadcast_to_neighbors(u, u.status, u.info);
5  else
6      switch superstep()%3 do
7          case 1 do
               /* selection superstep                  */
8              if u.status = Unknown then
9                  u.count ← 0;
10                 for each (v, v.status, v.info) ∈ Msg(u) do
11                     if v ≺ u and v.status = Unknown
                       then
12                         u.count ← u.count + 1;
13                 if u.count = 0 then
14                     u.status ← In;
15                     broadcast_to_neighbors(u);
16         case 2 do
               /* deletion superstep                   */
17             if Msg(u) ≠ ∅ and u.status = Unknown then
18                 u.status ← NotIn;
19                 broadcast_to_neighbors(u);
20         case 0 do
21             if Msg(u) ≠ ∅ and u.status = Unknown then
22                 broadcast_to_neighbors(u, u.status, u.info);
23 VoteToHalt();
```

in the In set to the NotIn set (deletion supersteps) until no vertices left in the Unknown set (lines 6-22). Specifically, in a selection superstep, if a vertex u is still in the Unknown set (line 8), it checks all the received messages. If u dominates all its neighbors v that still in the Unknown set (lines 9-13), u is put in the In set by setting $u.\mathsf{status} = \mathsf{In}$ (line 14). The algorithm sends an empty message to notify all its neighbors (line 15). In a deletion superstep, if any neighbor of a vertex u is selected in the last selection superstep, u is put into the NotIn set by setting $u.\mathsf{status} = \mathsf{NotIn}$, and we informs its neighbors by sending an empty message (lines 17-19). At last, for the vertices in the Unknown set whose neighbors' state is changed in the last superstep, the message $(u, u.\mathsf{status}, u.\mathsf{info})$ is broadcast to its neighbors for the next selection superstep. The algorithm finishes when there are no vertices left in the Unknown set (line 23), and we call VoteToHalt() to terminate the vertex-centric procedure.

Example 3.1: Fig. 2 shows a running example of DisMIS. The vertex state in each superstep is presented. A vertex is

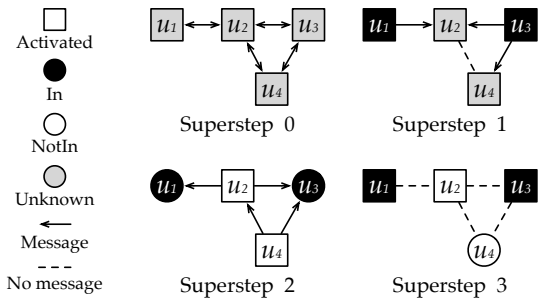


Fig. 2: Running example of Algorithm 1

activated (shown as a square) if it receives messages sent from the last superstep. Initially (superstep 0), all vertices are in the Unknown set and are activated. In superstep 1, every vertex receives all neighbors' messages, and two vertices u_1, u_3 satisfy line 13 of Algorithm 1. Therefore, we mark u_1 and u_3 as In and broadcast the updated messages to their neighbors, which is shown in the figure of superstep 1. Then in superstep 2, u_2 is activated by receiving the messages from u_1 and u_3. We mark u_2 as NotIn and notify its neighbors. We do the same thing for u_4 in superstep 2. Finally, in superstep 3, u_1, u_2, and u_3 are activated because they receive messages. However, there are no updates, and the algorithm terminates without any new message. □

DisMIS guarantees the result independent set is maximal, and we observe that the result quality produced by DisMIS is excellent. In our experimental results over fifteen real datasets from different domains, the average MIS size of DisMIS is around 98.2% of that of the state-of-the-art near-maximum independent set SPC algorithm [16], which is designed for single-machine in-memory scenarios. Maintaining MIS for dynamic graphs effectively and efficiently in single machines is already a challenge [15]. In this paper, we aim to maintain the exact good quality result derived by DisMIS for large-scale dynamic graphs in distributed environments. Specifically, given a graph G and a series of graph updates (edge insertions and/or deletions) δ_G, we aim to design an algorithm $\mathcal{A}$ such that $\mathcal{A}(M(G), \Delta_G) \Rightarrow M(G \biguplus \Delta_G)$, where $G \biguplus \Delta_G$ is the updated graph after edge insertions/deletions.

Challenges. Although DisMIS is effective and efficient to compute MIS, it is not easy extending it to handle dynamic graph updates. As shown in Algorithm 1, DisMIS follows a total degree order for vertices and a global selection-and-deletion scheme in each round of three supersteps, which makes vertices added to MIS in a certain order. We call this kind of algorithms *order dependent*. Specifically, a vertex u changes its status to In or NotIn in DisMIS only if there is no neighbor of u ranking higher than u with an Unknow status in the last superstep. Given an edge insertion or an edge deletion (u, v), the degrees of u and v change. The status of u, v and all their neighbors may be affected since the vertex rank is based on the degree. Given a set of edge updates, we call all terminal vertices and their neighbors the affected vertices.

To extend DisMIS for dynamic graphs, an intuitive method is to maintain all intermediate state (e.g., messages, status) for all vertices. The additional space takes $O(m \cdot k)$ where m is the number of edges and k is the number of supersteps. In the initial superstep, we process all affected vertices. All other vertices derive the status from the stored states, and do not send messages to neighbors. In following supersteps, if a vertex receives a message, we reprocess the vertex according to DisMIS and send messages if necessary. Otherwise, we used the stored history to recover the status of the vertex. Compared with the static DisMIS algorithm, the extension may save some network messages between irrelevant vertices. However, the update algorithm runs at least the same number of supersteps as DisMIS. In addition, this method maintains a bunch of side information which is not scalable for large graphs.

Algorithm 2: OIMIS

Input: A graph $G = (V, E)$;
Output: $M(G) = \{u | u.\mathsf{in} = true\}$

```
1  if superstep() = 0 then
2      u.in ← true;
3  else
4      old ← u.in; u.in ← true;
5      for each v ∈ N(u) do
6          if v ≺ u && v.in = true then
7              u.in ← false;
8              break;
9      if u.in ≠ old then
10         for each v ∈ N(u) do
11             activate v;
12 VoteToHalt();
```

IV. Order-Independent MIS

To avoid the order dependency in DisMIS, we first propose a novel distributed framework to compute MIS in statics in Section IV-A, which relaxes the order dependency. We will show that our new framework is more efficient and produces the same result as DisMIS. Based on our new framework for MIS computation, we propose our distributed algorithm for graph updates in Section IV-B.

A. Order Independent MIS Computation

Our main idea is to remove the selection-and-deletion computing order in DisMIS. We observe the following local property in the process of DisMIS.

Observation 4.1: *A vertex u is not in* MIS *if and only if there exists a vertex v such that $v \in \mathsf{nbr}(u)$ and $v \in$ MIS.*

Based on the above observation, we propose to exploit just one simple update function for each vertex for distributed processing, instead of the complex multi-phase scheme in DisMIS. In each superstep, each vertex updates its own attribute based on the messages from neighbors. Our new distributed framework for MIS computation is called OIMIS (order-independent MIS). The pseudocode is presented in Algorithm 2.

Following the vertex-centric computing framework, we conduct the local property check on each vertex in a series of

supersteps until all vertices are inactive. Specifically, each vertex u is assigned with just one boolean attribute u.in which indicates whether u is in $M(G)$ when the algorithm terminates. At the beginning, u.in is set as $true$ for all vertices (lines 2–3). Then, instead of selecting vertices in one phase and deleting their neighbors in the other phase, all vertices perform the same computation in all supersteps. Each vertex u changes its attribute u.in to true in line 4. Then u.in updates from $true$ to $false$ when one of its higher-ranking neighbors v satisfies $v.\mathsf{in} = true$ (lines 5–8). Otherwise, u.in stays as $true$. If u.in changes in the current superstep, u will activate its neighbors to re-examine the local property. The algorithm terminates when no vertex updates.

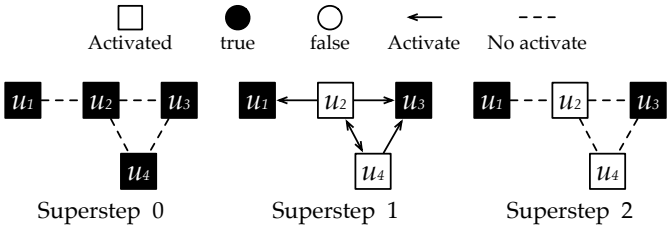


Fig. 3: A running example of OIMIS

Synchronization-based Computing Model. To implement Algorithm 2, a crucial step is to access the status of all neighbors (line 5) for each vertex. That means we need the information of a neighbor vertex v even though v was not active and changed its attribute in the last superstep. We may force all vertices send their own attributes to neighbors in each superstep in a traditional Pregel computing model [19]. It undoubtedly incurs considerable computation and communication costs. To handle this issue, we implement our algorithm based on a Pregel variation ScaleG [21]. ScaleG is a synchronization-based model for vertex centric computation. For each vertex u, it maintains a copy (called a guest vertex) of u in each other machine. At the end of each superstep, ScaleG updates the copy of u in each other machine if the state of u changes. In this way, each vertex can locally access all neighbors' information in the next superstep. To active vertices, ScaleG uses an inverted index to connect each guest to local vertices. A vertex can active its neighbors in other machines via its guest vertex to reduce communication cost. ScaleG is shown to be significantly efficient than Pregel [21], but at the price of sacrificing non-neighbor communication. Note that our algorithm is independent of platforms and works on all Pregel-like graph processing systems. Given an edge update, the updated neighbors of each vertex is only stored locally. For ScaleG, the updated degree of a vertex will be sent to its copies in other machines, and its neighbors may read the vertex's degree locally when necessary. Compared with directly sending to neighbors in Pregel, this mechanism helps reducing the communication cost. It is also worth to mention that there is no extra cost for vertex order maintenance since we do not need to know the rank of a vertex in the total order. In other words, we only require comparing ranks between two given vertices. To this end, we only need the degree and the id of each vertex, which is stored as local attributes in distributed systems.

Algorithm 3: Dynamic OIMIS (DOIMIS)

Input: A graph $G = (V, E)$ and $M(G)$; an update operation op
Output: $M(G \uplus op)$

1 activate vertices in $\mathcal{AFF}_{op}$;
2 **for each** *active vertex* **do**
3 repeat Line 3-12 in Algorithm 2;

Example 4.1: A running example of OIMIS is shown in Fig. 3. Initially, all vertices are activated and marked as $true$. Then, each vertex updates value based on its neighbors. In superstep 1, u_1.in holds as $true$ since all its neighbors ranks lower. u_2.in is set to $false$ because its neighbor u_1 ranks higher and u_1.in is $true$. Similarly, u_3.in $= true$ and u_4.in $= false$. After the second superstep, no vertex changes its value, and the algorithm stops. As we can see, the result is the same as the example (Fig. 2) of order-independent algorithm DisMIS. □

We prove the correctness of OIMIS as follows.

Theorem 4.1: *Given a graph $G = (V, E)$, the* MIS *computed by* DisMIS *and* OIMIS *are the same, i.e.,* DisMIS(G) = OIMIS(G).

Proof. Let $M_{\mathsf{dis}}(G)$ and $M_{\mathsf{oi}}(G)$ be the resulting MIS produced by DisMIS and OIMIS, respectively. To prove that $M_{\mathsf{dis}}(G) = M_{\mathsf{oi}}(G)$, we show that for any $u \in V$, $u \in M_{\mathsf{dis}}(G) \Rightarrow u \in M_{\mathsf{oi}}(G)$ and $u \notin M_{\mathsf{dis}}(G) \Rightarrow u \notin M_{\mathsf{oi}}(G)$. If $u \in M_{\mathsf{dis}}(G)$, then $u.\mathsf{status} = \mathsf{In}$. In Algorithm 1, $u.\mathsf{status}$ only changes to In in case 1 where all of its previous neighbors are deleted. According to Algorithm 2, if all of u's higher-ranking neighbors are deleted, then the condition in line 6 never stands. Thus, $u.in = true$ and $u \in M_{\mathsf{oi}}(G)$. If $u \notin M_{\mathsf{dis}}(G)$, $u.\mathsf{status} = \mathsf{NotIn}$. In Algorithm 1, $u.\mathsf{status}$ changes to NotIn in case 2 when any of its neighbors is selected. Assume w is the highest-ranking neighbor that broadcasts the selection message to u and causes $u.\mathsf{status} = \mathsf{NotIn}$. That is $w.\mathsf{status} = \mathsf{In}$. Then $w \prec u$. According to Algorithm 2, w also causes $u.\mathsf{in} \leftarrow false$ (line 6). Thus, $u \notin M_{\mathsf{oi}}(G)$. To sum up, $M_{\mathsf{dis}}(G) = M_{\mathsf{oi}}(G)$. □

Number of Supersteps. Recall that the status of each vertex only changes once in DisMIS. It is easy to see that for each vertex changing status from Unknown to In in the k-th superstep of DisMIS, the vertex must change status to true no later than k-th superstep and would not change any more in OIMIS. Therefore, the number of supersteps of OIMIS is guaranteed not larger than that of DisMIS. As discussed in [22], the number of supersteps of DisMIS is bounded by $O(n)$. Our proposed OIMIS is also bounded by $O(n)$.

B. Dynamic Graphs

Based on the new framework OIMIS for MIS computation, we show our MIS maintenance algorithm, called DOIMIS,

for dynamic graphs. Algorithm 3 presents the pseudo-code of DOIMIS when inserting a new edge or deleting an old edge. The idea of DOIMIS is to first locate an affected area and then activate all affected vertices to perform the same procedure as OIMIS. Given a graph G and an update operation op, the algorithm first updates the graph $G' = G \biguplus op$, and activates all affected vertices $\mathcal{AFF}_{(u,v)}$ from the update operation (Line 1). Then, the active vertices conduct the same steps as lines 3–12 of Algorithm 2 until no vertex is active. The affected vertices are derived as follows.

Definition 4.1: (Affected Vertices) Given a new edge or an existing edge (u, v), the affected vertices of (u, v), denoted by $\mathcal{AFF}_{(u,v)}$, is the terminal vertices and all their neighbors, i.e., $\mathcal{AFF}_{(u,v)} = \{u\} \cup \{v\} \cup \mathsf{nbr}(u) \cup \mathsf{nbr}(v)$.

The idea behind the definition of affected vertices is that the update operation either increases or decreases the degree of terminal vertices. This would break the local property at terminal vertices and their neighbors. Thus, these vertices are activated to re-run the algorithm till local property is maintained at all vertices. Note that the result is still a MIS after inserting or deleting an edge between two vertices that are not in MIS. For the case of edge insertion, we do not active any vertex for the edge (line 1 of Algorithm 3). However, for the case of edge deletion, the quality of the result may be affected. To maintain a high-quality MIS, we follow existing studies [24–27] to maintain the MIS derived based on the degree order (Definition 3). Assume that we delete an edge (u, v) where u and v are not in MIS. The degrees of both u and v are decreased. Assume that there is a neighbor w of u such that w is in MIS, decreasing the degree of u may make u rank higher than w. As a result, we may add u into MIS instead of w. Therefore, even if two terminals of the deleted edge are not in MIS, they need to be processed to guarantee the correctness.

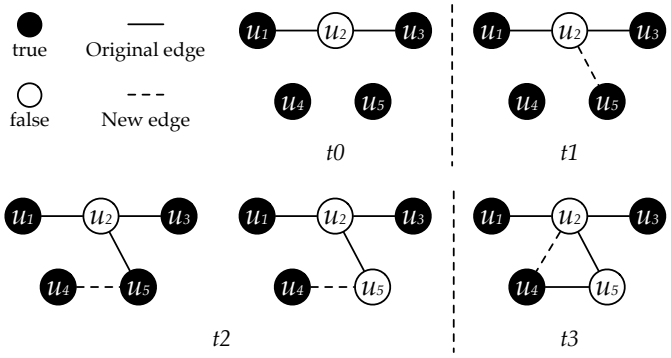


Fig. 4: A running example of DOIMIS

Example 4.2: A running example of DOIMIS is shown in Fig. 4. At t_1, an edge (u_2, u_5) is inserted. The degree of both u_2 and u_5 both increases by 1. All vertices in the affected vertex set $\{u_2, u_5, u_1, u_3\}$ are activated. After the computation, no vertex's value changes, and the algorithm stops. We perform the similar process for insertions of (u_4, u_5) at t_2 and (u_2, u_4) at t_3. □

Next, we theoretically prove the result quality of DOIMIS.

Theorem 4.2: *Given a graph G and an edge update op, the independent set generated by* DOIMIS *is the same as that of* OIMIS*, i.e.,* $\mathsf{DOIMIS}(G, \mathsf{OIMIS}(G), op) = \mathsf{OIMIS}(G \biguplus op)$*, where $G \biguplus op$ is the updated graph.*

Proof. We use M_{doi} and M_{oi} to denote the MIS computed by DOIMIS and OIMIS, respectively. For any $u \in V$, either $u \in M_{\mathsf{doi}}(G, M_{\mathsf{oi}}(G), op)$ or $u \notin M_{\mathsf{doi}}(G, M_{\mathsf{oi}}(G), op)$ is true. Similar to the proof of Theorem 4.1, we show that for any $u \in V$, $u \in M_{\mathsf{doi}}(G, M_{\mathsf{oi}}(G), op) \Rightarrow u \in M_{\mathsf{oi}}(G')$ and $u \notin M_{\mathsf{doi}}(G, M_{\mathsf{oi}}(G), op) \Rightarrow u \notin M_{\mathsf{oi}}(G')$ to prove that $M_{\mathsf{doi}}(G, M_{\mathsf{oi}}(G), op) = M_{\mathsf{oi}}(G')$ where $G' = G \biguplus op$. If $u \in M_{\mathsf{doi}}(G, M_{\mathsf{oi}}(G), op)$, then $u.\mathsf{in} = true$ after the computation of Algorithm 2. Then, all of $u's$ higher-ranking neighbors where $v \prec u$ has $v.\mathsf{in} = false$. Since the computation is on updated graph, this conclusion also stands for $M_{\mathsf{oi}}(G')$. If $u \notin M_{\mathsf{oi}}(G')$, then there must exist a higher-ranking neighbor $v \in N(u)$ and $v \prec u$ where $v.\mathsf{in} = true$. This contradicts with the above conclusion. Thus, $u \in M_{\mathsf{oi}}(G')$. Similarly, we can prove that if $u \notin M_{\mathsf{doi}}(G, M_{\mathsf{oi}}(G), op)$, then $u \notin M_{\mathsf{oi}}(G')$. Therefore, $M_{\mathsf{doi}}(G, M_{\mathsf{oi}}(G), op) = M_{\mathsf{oi}}(G')$.

□

Order Independency. Based on Theorem 4.2, we can see that the result of DOIMIS is independent of the order of update operations. The result is only dependent on the updated graph structure. As long as the final updated graph G' is the same, $M(G')$ is the same.

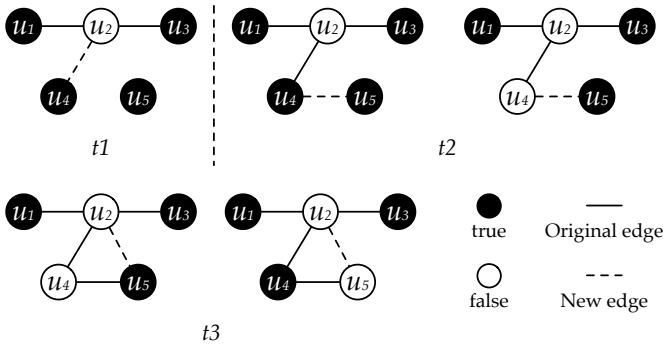


Fig. 5: A running example of DOIMIS for a different order of edge insertions

Example 4.3: A running example of a different edge insertion order from Example 4.2 is shown in Fig. 5. As we can see, the MIS results are the same. □

V. Selective Activation

In this section, we introduce several optimization techniques to improve the efficiency of our algorithms. Our idea is to reduce the number of active vertices in each supersteps, while guarantees the correctness. Less active vertices helps reducing the communication cost (messages between machines) and the computation cost (local vertex computation). The optimizations can be applied in both OIMIS and DOIMIS.

A. Activating Lower-Ranking Neighbors

When a vertex's status changes (line 9 of Algorithm 2), its neighbors are activated to update their states. We make the following observation.

OBSERVATION 5.1: *A vertex is only influenced by its higher-ranking neighbors in Algorithm 2.*

To explain Observation 5.1, we show the following lemma.

Lemma 5.1: *In Algorithm 2, a vertex u changes its* in *status only if there is a higher-ranking neighbor v of u (i.e., $v \in \mathsf{nbr}(u) \wedge v \prec u$) changes status in the last superstep.*

Proof. The lemma is straightforward since we only consider higher-ranking neighbors (line 6 of Algorithm 2) in the computation of each vertex. □

Therefore, the result of Algorithm 2 is the same if we only activate lower-ranking neighbors for each vertex. To implement this, line 10 in Algorithm 2 is replaced with for each $v \in \mathsf{nbr}(u) : u \prec v$. The selective activation strategy significantly reduces computation and communication cost of OIMIS since less vertices are activated. This is proved by our experimental results in Section VII-B. Note that the supersteps are not reduced by lower-ranking activation because of Observation 5.1.

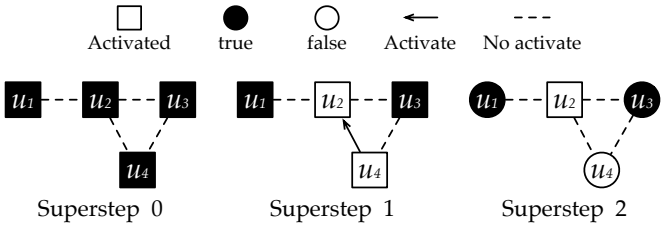


Fig. 6: An example of activating only lower-ranking neighbors in OIMIS

Example 5.1: A running example of activation lower-ranking neighbors is shown in Fig. 6. In superstep 2, when vertex u_2.in changes, no neighbor is activated since all its neighbors ranks higher than u_2. Compared with Fig. 3, only u_2 is active in superstep 2. This helps to reduce the communication cost from five messages in Fig. 3 to only one message in Fig. 6. □

B. Activating Same-Status Neighbors

We propose another optimization to further reduce the number of activated vertices, which is motivated by the following observation.

OBSERVATION 5.2: *A vertex updates its status only if there is a conflict (the same status value) between the vertex and its neighbors.*

Intuitively, a vertex cannot be in MIS if there is a neighbor with higher priority already in MIS. On the other hand, a vertex not in MIS should be considered joining MIS to enlarge MIS if there is no neighbor in MIS. Based on Observation 5.2 and Lemma 5.1, we have the following lemma.

Lemma 5.2: *In Algorithm 2, a vertex u changes its* in *status only if there is a neighbor v of u changes status to the same as u in the last superstep.*

Proof. We first consider the case that u.in changes from $true$ to $false$. Line 6 of Algorithm 2 never holds in the last superstep such that u is active. Otherwise, u.in cannot be $true$ in the beginning of the current superstep. Assume there is no neighbor changing status to $true$ in the last superstep. Line 6 of Algorithm 2 would still not hold in the current superstep. u.in cannot be $false$, which contradicts our case. Then, we consider the case that u.in changes from $false$ to $true$. Line 6 of Algorithm 2 holds in the last superstep such that u is active. Assume there is no neighbor changing status to $false$. Line 6 of Algorithm 2 would still hold in the current superstep, which contradicts that u.in changes from $false$ to $true$. □

Based on Lemma 5.2, we can only activate neighbors whose status are the same as the current vertex. To apply this, we modify the line 10 of Algorithm 2 as for each $v \in \mathsf{nbr}(u) : u \prec v \wedge v.\mathsf{in} = u.\mathsf{in}$. Different from lower-ranking activation, supersteps may be reduced by same-status activation. This is because vertices that have different in values that are activated without same-status activation are not activated now. This may make the algorithm converge early.

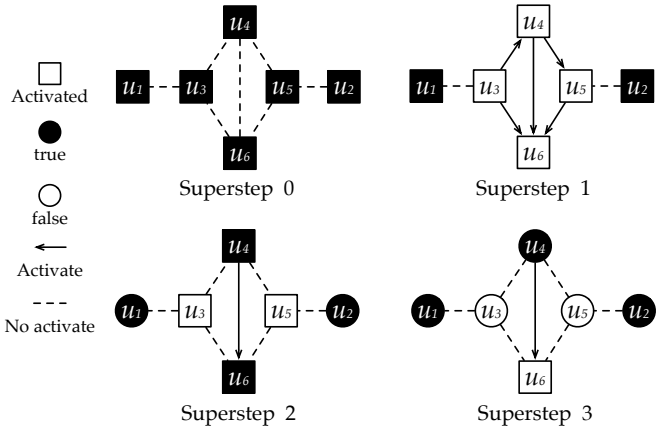


Fig. 7: Activating same-status neighbors in OIMIS

Example 5.2: An example is shown in Fig. 7. During the computation, when u_4.in changes in superstep 2 and needs to activate its neighbors, both u_5 and u_6 rank lower than u_4. However, we only activate u_6 because u_4.in is $true$ at the end of superstep 2 while u_5.in is $false$. We can see the result is correct when algorithm terminates. We avoid the computation and communication cost of u_5 when only activating same-status neighbors. □

Example 5.3: We also show an example in Fig. 8, which presents the effectiveness of only activating same-status neighbors on dynamic MIS maintenance. When edge (u_3, u_4) is deleted, vertex u_3 and u_4 recompute their values. u_3.in changes from $false$ to $true$. Then u_3 needs to activate its lower-ranking neighbors which is u_2. Given that u_2.in $=$

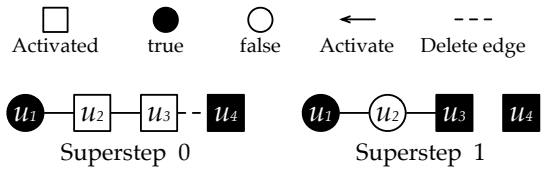


Fig. 8: Activating same-status neighbors in DOIMIS

$false$ and $u_3.\mathsf{in} = true$, u_3 does not activate u_2 even though u_2 ranks lower than u_3. The algorithm stops after the superstep 1. As we can see, the strategy of only activating same-status neighbors saves computation and communication cost of one superstep in this example. □

VI. Handleing Batch Update

In this subsection, we show that our algorithm supports batch update operations effectively. Let OP be a set of graph updates. For each edge $e \in OP$, e can be a new edge for insertion or an existing edge for deletion.

Based on Definition 4.1 and Theorem 4.2, the affected vertices are all vertices in OP and their neighbors, i.e., $\mathcal{AFF}_{OP} = \{u | u \in OP \vee \mathsf{nbr}(u) \cap OP \neq \emptyset\}$. We activate all vertices in $\mathcal{AFF}_{OP}$ in line 1 of Algorithm 3 given a set of edge updates OP. The rest of the batch update algorithm is the same as Algorithm 3.

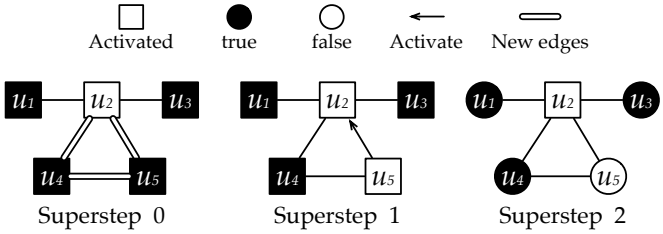


Fig. 9: A running example of batch updates by DOIMIS

Example 6.1: A running example of DOIMIS for batch update is shown in Fig. 9. All updates are new edges, i.e., $OP = \{(ins, u_2, u_4), (ins, u_2, u_5), (ins, u_4, u_5)\}$. In this case, all vertices are affected and activated initially. Only u_5 updates its status in superstep 1. Then, by applying our optimizations, only u_2 is activated in superstep 2, and algorithm terminates.The result is the same as that of processing each single update sequentially in Fig. 4. □

Theorem 6.1: *Given a graph G and update operations $OP = \{op_1, op_2, ..., op_k\}$, updated graph $G' = G \uplus OP$. The independent set generated by* DOIMIS *is the same as that of* OIMIS. *That is, $M_{\mathsf{doi}}(G, M_{\mathsf{oi}}(G), OP) = M_{\mathsf{oi}}(G')$.*

Proof. This is intuitive because the proof of Theorem 4.1 shows that $M_{\mathsf{doi}}(G, M_{\mathsf{oi}}(G), OP)$ is independent of the update order but only the graph structure after updating. We can prove this theorem similar to that of Theorem 6.1. The only minor difference is that multiple update operations are applied on the graph. However, the local property is kept on all vertices. Therefore, $M_{\mathsf{doi}}(G, M_{\mathsf{oi}}(G), OP) = M_{\mathsf{oi}}(G')$. □

TABLE I: Characteristics of datasets. $|V|$ and $|E|$ represent the number of vertices and edges respectively. deg_{avg} denotes the average vertex degree in each dataset.

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	deg_{avg}
Slashdot/SL	82,168	504,230	12.27
Amazon/AM	334,863	925,872	5.53
Google/GO	875,713	4,322,051	9.87
Dblp/DB	986,207	13,414,472	27.20
Skitter/SKI	1,696,415	11,095,298	13.08
Wikitalk/WK	2,394,385	4,659,565	3.89
Orkut/OR	2,997,167	106,349,209	70.97
UK-2002/UK02	18,520,343	261,787,258	28.27
Twitter/TW	41,652,230	1,468,365,182	70.51
SK-2005/SK05	50,636,154	1,810,063,330	71.49
Friendster/FR	65,608,366	1,806,067,135	55.06
UK-2006/UK06	92,734,067	2,797,759,396	60.34
UK-2007/UK07	109,499,800	3,448,528,200	62.99
UK-2014/UK14	787,801,471	47,614,527,250	120.88
Clueweb12/CW	978,409,098	42,574,107,469	87.03
GSH-2015/GSH	988,490,691	33,877,399,152	68.54

Vertex Insertion and Deletion. Our algorithms also support vertex insertion and deletion. When a vertex u is inserted, we first add u to the result MIS. That is, $u.\mathsf{in} \leftarrow true$. Then, we insert all edges incident to u as a batch update and run the algorithm. When a vertex u is deleted, we first run the batch update algorithm by deleting all edges incident to u. Then, we remove the vertex u.

VII. Experiments

Datasets. We use 16 real-world datasets [27], [28] categorized to small and large groups according to the dataset size. The dataset details are summarized in Table I. $|V|$ and $|E|$ represent the number of vertices and edges respectively. deg_{avg} denotes the average vertex degree in each dataset.

Experimental settings. Similar to [14], [15], we randomly insert and delete edges to simulate update operations. To evaluate the independent set size maintenance, we randomly delete k edges and insert them back in total of $2k$ update operations to test the result consistency. If not stated, we set $k = 50,000$. That is 100,000 updates consisting of 50,000 edge deletions and insertion of the same 50,000 edges. We conduct single update experiments where edge updates are processed one by one. We also conduct batch update experiments where each batch contains multiple edge update operations. We utilize b to represent batch size. $b = 1$ means single update. All experiments are conducted on a cluster of 10 machines connected by Gigabit Ethernet. Each machine has one 3.0GHz Intel Xeon E3-1120 CPU, 64GB DDR3 RAM and 1.1TB disk. Unless specified, we use 10 machines each with one core.

Algorithms. We involve existing single machine dynamic algorithms and distributed algorithms in our experiments. For single machine algorithms, DGOne and DGTwo are centralized dynamic algorithms employing degeneracy graph

TABLE II: Test of order independence.

Dataset	Response Time (seconds)		Communication Cost (MB)		Memory Cost (MB)		Superstep Number	
	DisMIS	OIMIS	DisMIS	OIMIS	DisMIS	OIMIS	DisMIS	OIMIS
SKI	0.94	0.33	92	46	47	44	13	11
TW	17.55	11.36	3183	1641	886	820	19	17
UK07	44.80	31.26	9335	4991	2367	2122	175	173
UK14	410.44	290.34	71649	40013	17107	15828	1723	1723
CW	459.62	330.40	86079	46222	21129	19450	733	733
GSH	477.89	340.60	93111	50449	22416	21153	97	95

TABLE III: Test of optimization techniques. +LR represents applying lower-ranking activation on OIMIS. +SS represents applying both lower-ranking and same-status activation on OIMIS.

Dataset	Response Time(seconds)			Active Vertex Number			Superstep Number			CommunicationCost(MB)			Memory Cost(MB)		
	OIMIS	+LR	+SS	OIMIS	+LR	+SS	OIMIS	+LR	+SS	OIMIS	+LR	+SS	OIMIS	+LR	+SS
SKI	0.38	9.82%	2.41%	3752337	39.00%	6.62%	12	12	11	48	1.63%	2.99%	48	11.98%	1.408%
TW	12.78	2.53%	8.91%	104298308	35.36%	11.36%	18	18	17	1705	0.15%	3.65%	926	11.12%	0.365%
UK07	35.31	7.12%	4.69%	275903847	37.96%	7.01%	174	174	173	5231	2.22%	2.43%	2568	17.33%	0.047%
UK14	373.11	15.96%	7.41%	2195311733	34.39%	10.04%	1723	1723	1723	42358	2.69%	2.93%	18882	16.26%	0.000%
CW	408.65	10.11%	10.06%	2521313719	30.94%	10.54%	733	733	733	48325	1.40%	2.99%	22516	13.61%	0.003%
GSH	369.30	4.31%	6.44%	2613212083	23.94%	12.21%	96	96	95	52558	1.75%	2.30%	23194	8.79%	0.005%

proposed in [15]. DOSwap, DTSwap, LazyDOSwap and LazyDTSwap are swap-based centralized dynamic algorithms proposed in [14]. For distributed algorithms, Naive recomputes the result on the updated graph for each batch update. SCALL scans all neighbors to decide the vertex status. DOIMIS is our implementation of Algorithm 3. DOIMIS+ applies the lower-ranking activation on DOIMIS. DOIMIS$*$ combines the same-status activation technique with DOIMIS+.

Metrics. We report the *independent set size* and *response time* to show the algorithm effectiveness and efficiency respectively. *Response time* is counted only for processing the update operations. Note that data loading and result dumping time are excluded. We also report *communication cost* and *memory cost* of to study the performance of compared algorithms. *Communication cost* is the sum of data size transferred among workers in the cluster while updating the results. *Memory cost* reports the highest memory consumption of each algorithm during the updating procedure.

A. Test of Order Independence

In this subsection, we first conduct tests to compare the performance of DisMIS and OIMIS. Note that, we implement both DisMIS and OIMIS on ScaleG. Thus, the activation are automatically handled by the system. The experimental results are presented in Table II. From the results, we can find that OIMIS runs faster than DisMIS over all datasets. Considering they output the same independent sets, this shows the out-performance of our OIMIS over DisMIS. We can also find that the communication cost of OIMIS is nearly half of that of DisMIS, which supports the high efficiency of OIMIS. The reason that DisMIS generates higher communication cost is that vertices in DisMIS have three states and DisMIS has more vertex values synced between different machines than that of OIMIS. With the design of OIMIS, vertices only have two status to synced. As expected, the supersteps of OIMIS are bounded by that of DisMIS. In some tests like SKI, CW and GSH, OIMIS costs fewer supersteps than DisMIS.

B. Test of Optimization Techniques

In this subsection, we evaluate the effect of our optimization techniques proposed in Section V in improving OIMIS's performance. The experimental results of running basic OIMIS, as well as equipped with lower-ranking activation denoted by +LR and same-status activation denoted by +SS on representative datasets are shown in Table III. It is clear that the response time decreases with the application of our selective activation techniques. Taking UK14 as an example, the lower-ranking activation reduces response time of OIMIS by 15.96% and the same-status activation further improves its efficiency by 7.41%. This is because lower-ranking and same-status save computation and communication cost as introduced in Section V. The reported results of active vertex number demonstrates the computation cost saved by our optimization techniques. For example, the lower-ranking activation reduces the number of active vertices of OIMIS on UK07 by 37.96% and same-status activation further reduces 7.91% active vertex number. The reported communication cost also validates the effectiveness of our two optimization techniques on reducing the communication cost. For example, the lower-ranking activation reduces the communication cost of OIMIS on UK14 by 2.69% and same-status activation further reduces 2.93% communication cost. Table III also shows the memory usages during the update process. As we can see from the results, the lower-ranking activation is more effective than same-status activation on reducing the memory cost. The reason is that lower-ranking activation reduces more active vertices than same-status activation. We also report the number of supersteps which is consistent with our analysis in Section V.

C. Test of Effectiveness

In this subsection, we evaluate the effectiveness of our algorithm maintaining the independent set size. Table IV gives the independent set size of our algorithm as well as

TABLE IV: Independent size comparison. $|MIS|$ denotes the independent set size. $prec$ denotes the percentage of the independent set size of DOIMIS divided by that of compared algorithm.

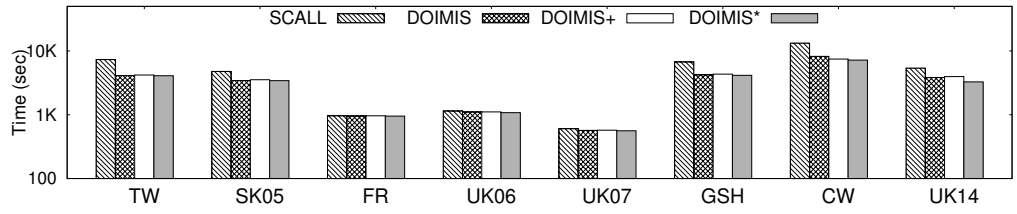
Dataset	DOIMIS	ARW		DGTwo		DTSwap		LazyDTSwap	
	$\|MIS\|$	$\|MIS\|$	$prec$	$\|MIS\|$	$prec$	$\|MIS\|$	$prec$	$\|MIS\|$	$prec$
SL	55976	56395	99.26%	56382	99.28%	56396	99.25%	56398	99.25%
AM	171362	174359	98.28%	174498	98.20%	174632	98.13%	174632	98.13%
GO	523125	528495	98.98%	527603	99.15%	529138	98.86%	529138	98.86%
DB	486736	487213	99.90%	487236	99.90%	487238	99.90%	487238	99.90%
WK	2337612	2338222	99.97%	2338222	99.97%	2338222	99.97%	2338222	99.97%
SKI	1140983	1169605	97.55%	1168413	97.65%	1170580	97.47%	1170580	97.47%
OR	774071	803511	96.34%	816184	94.84%	803557	96.33%	803545	96.33%
UK02	11651974	11860328	98.24%	11821051	98.57%	11860441	98.24%	11860441	98.24%
TW	28141905	28732311	97.95%	28745041	97.90%	28732316	97.95%	28732316	97.95%
SK05	28780235	30204534	95.28%	OOM	-	30204547	95.28%	30204547	95.28%
FR	35854798	36233627	98.95%	36301980	98.77%	36233656	98.95%	36233656	98.95%
UK06	56900969	58359527	97.50%	OOM	-	OOM	-	58359566	97.50%
UK07	67473042	69230655	97.46%	OOM	-	OOM	-	69230723	97.46%
UK14	413485972	OOM	-	OOM	-	OOM	-	OOM	-
CW	504445439	OOM	-	OOM	-	OOM	-	OOM	-
GSH	384244969	OOM	-	OOM	-	OOM	-	OOM	-
AVG			98.24%		98.42%		98.21%		98.10%

existing state-of-the-art centralized dynamic MIS algorithms including DGTwo, DTSwap and LazyDTSwap for 100,000 update operations. Note that the independent set size of DOIMIS are the same as that of OIMIS. DGOne, DOSwap and LazyDOSwap are omitted because their independent sets are smaller than that of DGTwo, DTSwap, LazyDTSwap respectively. We also report the static 1-swap independent set algorithm ARW [16] adopted by DTSwap. Beside the independent set size, Table IV also gives $prec$ which denotes the percentage of the independent set size of DOIMIS divided by that of compared algorithm. From the results, we can see that the independent set size of DOIMIS is all above 98% of that of existing static and dynamic algorithms. Please note that existing dynamic algorithms can't finish on large-scale graphs like UK06, UK07, UK14, CW and GSH. DGTwo can't finish on even smaller graph SK05. Even for LazyDTSwap with optimization technique, it still can't process the largest three graphs in our experiments. This is because centralized dynamic algorithms put the whole graph in one machine and employ sophisticated auxiliary data structures like the degeneracy graph in DGTwo and swap-based index in DTSwap, to help improving the result quality. However, these data structures can easily cause out of memory for large-scale graphs. These results prove the effectiveness of DOIMIS as a distributed algorithm dynamically maintaining independent set with good quality for large-scale graphs.

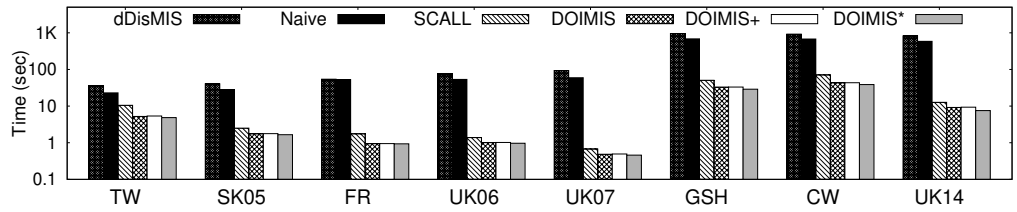
D. Test of Efficiency

In this subsection, we conduct experiments to show the efficiency of our algorithm in maintaining independent set size. We focus on large-scale graphs and compare our algorithms with distributed baselines. We also include dDisMIS which recomputes the result on the updated graph using DisMIS from scratch for each batch update. The response time for 100,000 updates processed one by one of all distributed algorithms are shown in Fig. 10(a). The algorithms show similar relative performance over all datasets. More specifically, all other algorithms outperform Naive and DisMIS because they compute from scratch for each update which is obviously inefficient. Note that the response time of Naive and dDisMIS for single update are omitted because none of them can finish in 24 hours. From the results, we can also find that SCALL is outperformed by DOIMIS, DOIMIS+ and DOIMIS$*$ because it causes more scanning cost. DOIMIS$*$ outperforms all distributed baseline algorithms over all datasets. This benefits from the computation and communication cost saved by lower-ranking and same-status activation. We also present the response time taken by all compared algorithms processing 100,000 updates in 2 batches (deletion and insertion batch) in Fig. 10(b). The algorithms show similar relative performances with single update.

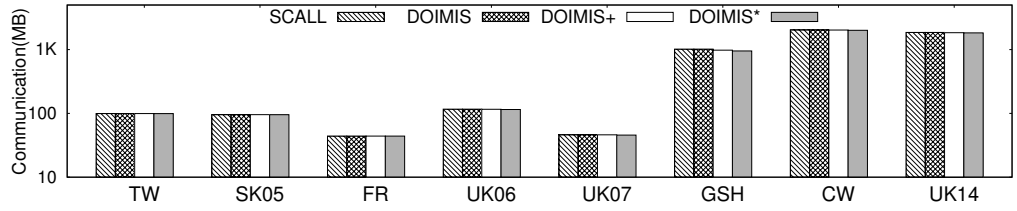
The communication cost of all compared algorithms are also provided in Fig. 10(c). From the results, we find that the communication cost of SCALL and DOIMIS are the same. The reason is that even though DOIMIS saves scanning cost compared with SCALL, the vertex status values in each superstep are the same. Thus, the number of transferred messages and activated vertices are the same. The selective activation optimization techniques are still effective in reducing the communication cost. However, they cause less communication cost reduction compared with our static algorithm. This is reasonable since dynamic experiments uses much less supersteps to finish compared with static experiments.



(a) Response time (b=1)

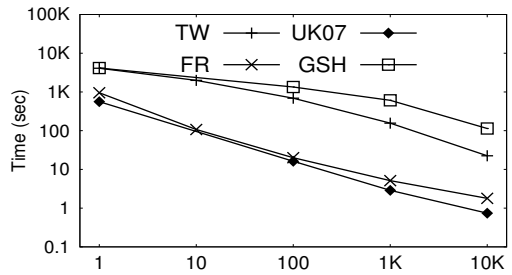


(b) Response time (b=50000)

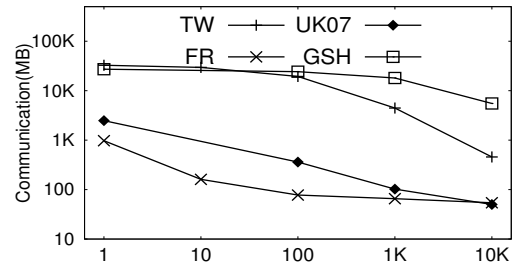


(c) Communication cost (b=50000)

Fig. 10: Test of efficiency (100,000 updates)



(a) Response time



(b) Communication cost

Fig. 11: Test of batch size.

E. Test of Batch Size

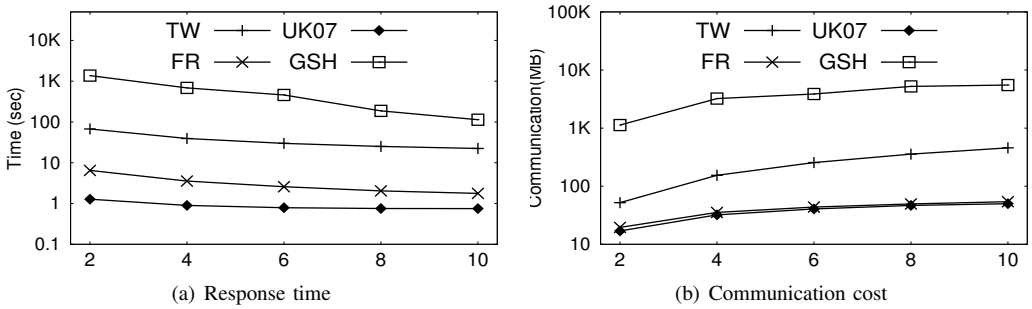
In this subsection, we study the performance of our algorithm DOIMIS$*$ equipped with all optimization techniques dealing with different number of update operations in each batch denoted by b. We report the response time and communication cost of DOIMIS$*$ when b is set as 1, 10, 100, 1,000 and 10,000 respectively. The results are presented in Fig. 11. The response time and communication cost both decrease with the increasing of batch size. This shows the effectiveness of our algorithm processing batch updates. Note that the independent set sizes are omitted since they are the same for different values of b.

F. Test of Scalability

In this subsection, we conduct experiments to test the scalability performance of our algorithm DOIMIS$*$.

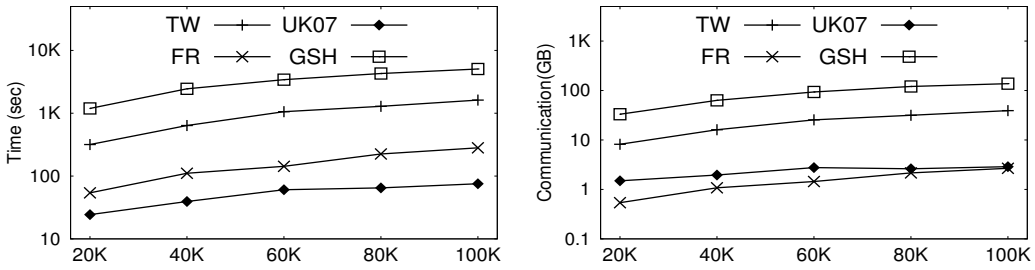
Varying the Number of Machines. We first evaluate the scalability of our algorithm by varying the number of machines denoted by $|W|$. 100,000 updates are processed in the batch size of 10,000. The experimental results are presented in Fig. 12. According to Fig. 12(a), it is clear that the response time decreases with the increasing of the machine number in the cluster over all datasets. For example, DOIMIS$*$ is two times faster when $|W| = 10$ compared with $|W| = 2$ on TW. Note that the improving ratio is not linear with the increasing of $|W|$. This is because more machines leads to more communication cost. Fig. 12(b) validates this fact. The communication cost of DOIMIS$*$ on TW for $|W| = 10$ is 8 times as that for $|W| = 2$.

Varying the Number of Update Operations. We also test the scalability of our algorithm by varying the number of



(a) Response time (b) Communication cost

Fig. 12: Scalability test: varying the number of machines.



(a) Response time (b) Communication cost

Fig. 13: Scalability test: varying the number of updates.

update operations denoted by $|U|$ from 200,000 to 1,000,000. The update operations are processed in the batch size of 1000. The experimental results are presented in Fig. 13. From the results in Fig. 13(a), we see that the response time increases with the growth of update operations. We also report the communication cost in Fig. 13(b). It is clear that the communication cost also increases with the increasing of $|U|$. These experimental results show the good scalability of our algorithm.

VIII. Related Work

A. Static MIS Algorithms

There have been many efforts on studying the static MIS (approximate) algorithms. A bunch of studies solve exact MIS problem due to its NP-hardness [29]–[32]. The branch-and-bound framework is widely used in these works with vertex reordering and pruning techniques. Much research has also been focusing on reducing the time complexity [30], [31]. In the meantime, computing the approximate MIS is also hard to have a constant approximation factor for general graphs [33], [34]. Therefore, many algorithms employ different heuristics to generate approximate MIS [16], [24], [35], [36]. Among them, Chang et al. [24] propose the state-of-the-art static approximated algorithm which iteratively applies reduction rules and remove highest degree vertex when reduction rules cannot be applied. [37], [38] focus on accelerating MIS computation on GPUs. [25] and [26] focus on studying the techniques to remove unpromising vertex from the results in distributed setting. [10], [11], [39] study the MIS problem on weighted graphs. Though effective and efficient for the above existing works, they assume that the graph is static which is usually not the case for real-life graphs. They can't be applied directly on dynamic graphs.

B. Dynamic MIS Algorithms

[13] is the first work studying dynamic MIS computation. However, the returned independent set quality depends on initial solution. In [15], Zheng et al. propose a degeneracy graph based updating algorithm that is insensitive to initial input. Batch updates are also supported in this work. Gao et al. propose a MIS maintenance framework with approximation guarantee in [14]. Swap-based index structure is designed to search for possible swaps to improve the independent set size. These works all assume that the input graph can fit in one machine which is not always true in real-world scenario. They can't be directly utilized to process large-scale graphs kept in distributed environments. Note that there is a serious of studies on dynamic maximal independent set adopting probabilistic model [40], [41]. However, they only focus on theoretical complexities and cannot guarantee deterministic results. To the best of our knowledge, this is the first work to maintain a good quality maximal independent set over dynamic graphs in a distributed environment with deterministic results.

IX. Conclusion

In this paper, we study the problem of near-maximum independent set maintenance for large dynamic graphs in distributed setting. We propose a novel framework to efficiently compute a near maximum independent set without any order dependency. We prove the effectiveness of our framework for dynamic MIS maintenance. Our framework generates a consistent independent set result regardless of the order of edge updates. We also propose several optimization techniques to further improve its efficiency. Extensive experiments over a wide range of real graphs verify the effectiveness and the efficiency of our algorithms on maintaining independent set.

REFERENCES

[1] F. Araújo, J. Farinha, P. Domingues, G. C. Silaghi, and D. Kondo, "A maximum independent set approach for collusion detection in voting pools," *J. Parallel Distributed Comput.*, vol. 71, no. 10, pp. 1356–1366, 2011.

[2] D. Puthal, S. Nepal, C. Paris, R. Ranjan, and J. Chen, "Efficient algorithms for social network coverage and reach," in *Proceedings of IEEE International Congress on Big Data*, 2015, pp. 467–474.

[3] T. Strijk, A. Verweij, and K. Aardal, "Algorithms for maximum independent set applied to map labelling," *Technical Report, Utrecht University*, 2000.

[4] A. W. Fu, H. Wu, J. Cheng, and R. C. Wong, "IS-LABEL: an independent-set based labeling scheme for point-to-point distance querying," *Proc. VLDB Endow.*, vol. 6, no. 6, pp. 457–468, 2013.

[5] M. Jiang, A. W. Fu, R. C. Wong, and Y. Xu, "Hop doubling label indexing for point-to-point distance querying on scale-free networks," *Proc. VLDB Endow.*, vol. 7, no. 12, pp. 1203–1214, 2014.

[6] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li, "New algorithms for fast discovery of association rules," in *Proceedings of KDD*, 1997, pp. 283–286.

[7] S. Butenko, P. M. Pardalos, I. Sergienko, V. Shylo, and P. Stetsyuk, "Finding maximum independent sets in graphs arising from coding theory," in *Proceedings of SAC*, 2002, pp. 542–546.

[8] J. M. Robson, "Algorithms for maximum independent sets," *J. Algorithms*, vol. 7, no. 3, pp. 425–440, 1986.

[9] P. Berman and M. Fürer, "Approximating maximum independent set in bounded degree graphs," in *Proceedings of SODA*, D. D. Sleator, Ed., 1994, pp. 365–371.

[10] C. Joo, X. Lin, J. Ryu, and N. B. Shroff, "Distributed greedy approximation to maximum weighted independent set for scheduling with fading channels," in *Proceedings of the fourteenth ACM international symposium on Mobile ad hoc networking and computing*, 2013, pp. 89–98.

[11] ——, "Distributed greedy approximation to maximum weighted independent set for scheduling with fading channels," *IEEE/ACM Transactions on Networking*, vol. 24, no. 3, pp. 1476–1488, 2015.

[12] R. Bar-Yehuda, K. Censor-Hillel, M. Ghaffari, and G. Schwartzman, "Distributed approximation of maximum independent set and maximum matching," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 2017, pp. 165–174.

[13] W. Zheng, Q. Wang, J. X. Yu, H. Cheng, and L. Zou, "Efficient computation of a near-maximum independent set over evolving graphs," in *Proceedings of ICDE*, 2018, pp. 869–880.

[14] X. Gao, J. Li, and D. Miao, "Dynamic approximate maximum independent set on massive graphs," in *Proceedings of ICDE*, 2022, pp. 1836–1848.

[15] W. Zheng, C. Piao, H. Cheng, and J. X. Yu, "Computing a near-maximum independent set in dynamic graphs," in *Proceedings of ICDE*, 2019, pp. 76–87.

[16] D. V. Andrade, M. G. Resende, and R. F. Werneck, "Fast local search for the maximum independent set problem," *Journal of Heuristics*, vol. 18, no. 4, pp. 525–547, 2012.

[17] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[18] W. Zheng, Q. Wang, J. X. Yu, H. Cheng, and L. Zou, "Efficient computation of a near-maximum independent set over evolving graphs," in *Proceedings of ICDE*, 2018, pp. 869–880.

[19] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of SIGMOD*, 2010, pp. 135–146.

[20] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of OSDI*, 2012, pp. 17–30.

[21] X. Wang, D. Wen, L. Qin, L. Chang, Y. Zhang, and W. Zhang, "Scaleg: A distributed disk-based system for vertex-centric graph processing," *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[22] D. Peleg, *Distributed computing: a locality-sensitive approach*, 2000.

[23] M. Luby, "A simple parallel algorithm for the maximal independent set problem," in *Proceedings of STOC*, 1985, pp. 1–10.

[24] L. Chang, W. Li, and W. Zhang, "Computing a near-maximum independent set in linear time by reducing-peeling," in *Proceedings of SIGMOD*, 2017, pp. 1181–1196.

[25] T. George, M. S. D. Hespe, and M. S. S. Lamm, "Distributed kernelization for independent sets," 2018.

[26] D. Hespe, C. Schulz, and D. Strash, "Scalable kernelization for maximum independent sets," *Journal of Experimental Algorithmics*, vol. 24, pp. 1–22, 2019.

[27] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[28] law.di.unimi.it, "The laboratory for web algorithmics," 2002. [Online]. Available: http://law.di.unimi.it/

[29] R. E. Tarjan and A. E. Trojanowski, "Finding a maximum independent set," *SIAM Journal on Computing*, vol. 6, no. 3, pp. 537–546, 1977.

[30] F. V. Fomin, F. Grandoni, and D. Kratsch, "A measure & conquer approach for the analysis of exact algorithms," *Journal of the ACM (JACM)*, vol. 56, no. 5, pp. 1–32, 2009.

[31] M. Xiao and H. Nagamochi, "Exact algorithms for maximum independent set," *Information and Computation*, vol. 255, pp. 126–146, 2017.

[32] T. Akiba and Y. Iwata, "Branch-and-reduce exponential/fpt algorithms in practice: A case study of vertex cover," *Theoretical Computer Science*, vol. 609, pp. 211–225, 2016.

[33] J. M. Robson, "Algorithms for maximum independent sets," *Journal of Algorithms*, vol. 7, no. 3, pp. 425–440, 1986.

[34] M. Halldórsson and J. Radhakrishnan, "Greed is good: Approximating independent sets in sparse and bounded-degree graphs," in *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, 1994, pp. 439–448.

[35] Y. Liu, J. Lu, H. Yang, X. Xiao, and Z. Wei, "Towards maximum independent sets on massive graphs," *Proceedings of the VLDB Endowment*, 2015.

[36] S. Lamm, P. Sanders, C. Schulz, D. Strash, and R. F. Werneck, "Finding near-optimal independent sets at scale," in *2016 Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2016, pp. 138–150.

[37] M. Burtscher, S. Devale, S. Azimi, J. Jaiganesh, and E. Powers, "A high-quality and fast maximal independent set implementation for gpus," *ACM Transactions on Parallel Computing*, vol. 5, no. 2, pp. 1–27, 2018.

[38] T. Imanaga, K. Nakano, R. Yasudo, Y. Ito, Y. Kawamata, R. Katsuki, Y. Tabata, T. Yazane, and K. Hamano, "Simple iterative trial search for the maximum independent set problem optimized for the gpus," *Concurrency and Computation: Practice and Experience*, p. e6681, 2021.

[39] J. Gu, W. Zheng, Y. Cai, and P. Peng, "Towards computing a near-maximum weighted independent set on massive graphs," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, 2021, pp. 467–477.

[40] S. Assadi, K. Onak, B. Schieber, and S. Solomon, "Fully dynamic maximal independent set with sublinear update time," in *Proceedings of the 50th Annual ACM SIGACT Symposium on theory of computing*, 2018, pp. 815–826.

[41] S. Behnezhad, M. Derakhshan, M. Hajiaghayi, C. Stein, and M. Sudan, "Fully dynamic maximal independent set with polylogarithmic update time," in *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2019, pp. 382–405.