2018 IEEE 34th International Conference on Data Engineering

# Efficient Computation of A Near-Maximum Independent Set Over Evolving Graphs

Weiguo Zheng [#1], Qichen Wang [*2], Jeffrey Xu Yu [#3], Hong Cheng [#4], Lei Zou [†5]

[#] *The Chinese University of Hong Kong*
[1,3,4] {wgzheng,yu,hcheng}@se.cuhk.edu.hk

[*] *Hong Kong University of Science and Technology*
[2] qwangbp@connect.ust.hk

[†] *Peking University*
[5] zoulei@pku.edu.cn

***Abstract*—Most existing algorithms computing the maximum independent set (MIS) or independent set (IS) are designed for handling static graphs, which may not be practicable as many networks are dynamically evolving over time. In this paper, we study the MIS/IS problem in evolving graphs by considering graph update operations: vertex/edge addition and vertex/edge deletion. Instead of computing the MIS/IS of the updated graph from scratch, we propose a baseline algorithm that finds the MIS/IS at time $t_{i+1}$ based on the MIS/IS at time $t_i$. Due to the hardness of computing an exact MIS, we develop an efficient constant-time algorithm *LSTwo* to return a high-quality (large-size) independent set. Then we design a lazy search algorithm which produces higher-quality independent sets. To improve the time efficiency further, we devise the conditional besieging and *k-petal* based methods to reduce the search space. Extensive experimental studies over large-scale graphs confirm the effectiveness and efficiency of our proposed algorithms.**

## I. Introduction

Graphs have been widely used to model a vast variety of real-world data, such as social networks, biological networks, and knowledge graphs. To manage and analyze graphs efficiently, a lot of problems, e.g., reachability query [1], shortest path computation [2], and subgraph search [3] have been extensively studied. As one of the fundamental problems, the *maximum independent set* (*MIS*) problem is a classic NP-hard problem [4]. It takes an undirected graph $G = (V_G, E_G)$ as inputs and tries to find the largest independent set, where an independent set is subset of vertices $S \subseteq V_G$ such that each two vertices in $S$ have no edges. Actually, the *MIS* problem is closely related to two well-known optimization problems, i.e., the minimum vertex cover problem and the maximum clique problem [5]. Besides the significance in theory, the *MIS* problem has a wide range of applications spanning many fileds, such as collusion detection [6], computer graphics [7], automated map labeling [8], and road network routing [9].

### A. Previous Work On MIS

***Exact Algorithms.*** The state-of-the-art algorithms follow the branch-and-bound framework [10], [11]. In the computation process, reduction rules can be applied to reduce the search space by removing some vertices that are (or not) definitely contained in a certain *MIS* [12], [13]. Thus the graph size can be reduced greatly while preserving the correctness of the results. Fomin et al. propose a measure and conquer approach with the time complexity $O(1.2201^n)$, where $n = |V_G|$ is the number of vertices [11]. By introducing a new branching rule, called "branching on edges", Xiao et al. design an $O(1.1996^n n^{O(1)})$-time polynomial-space algorithm [14]. Recently, Akiba et al. present a branch-and-reduce algorithm for the vertex cover problem[1] using the techniques developed for theoretical algorithms [10]. Due to the exponential time complexity, they are quite inefficient to handle large graphs.

***Greedy Algorithms.*** Håstad has shown that it is NP-hard to compute the *MIS* with the approximation ratio $n^{1-\epsilon}$ or less, where $0 < \epsilon < 1$ [15]. In contrast to the exact algorithms, there are a bunch of approximate algorithms that employ some heuristics and local search techniques to compute a high-quality (not necessary to be the *MIS*) independent set [16], [17], [18]. Andrade et al. introduce two local search routines to improve the solution quality [5]. By using a preliminary solution of the evolutionary algorithm, Lamm et al. repeatedly kernelize the graph until a large independent set is found [19]. In addition to applying reductions on the fly, Dahlum et al. cut a few high-degree vertices during local search [13]. Liu et al. study the *MIS* problem under the semi-external setting and devise a general vertex-swap framework to incrementally increase the size of independent sets [20]. Recently, Li et al. develop a Reducing-Peeling framework which iteratively applies reduction rules on vertices (Reducing) and removes the vertex with the largest degree (Peeling) [21].

### B. MIS Over Evolving Graphs

Although there have been lots of methods designed for computing the (approximate) *MIS*, they demand that the input graph $G$ is static without any changes. However, this requirement is not practicable in many real applications since these graphs are usually changing continuously.

***Applications***. The clique is a foundational idea for studying cohesive subgraphs in social networks [22]. However, the structure may change over time as a user can add or remove

[1] $V_G/S$ is an *MIS* if the vertex set $S$ is a minimum vertex cover.

2375-026X/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDE.2018.00083

edges between his neighbors and himself. As discussed above, computing cliques equals computing independent sets in the complementary graph. Hence, we can report cliques in real time by maintaining independent sets in the complement of the input graph. In parallel computation, the subtasks assigned on each processing element should be independent with each other so that they can be executed simultaneously to maximize the parallel computing ability. Actually, as some subtasks finish the dependency relations will change, which demands recomputation of the dependency relations to enhance the computing ability. Moreover, the stock price data over a certain period of time can be modeled as a market graph, where the vertex represents the financial instruments (i.e., stocks) and an edge connects two stocks if the cross-correlation between their price fluctuations exceeds a specified threshold [23], [24]. Finding a completely diversified portfolio can be done by finding the maximum independent set in the market graph [25]. Since the "correlations" may change over time, providing the real-time independent sets is useful for investors.

Clearly, studying the *MIS/IS* problem over evolving graphs is very interesting and has high practical significance. To the best of our knowledge, we are the first to study the *MIS* (and near-*MIS*) problem over evolving graphs by considering four graph update operations, i.e., vertex/edge additions and vertex/edge removals. A straightforward method is to compute the *MIS/IS* once the graph changes by using existing approaches. It is obvious that this method is extremely inefficient since each time the *MIS/IS* is calculated from scratch. In particular, most modern graphs have large numbers of vertices and edges, which increases the time cost significantly.

***Challenges and Our Approaches***. Firstly, there is no local property for the evolving MIS. It is NP-hard to compute $MIS(G_{i+1})$ based on $MIS(G_i)$ (as shown in Theorem 2.1), where $MIS(G_i)$ denotes the *MIS* of graph $G$ at time $t_i$. We develop an exact algorithm, which constructs $MIS(G_{i+1})$ by finding a set of non-*MIS* vertices $S$ to replace a set of *MIS* vertices $T$, where $|S| \geq |T|$. Unlike existing swapping algorithms (e.g., [26] and [20]), the exploration starts from the updated vertices or edges based on $MIS(G_i)$.

Secondly, while the exact algorithm proposed above can find the optimal solution, it suffers from high time complexity. Thus a critical issue is to strive for the nontrivial tradeoff between the accuracy and efficiency. To this end, we devise an algorithm *LSTwo* which only considers the vertices within 2-hop neighbors of the updated vertex or the incident vertices of an updated edge. In order to improve the solution quality, we should not restrict the depth of the search but abandon some unpromising vertices. Then we propose an efficient linear algorithm, called *lazy search*, which performs the search in the way of interleaving depth-first-search and breadth-first-search. It conducts the pruning according to the searching records, and achieves better performance (takes similar time as *LSTwo* but computes larger independent sets). To further improve the time efficiency, we propose the conditional besieging and *k-petal* based approaches following the intuitions that conditional besieging is transitive and the high-degree vertices are less likely to be included in a maximum independent set [13].

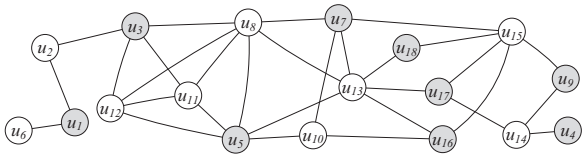



Fig. 1. A running example.

***Contributions***. We summarize the contributions as follows.

- To the best of our knowledge, we are the first to study the *MIS* problem in evolving graphs, and develop an exact algorithm to solve the problem, which does not compute the *MIS* from scratch.
- Instead of computing the exact solution, we propose a very efficient constant-time algorithm, *LSTwo*, to return the high-quality independent sets.
- We design a lazy search algorithm to improve the solution quality, which performs the pruning according to the searching records. We also devise the conditional besieging and *k-petal* based methods to further reduce the time cost.
- We evaluate the proposed methods by conducting extensively experimental evaluations on large-scale graphs.

## II. Problem Definition

In this section, we formally define the problem. For simplicity, we focus on *unweighted undirected graphs*.

*Definition 2.1:* (Independent Set). Given a graph $G$, its independent set, denoted by $IS(G)$, is a set of vertices in which any two vertices have no edge in $G$.

*Definition 2.2:* (Maximum Independent Set). Given a graph $G$, its maximum independent set, denoted by $MIS(G)$, is the largest independent set of $G$.

Figure 1 presents a running example, where the grey vertices constitute the $MIS$. In an evolving graph, there are 4 update operations, i.e., vertex addition, vertex deletion, edge addition, and edge deletion. In this paper, we assume that there is only one update operation over graph $G_i$ at time $t_{i+1}$. It is NP-hard to compute $MIS(G_{i+1})$ based on $MIS(G_i)$ when there is an update operation $o_i$ over $G_i$ at time $t_{i+1}$.

*Theorem 2.1:* Given a maximum independent set $MIS(G_i)$ at time $t_i$ and an update operation $o_i$ over $G_i$ at time $t_{i+1}$, computing the $MIS(G_{i+1})$ of $G_{i+1}$ at time $t_{i+1}$ is NP-hard.

*Proof:* Let us consider the update operations vertex and edge additions. Case 1: The update operation $o_i$ is adding a vertex $v$. We can add $v$ into $MIS(G_i)$ directly. Thus, Case 1 can be computed in $O(1)$. Case 2: the update operation $o_i$ is adding an edge. The traditional $MIS$ computation problem, i.e., compute the $MIS(G)$ of a graph $G$, can be reduced to our problem. To compute the $MIS$ of a graph $G$, we first consider a special graph $G_0$ that consists of all vertices in $G$ without any edges. Obviously, $V(G_0)$ is the $MIS$ of $G_0$. Then we add the edges $E(G)$ into $G_0$ one by one. The process of adding an edge to $G_i$ is an instance of our problem. If our problem can be solved in polynomial time, the $MIS(G)$ of graph $G$ can be computed in polynomial time as well, which contradicts the NP-hardness of computing $MIS(G)$ for $G$. ■

Computing a high-quality independent set (not necessary to be the maximum one) in real applications is acceptable and interesting. Therefore, the task in this paper is as follows.

**Problem Statement**. *Given an independent set $IS(G_i)$ of graph $G_i$ at time $t_i$ and an update operation $o_i$ over $G_i$ at time $t_{i+1}$, the goal is to compute a high-quality maximal independent set $IS(G_{i+1})$ of $G_{i+1}$ at time $t_{i+1}$.*

Note that the input independent set $IS(G_i)$ can be the largest or not, the goal is to deliver a high-quality solution.

## III. Overview

In this section, we present the basic ideas of our approachs.

### A. Exact Algorithm

The rationale of the method is trying to add some *non-MIS* vertices into the independent set when there is an update operation. A natural way is to perform the depth-first-search to explore the graph. For example, to determine whether a vertex $u \in MIS$ is removable, we check whether one of its neighbors $v$ is insertable, which depends on $v$'s unvisited $MIS$ neighbors.

Hence, there is *a state expanding tree* in the searching process, where (1) The root is a vertex in $MIS$ or non-$MIS$; (2) If the root or intermediate node is an $MIS$ vertex $u$, its children are $u$'s neighbors excluding $u$'s ancestors and the vertices that are adjacent to $u$'s non-$MIS$ ancestors; (3) If the root or intermediate node is a non-$MIS$ vertex $v$, its children are $v$'s neighbors that are in $MIS$ excluding $v$'s ancestors, the siblings of $v$'s $MIS$ ancestors, and the vertices that are certain to be visited through $v$'s other $MIS$ neighbors.

If we explore the state expanding tree exhaustively, we can obtain the correct conclusion that $v$ (resp. $u$) is insertable (resp. removable) or not, which will lead to an exact solution.

### B. An Efficient Greedy Algorithm

The process above returns the exact solution, but it suffers from high time complexity. To guarantee the efficiency, we propose an efficient greedy algorithm that constrains the depth $h$ of search. Generally, we can derive the lower bound and upper bound for $MIS(G_{i+1})$ as presented in Section V-B. By imposing the constraint of $h$, we can examine the vertices within $h$-hop elegantly for a specified $h$. In other words, we can distinguish the cases that do not affect the *MIS* size (more details can be found in Section V-A).

Let us consider the update operation vertex deletion. When $h = 1$, only the direct neighbors of the involved vertices of the update operation can be considered. It leads to very limited improvement on solution quality. Figure 2 presents the structure $g_1$ that contributes to an accurate solution, where $u_1 \in MIS$ is the vertex to delete and $v_1 \notin MIS$ has only one neighbor. If $h = 2$, there are three cases $g_1$, $g_2$, and $g_3$ as shown in Figure 2, where the grey vertices and white vertices represent *MIS* and *non-MIS* vertices, respectively. The structure $g_2$ represents only one of the sets of structures in which $v_1$ has no neighbors in *MIS*. Both $g_1$ and $g_2$ contribute to an accurate solution. Besides structures $g_1$, $g_2$, and $g_3$, there is a structure $g_4$ if $h = 3$, where $v_2$ is a vertex of degree 1. Actually,

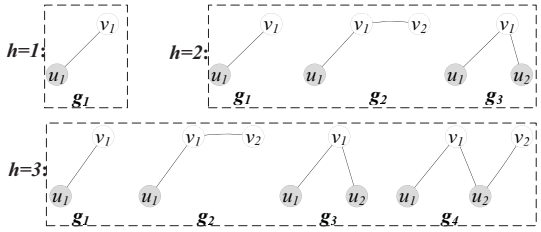



Fig. 2. Structures corresponding to different $h$ for vertex deletion.

$g_4$ comprises $g_3$ and $g_1$. Let $Pr(g_i)$ denote the probability of a structure occurs. Then we have $Pr(g_4) = Pr(g_3) * Pr(g_1)$. We can compute the probability $Pr(h = i)$ of finding the exact solutions for a given value $h$. Specifically, we have $Pr(h = 1) = Pr(g_1)$, $Pr(h = 2) = Pr(g_1) + Pr(g_2)$, and $Pr(h = 3) = Pr(g_1) + Pr(g_2) + Pr(g_3) * Pr(g_1)$. Clearly, the improvement by $Pr(h = 3)$ over $Pr(h = 2)$ is little compared with the improvement by $Pr(h = 2)$ over $Pr(h = 1)$.

Similarly, we can also enumerate the cases for adding and removing edges. We omit the details due to the space limitation. In summary, it is obvious that the improvement of $h = 3$ is marginal, but the time cost increases greatly, which is also confirmed in our experiments. Hence, we set $h$ to 2 in this paper.

### C. Lazy Search

The greedy algorithm above tries to exhaust all cases by restricting the depth of search, which is very efficient but may degrade the solution quality. Different from that, we propose a novel search strategy that does not constrain the depth of search. In order to guarantee the time efficiency, we design a systematic method to prune the search space, that is, determining whether a vertex should be explored.

The proposed lazy search performs the search in the fashion of interleaving depth-first-search and breadth-first-search. Notice that there is an observation: if a vertex $v \notin MIS$ is not insertable in the search, it is very likely to find an exact solution without considering $v$ in the latter search. The reasons lie in two-fold: First, the vertex $v$ is indeed not insertable. Second, we can find other solutions regardless of vertex $v$. We conduct empirically studies over several real datasets (e.g., GrQc, AstroPh, and BerkStan). The average precision is 99.89%, which indicates that the observation is reasonable in real data. Therefore, the basic idea of lazy search is: If a solution is found in the search, an exact *MIS* is delivered. Otherwise, we invalidate some vertices that will not be considered in the next search.

Beyond that, we also devise two pruning techniques, i.e., the conditional besieging and petal based pruning. Since we determine whether a vertex can be added into or removed from *MIS* according to whether it is besieged or not in the search tree, we can compute the conditional besieging status of vertices in the off-line phase, based on which it is easy to determine the besieging status of a vertex $v$ without exploring $v$ and its neighbors. Moreover, we design the petal based pruning. We know that high-degree vertices are less likely to

**Algorithm 1** *IsRemovable*($MIS(G_0), Out, In, u$)

**Input:** *Out*: the vertex set to be swapped out, *In*: the vertex set to be swapped in, the vertex $u \in MIS$ to be removed;
**Output:** true or false.

```
1: for each unvisited vertex v ∈ Nei(u) do
2:    X ← Nei(v) ∩ MIS(G0)/(Out ∪ {u})
3:    if X = ∅ then
4:       Out ← Out ∪ {u}, In ← In ∪ {v}
5:       return true
6:    W ← ∅, Y ← Nei(v) ∩ non-MIS(G0)/(In ∪ {v})
7:    for each unvisited vertex y ∈ Y do
8:       set y visited, W ← W ∪ {y}
9:    for each unvisited vertex w ∈ X do
10:      set w visited, W ← W ∪ {w}
11:      α ← ExactReVertex(Out ∪ {u} ∪ X, In ∪ {v}, w)
12:      if α = false then
13:         recover the visiting status of each vertex in W
14:         turn to line 1
15:   return true
16: return false
```

be in a maximum independent set [13]. Following the similar intuition, we find two sets of vertices, i.e., petal and pistil, based on which some vertices can be filtered out in the search.

## IV. An Exact Algorithm

In this section, we propose an algorithm to compute the exact $MIS(G_{i+1})$. The main idea is trying to add some *non-MIS* vertices into the independent set if possible.

### A. Determination of Insertable or Removable

The basic task is to determine whether a vertex $v \in$ *non-MIS* can be inserted into *MIS* and a vertex $u \in MIS$ can be removed from *MIS*. We introduce two definitions as follows.

*Definition 4.1:* (Vertex Removable). Given a set of vertices $S \subseteq$ *non-MIS*, a vertex $u \in MIS$ is $S$-removable if there is a vertex $v \in Nei(u)/S$ such that $v$ is $\{u\}$-insertable.

*Definition 4.2:* (Vertex Insertable). Given a set of vertices $S \subseteq MIS$, a vertex $v \in$ *non-MIS* is $S$-insertable if $v$ has no neighbor in the set $MIS/S$, i.e., $Nei(v) \cap MIS/S = \emptyset$, or each vertex $u' \in Nei(v) \cap MIS/S$ is $\{v\}$-removable.

Algorithm 1 presents the details of determining whether a vertex $u$ is removable from $MIS(G_0)$. Basically, if one of the two cases is satisfied $u$ is removable: (1) $v$ has no unvisited neighbors in *MIS* (lines 3-5); (2) all the unvisited neighbors of $v$ in *MIS* are removable (lines 9-15). Based on Algorithm 1, it is easy to determine whether a vertex $v$ is insertable as shown in Algorithm 2.

As shown in Algorithm 1, to determine whether a vertex $u$ is removable, there is a state expanding tree in the searching process. Thus we can perform the following pruning.

*Lemma 4.1:* If a vertex $u_1 \in MIS(G)$ is not removable in the current branch of search, it is not necessary to consider $u_1$ in the subtrees that are rooted at $u_1$'s siblings.

*Proof:* Assume $u_2$ is one sibling of $u_1$. Provided that we can find a solution with the subtree $ST_2$ rooted at $u_2$ and $ST_2$ contains $u_1$. Then we can also find the corresponding solution with the subtree $ST_1$ rooted at $u_1$ since $u_1$ and $u_2$ share the same ancestors. ■

**Algorithm 2** *IsInsertable*($MIS(G_0), Out, In, v$)

**Input:** *Out*: the vertex set to be swapped out, *In*: the vertex set to be swapped in, the *non-MIS* vertex $v$ to be inserted;
**Output:** true or false.

```
1: X ← Nei(v) ∩ MIS(G0)/Out
2: if X = ∅ then
3:    return true
4: for each unvisited vertex u ∈ X do
5:    α ← ExactReVertex(Out ∪ X, In ∪ {v}, u)
6:    if α = false then
7:       return false
8: return true
```

### B. Exact MIS Update

By using the two procedures above to determine whether a vertex is removable or insertable, we can compute the *MIS*.

*1) Vertex Deletion:* If the vertex $u$ to be deleted belongs to *non-MIS*, $MIS(G_1)$ equals $MIS(G_0)$. Otherwise, we need to determine whether $u$ is removable by exploiting Algorithm 1. If yes, we can obtain the updated $MIS(G_1) = MIS(G_0)/Out \cup In$. If $u$ is not removable, it indicates that there is no $MIS(G_0)'$ such that $u \notin MIS(G_0)'$ and $|MIS(G_0)| = |MIS(G_0)'|$. Therefore, we can just delete $u$ from $MIS(G_0)$ to obtain $MIS(G_1)$.

*2) Edge Addition* $(u, v)$*:* If at least one of vertices $u$ and $v$ does not belong to *MIS*, the size of $MIS(G_1)$ will not change. Hence, we just consider the case that both $u$ and $v$ belong to $MIS(G_0)$. To compute $MIS(G_1)$, we can check $u$ and $v$ separately by invoking the procedure (i.e., Algorithm 1). If either $u$ or $v$ is removable, we can obtain $MIS(G_1) = MIS(G_0)/Out \cup In$. Otherwise, we just need to remove $u$ or $v$ from $MIS(G_0)$.

*3) Edge Deletion:* To remove an edge $(u, v)$, there are two cases to deal with, i.e., (1) $u \in MIS(G_0)$, $v \notin MIS(G_0)$: We need to check whether $v$ is insertable using Algorithm 2. If $v$ is insertable, we can obtain the updated $MIS(G_1) = MIS(G_0)/Out \cup In$. Otherwise, $MIS(G_1) = MIS(G_0)$. (2) $u \notin MIS(G_0)$, $v \notin MIS(G_0)$: we can check whether $u$ and $v$ are insertable by invoking Algorithm 2. If both $u$ and $v$ are insertable, we can obtain $MIS(G_1)$ of size $|MIS(G_0)| + 1$.

*Lemma 4.2:* Given the edge $(u, v)$ to be removed ($u, v \notin MIS(G_0)$), if $u$ is insertable, $v$ must be insertable as well.

*Proof:* Assume that $u$ is insertable and $v$ is not insertable. In the original graph $G_0$, we obtain a larger independent set $MIS(G_0)' = MIS(G_0)/Out \cup In \cup \{u\}$. It obviously contradicts that $MIS(G_0)$ is the maximum independent set. ■

Similarly, if $u$ is not insertable, $v$ cannot be insertable either. Lemma 4.2 indicates that if $|MIS(G_1)| = |MIS(G_0)|+1$, $u$ and $v$ must be swapped into $MIS(G_1)$ together. Hence, it only needs to check one of the two vertices $u$ and $v$.

**Time Complexity.** Clearly, Algorithm 1 dominates the process of handling vertex deletion and edge addition. Since the exploration proceeds in a recursive manner, the time complexity is $O(d^{|V(G)|})$ in the worst case, where $d$ is the average vertex degree. To deal with edge deletion, Algorithm 2 is invoked, which may invoke Algorithm 1 for each vertex in $X$. Therefore, the worst time complexity of Algorithm 2 is $O(d^{|V(G)|+1})$.

**Algorithm 3** *RemoveVertex*$(G_0, MIS(G_0), v)$

**Input:** $G_0$, $MIS(G_0)$, and a vertex $v$ removed from $G_0$;
**Output:** An independent set $M_B(G_1)$ over $G_1$.

```
1: if v ∈ MIS(G_0) then
2:     MIS(G_0) ← MIS(G_0)/{v} // case 1.1
3:     for each vertex u ∈ Nei(v) do
4:         if Nei(u) ∩ MIS(G_0) = ∅ then
5:             M_B(G_1) ← MIS(G_0) ∪ {u} // case 1.2
6:             return M_B(G_1)
7:     M_B(G_1) ← MIS(G_0)
8: else
9:     M_B(G_1) ← MIS(G_0) // case 2
10: return M_B(G_1)
```

## V. An Efficient Greedy Algorithm

Let $M_B(G_1)$ denote an independent set that is returned by our algorithm. If the update operation is adding a vertex $v$, we have $M_B(G_1) = MIS(G_0) \cup \{v\}$ since $v$ is an isolated vertex.

### A. LSTwo

The basic idea of the greedy algorithm is restricting the depth of search to $h$. As discussed in Section III-B, we set $h$ to 2 to balance the quality and time efficiency. Before going to the details of LSTwo, we first distinguish the cases where the update operation does not affect the *MIS* size, i.e., $|MIS(G_{i+1})| = |MIS(G_i)|$. Given two graphs $G_1$ and $G_2$, $G_1$ is an ***induced subgraph*** of $G_2$ if $V(G_1) \subseteq V(G_2)$, $E(G_1) \subseteq E(G_2)$, and $(u, v) \in E(G_1)$ if $(u, v) \in E(G_2)$. We say $G_1$ is a ***spanning subgraph*** of $G_2$ if $V(G_1) = V(G_2)$ and $E(G_1) \subseteq E(G_2)$.

*Lemma 5.1:* If $G_1$ is an induced subgraph of $G_2$ or $G_2$ is a spanning subgraph of $G_1$, it holds that $|MIS(G_1)| \leq |MIS(G_2)|$.

*Proof:* Actually, any *MIS* of $G_1$ is an independent set of $G_2$, which can be proved by contradiction. If $MIS(G_1)$ is not an independent set of $G_2$, there must be an edge between $u$ and $v$ such that it does not belong to $G_2$, i.e., $(u, v) \notin E(G_2)$, which contradicts the premise $E(G_1) \subseteq E(G_2)$. ■

According to Lemma 5.1, we can conclude that deleting a *non-MIS* vertex $u$ from $G_0$ will not change the *MIS* since $G_1$ is an induced subgraph of $G_0$ when $u$ is deleted. Similarly, if at least one of $u$ and $v$ belongs to *non-MIS* and $(u, v)$ is a newly added edge to $G_0$, it holds that $MIS(G_1) = MIS(G_0)$.

***Vertex Deletion.*** There are two cases: $v \in MIS(G_0)$ and $v \notin MIS(G_0)$ as shown in Algorithm 3. If $v$ belongs to $MIS(G_0)$, we check each of its neighbors $u$ to see if $u$ has neighbors in $MIS(G_0)$. If yes, vertex $u$ is added to $M_B(G_1)$ (lines 1-6). Otherwise, $M_B(G_1)$ is returned just by deleting $v$.

***Edge Addition.*** Algorithm 4 gives the details to deal with edge addition. Since case 1 does not affect the *MIS* size as discussed above, we just need to handle the case that both $u$ and $v$ belong to the $MIS(G_0)$ (i.e., case 2). We examine the neighbors of $u$ and $v$. If $u' \in Nei(u)$ has only one neighbor $u$ in $MIS(G_0)$ or $v' \in Nei(v)$ has only one neighbor $v$ in $MIS(G_0)$, we can obtain $M_B(G_1)$ by removing $u$ (resp. $v$) and including $u'$ (resp. $v'$) as shown in lines 3-7. Beyond that, if there are two *non-MIS* vertices $u'$ and $v'$ whose neighbors in $MIS(G_0)$ are the same set $\{u, v\}$ and $u'$ is not incident to $v'$, an exact solution will be delivered (lines 8-9).

**Algorithm 4** *AddEdge*$(G_0, MIS(G_0), u, v)$

**Input:** $G_0$, $MIS(G_0)$, and an edge $(u, v)$ added into $G_0$;
**Output:** An independent set $M_B(G_1)$ over $G_1$.

```
1: if u ∉ MIS(G_0) or v ∉ MIS(G_0) then
2:     M_B(G_1) ← MIS(G_0) // case 1
3: else if u ∈ MIS(G_0) ∧ v ∈ MIS(G_0) then
4:     if there is u' ∈ Nei(u) ∧ Nei(u') ∩ MIS(G_0) = {u} then
5:         M_B(G_1) ← MIS(G_0))/{u} ∪ {u'} // case 2.1
6:     else if there is v' ∈ Nei(v) ∧ Nei(v') ∩ MIS(G_0) = {v} then
7:         M_B(G_1) ← MIS(G_0)/{v} ∪ {v'} // case 2.1
8:     else if there are two vertices u' and v' such that Nei(u') ∩
       MIS(G_0) = Nei(v') ∩ MIS(G_0) = {u, v} ∧ u' ∉ Nei(v') then
9:         M_B(G_1) ← MIS(G_0)/{u, v} ∪ {u', v'} // case 2.2
10:    else
11:        M_B(G_1) ← MIS(G_0)/{u} // case 2.3
12: return M_B(G_1)
```

***Edge Deletion.*** When the two incident vertices are both *non-MIS* vertices and their *MIS* neighbor is the same vertex $u'$, we can obtain a larger $M_B(G_1)$ by replacing $u'$ with $u$ and $v$ (lines 2-3 in Algorithm 5). When $u$ is an *MIS* vertex and $v$ is a *non-MIS* vertex, an exact solution can be returned if $u$ is the only *MIS* neighbor vertex of $v$ (lines 7-8). Moreover, if each *MIS* neighbor of $v$ has a 1-degree neighbor, the set $MIS(G_0) \cup \{v\} \cup S/T$ will be delivered, where $T = Nei(v) \cap MIS(G_0)$ and $S$ is the set of 1-degree neighbors of $T$ (lines 9-10).

**Time complexity**. As shown in Algorithms 3-5, it may need to examine 2-hop neighborhoods of the incident vertices. Therefore, the time complexity is $O(d^2)$, which indicates that time complexity of LSTwo is $O(d^2)$.

### B. Tightness Discussion

*Theorem 5.1:* Given an update operation from $G_0$ to $G_1$, it holds that $0 \leq |MIS(G_1)| - |M_B(G_1)| \leq 1$.

*Proof:* *Vertex deletion*: According to Algorithm 3, $|M_B(G_1)| = |MIS(G_0)|$ or $|MIS(G_0)| - 1$. If $v$ exists in all $MIS(G_0)$, it holds that $M_B(G_1) = MIS(G_1)$; Otherwise, $|MIS(G_1)| = |MIS(G_0)|$. Hence, we have $|MIS(G_1)| = |M_B(G_1)|$ or $|MIS(G_1)| = |M_B(G_1)| + 1$.

*Edge addition*: if at least one of $u$ and $v$ belongs to *non-MIS*, it holds that $MIS(G_1) = MIS(G_0)$ based on Lemma 5.1.

**Algorithm 5** *RemoveEdge*$(G_0, MIS(G_0), u, v)$

**Input:** $G_0$, $MIS(G_0)$ and an edge $(u, v)$ removed from $G_0$;
**Output:** An independent set $M_B(G_1)$ over $G_1$.

```
1: if u ∉ MIS(G_0) ∧ v ∉ MIS(G_0) then
2:     if {u'} = Nei(u) ∩ MIS(G_0) = Nei(v) ∩ MIS(G_0) then
3:         M_B(G_1) ← MIS(G_0)/{u'} ∪ {u, v} // case 1.1
4:     else
5:         M_B(G_1) ← MIS(G_0) //case 1.2
6: else if u ∈ MIS(G_0) ∧ v ∉ MIS(G_0) then
7:     if Nei(v) ∩ MIS(G_0) = {u} then
8:         M_B(G_1) ← MIS(G_0) ∪ {v} // case 2.1
9:     else if each MIS neighbor of v has a 1-degree neighbor then
10:        M_B(G_1) ← MIS(G_0) ∪ {v} ∪ S/T // case 2.2
11:    else
12:        M_B(G_1) ← MIS(G_0) // case 2.3
13: return M_B(G_1)
```

When both $u$ and $v$ belong to $MIS$: If $u$ and $v$ belong to all maximum independent sets of $G_0$, it holds that $|MIS(G_1)| = |MIS(G_0)| - 1$. Otherwise, $|MIS(G_1)| = |MIS(G_0)|$.

*Edge deletion*: Adding an edge between $u$ and $v$ over $G_1$ will lead to $G_0$. Based on the conclusion of edge addition, we have $|MIS(G_1)| - 1 \leq |MIS(G_0)| \leq |MIS(G_1)|$. Therefore, $|MIS(G_0)| \leq |MIS(G_1)| \leq |MIS(G_0)| + 1$. Since $|M_B(G_1)| = |MIS(G_0)|$ or $|MIS(G_0)| + 1$, the proof is achieved. ■

Table I presents the summary of all cases in LSTwo, where $\delta$ is the gap between the output independent set and the maximum independent set, i.e., $\delta = |MIS(G_1)| - |M_B(G_1)|$.

Actually, LSTwo works well even if the initial independent set is not the maximum. Let $M_A(G_0)$ denote the independent set computed by an approxiamte algorithm $M_A$. We can replace $MIS(G_0)$ in Algorithms 3-5 with $M_A(G_0)$.

*Theorem 5.2:* If $0 \leq |MIS(G_0)| - |M_A(G_0)| \leq k$, it holds that $0 \leq |MIS(G_1)| - |M_B(G_1)| \leq k + 1$.

*Proof:* (1). Add a vertex $v$: Since $MIS(G_1) = |MIS(G_0)| + 1$ and $M_B(G_1) = M_A(G_0) + 1$, it is clear that $0 \leq |MIS(G_1)| - |M_B(G_1)| \leq k$. (2). Remove a vertex $v$ or add an edge $(u, v)$: We have $|M_B(G_1)| = |M_A(G_0)|$ or $|M_A(G_0)| - 1$. According to Theorems 5.1, it holds that $|MIS(G_1)| = |MIS(G_0)|$ or $|MIS(G_1)| = |MIS(G_0)| - 1$. Thus, $0 \leq |MIS(G_1)| - |M_B(G_1)| \leq k+1$. (3). Remove an edge $(u, v)$: We have $|M_B(G_1)| = |M_A(G_0)|$ or $|M_A(G_0)| + 1$. According to Theorem 5.1, it holds that $|MIS(G_1)| \leq |MIS(G_0)| + 1$. ■

## VI. A Linear Algorithm

To strive for the nontrivial tradeoff between accuracy and efficiency, we propose an efficient and effective algorithm.

### A. Lazy Search

Let us recall the search processing in Algorithm 1. Actually, a state expanding tree, denoted by *SE-Tree*, consists of multiple search trees, where a search tree is defined as follows.

*Definition 6.1:* (Search Tree). A search tree, denoted as *S-Tree*, is a subtree of state expanding tree, where (1) The root is the update vertex; (2) If the node is an $MIS$ vertex $u$, its child is one of $u$'s children in *SE-Tree*; (3) If the node is a *non-MIS* vertex $v$, its children are $v$'s children in *SE-Tree* excluding the vertices that are certain to be accessed through $v$'s other $MIS$ neighbors in *S-Tree*.

TABLE I
Summary of all cases

| Update cases | | $M_B(G_1)$ | $\lvert MIS(G_1)\rvert$ | $\delta$ |
|---|---|---|---|---|
| add $v$ | | $MIS(G_0) \cup v$ | $\lvert MIS(G_0)\rvert + 1$ | 0 |
| remove $v$ | 1.1 | $MIS(G_0)/v$ | $\lvert MIS(G_0)\rvert$ or $\lvert MIS(G_0)\rvert - 1$ | 0 or 1 |
| | 1.2 | $MIS(G_0)/v \cup u$ | $\lvert MIS(G_0)\rvert$ | 0 |
| | 2 | $MIS(G_0)$ | $\lvert MIS(G_0)\rvert$ | 0 |
| add $(u, v)$ | 1 | $MIS(G_0)$ | $\lvert MIS(G_0)\rvert$ | 0 |
| | 2.1 | $MIS(G_0)/\{u\} \cup \{u'\}$ or $MIS(G_0)/\{v\} \cup \{v'\}$ | $\lvert MIS(G_0)\rvert$ | 0 |
| | 2.2 | $MIS(G_0)/\{u, v\} \cup \{u', v'\}$ | $\lvert MIS(G_0)\rvert$ | 0 |
| | 2.3 | $MIS(G_0)/\{u\}$ | $\lvert MIS(G_0)\rvert$ or $\lvert MIS(G_0)\rvert - 1$ | 0 or 1 |
| remove $(u, v)$ | 1.1 | $MIS(G_0)/\{u'\} \cup \{u, v\}$ | $\lvert MIS(G_0)\rvert + 1$ | 0 |
| | 1.2 | $MIS(G_0)$ | $\lvert MIS(G_0)\rvert$ or $\lvert MIS(G_0)\rvert + 1$ | 0 or 1 |
| | 2.1 | $MIS(G_0) \cup \{v\}$ | $\lvert MIS(G_0)\rvert + 1$ | 0 |
| | 2.2 | $MIS(G_0) \cup \{v\} \cup S/T$ | $\lvert MIS(G_0)\rvert + 1$ | 0 |
| | 2.3 | $MIS(G_0)$ | $\lvert MIS(G_0)\rvert$ or $\lvert MIS(G_0)\rvert + 1$ | 0 or 1 |

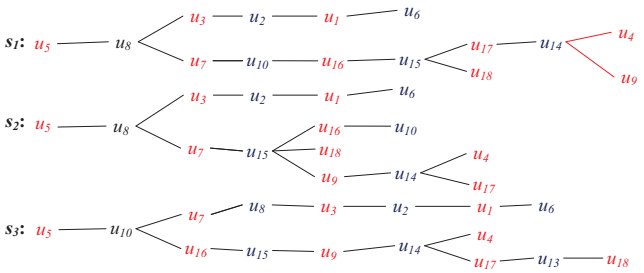



Fig. 3. Several search trees for the running example.

Figure 3 lists 3 *S-Tree*s for the running example in Figure 1. Each *S-Tree* corresponds to a possible search instance. We say a branch in *S-Tree* is ***besieged*** if the corresponding leaf node is a *non-MIS* vertex. Then we have the following lemma.

*Lemma 6.1:* If an *S-Tree* $s$ forms a solution, each branch must be besieged.

*Proof:* If $s$ forms a solution, it indicates that $|s.MIS| = |s.non\text{-}MIS|$ (resp. $|s.MIS|+1 = |s.non\text{-}MIS|$ corresponding to edge removal), where $|s.MIS|$ and $|s.non\text{-}MIS|$ are the number of $MIS$ and *non-MIS* vertices in $s$, respectively. When the root node is an $MIS$ vertex corresponding to vertex removal or edge addition between two $MIS$ vertices (resp. a *non-MIS* vertex corresponding to edge deletion), if a branch is not besieged, we get that $|s.MIS| > |s.non\text{-}MIS|$ (resp. $|s.MIS| = |s.non\text{-}MIS|$) since each intermediate $MIS$ node has a *non-MIS* child node. It contradicts the premise that $s$ forms a solution. ■

*Observation 1:* A certain subgraph may be visited repeatedly for many times in Algorithm 1.

In the running example of Figure 1, the subgraph consisting of vertices $\{u_4, u_9, u_{14}, u_{15}, u_{16}, u_{17}, u_{18}\}$ is visited 7 times, 3 of which are presented in Figure 3. Note that the repeatedly visited combinations contribute little to a solution. Furthermore, the combinations of different such subgraphs will significantly increase the search cost.

Based on the observation above, we propose an efficient method, i.e., the lazy search, whose basic idea is: we try to find a solution in the way of interleaving depth-first-search and breadth-first-search. If we can find such a solution, an accurate $MIS$ is returned. Otherwise, we invalidate some vertices that has been visited, which will not be considered in later search. Algorithm 6 gives the details.

Each time the algorithm is invoked, it needs to check *Que* first. If *Que* is empty, it indicates that we have found a solution (lines 1-2). Otherwise, the search process proceeds. We get the first vertex $u$ in *Que* and delete it from *Que*. Then we consider each insertable neighbor $v$ of vertex $u$ (lines 5-19). If $v$ has not been visited, we check each neighbor vertex $w$ of $v$. For each unvisited $w$, if $w$ belongs to $MIS$, $w$ is pushed into *Que*. Otherwise, $Visit[w]$ is set to be true. Then we search the next level. Note that, if we do not find a solution, the current *non-MIS* vertex that is being considered is set to be invalid

**Algorithm 6** *LazySearch*$(Q, Valid, Visit, curPos, QSize)$

**Input:** $Q$: a queue consisting of $MIS$ vertices to be swapped out, $Valid[v]$: whether vertex $v$ is able to insert into $MIS$, $Visit[v]$: visiting status of $v$, $curPos$: current search depth;
**Output:** true or false (true: find a solution, false: no solution.)

```
1: if curPos = QSize then
2:     return true
3: u ← Q[curPos]          /* pop a vertex from Que */
4: QSize_temp ← QSize
5: for each v ∈ Nei(u) do
6:     if Valid[v] = ture and Visit[v] = false then
7:         P ← ∅; R ← ∅
8:         Visit[v] ← true; P ← P ∪ {v}; R ← R ∪ {v}
9:         for each w ∈ Nei(v) do
10:            if Visit[w] = false then
11:                Visit[w] ← true; R ← R ∪ {w}
12:                if w ∈ MIS then
13:                    Q[QSize] ← w, QSize ← QSize + 1
14:        if LazySearch(Q, Valid, Visit, curPos + 1, QSize) = true
           then
15:            return true
16:        else
17:            QSize ← QSize_temp
18:            set the vertices in P invalid
19:            set the vertices in R unvisited
20: return false
```

(lines 17-19). In the following search, the vertices labeled as invalid will not be visited any more.

**Time Complexity**. As shown in Algorithm 6, if an *S-Tree* does not form a solution, the corresponding vertices and edges will not be considered later, which indicates that each edge is visited once at most. Hence, the time complexity is $O(E(G))$.

*Example 1:* Let us consider the graph in Figure 1. In the search tree $s_1$, the branch $v_{15} \rightarrow u_{18}$ is not besieged. According to Algorithm 6, the vertex $v_{15}$ is set to be "invalid". Then it can avoid repeated visiting of the subgraph that consists of $\{u_4, u_9, u_{14}, u_{15}, u_{16}, u_{17}, u_{18}\}$ as Algorithm 1 does.

### B. Conditional Besieging Based Pruning

Although lazy search reduces the search space significantly, we can further accelerate the searching process by employing the besieging transitivity as follows.

*1) Conditional Besieging:* Let us consider the running example. The lazy search may generate a search tree $s_1$ as shown in Figure 3. $s_1$ is not determined to be "unbesieged" until it reaches vertex $u_{18}$ after visiting 14 vertices. Actually, the process can be accelerated. The main idea is: we precompute the besieging status (besieged or not) of each vertex, based on which we can avoid lots of unnecessary searches. Generally, it is difficult to determine whether a vertex is besieged unless a specific exploring order is given. For example, $u_{14}$ is unbesieged if its neighbor $u_4$ is not visited before $u_{14}$.

*Definition 6.2:* (MIS vertex conditional besieged and non-MIS vertex conditional unbesieged). An MIS vertex $u$ (reps. a non-MIS vertex $v$) is conditional besieged (resp. unbesieged) by $v \in Nei(u)$ (resp. $u \in Nei(v)$), denoted as $u.Y(v)$ (resp. $v.N(u)$), if $v$ (resp. $u$) is besieged (resp. unbesieged) and visited after $u$ (resp.$v$).

TABLE II
BESIEGING STATUS OF VERTICES

| ID | $N()$ | $Y()$ | ID | $N()$ | $Y()$ | ID | $N()$ | $Y()$ |
|---|---|---|---|---|---|---|---|---|
| $u_1$ | – | $\{u_6\}$ | $u_7$ | $\{u_{10}, u_{13}, u_{15}\}$ | – | $u_{13}$ | $\{u_{17}\}$ | – |
| $u_2$ | – | $\{u_1\}$ | $u_8$ | $\{u_7\}$ | – | $u_{14}$ | $\{u_4\}$ | – |
| $u_3$ | – | $\{u_2\}$ | $u_9$ | $\{u_{14}\}$ | – | $u_{15}$ | $\{u_9\}$ | – |
| $u_4$ | $\emptyset$ | – | $u_{10}$ | $\{u_{16}\}$ | – | $u_{16}$ | $\{u_{13}, u_{15}\}$ | – |
| $u_5$ | – | – | $u_{11}$ | – | – | $u_{17}$ | $\{u_{14}, u_{15}\}$ | – |
| $u_6$ | – | $\emptyset$ | $u_{12}$ | – | – | $u_{18}$ | $\{u_{13}, u_{15}\}$ | – |

*Definition 6.3:* (MIS vertex conditional unbesieged and non-MIS vertex conditional besieged). An MIS vertex $u$ (reps. a non-MIS vertex $v$) is conditional unbesieged (resp. besieged) by $S \subseteq Nei(u)$ (resp. $S \subseteq Nei(v)$), denoted as $u.N(S)$ (resp. $v.Y(S)$), if the vertices in $S$ are unbesieged (reps. besieged) and are not visited before $u$ (resp. $v$).

If a vertex $u$ is unbesieged or besieged by a set $S$ or a vertex $v$, the set $S$ or $v$ is is called a besieging set or besieging vertex, respectively. Instead of enumerating all subsets of $Nei(u)$, we only consider some special vertices next.

*Lemma 6.2:* The vertex $u \in MIS$ (resp. $v \in non\text{-}MIS$) of degree 1 is unbesieged (resp. besieged) unless the update operation is deleting $u$ (resp. $v$).

*Proof:* If the update operation is not deleting $u$ (resp. $v$) and the search tree contains $u$ (resp. $v$), it is obvious that $u$ (resp. $v$) must be a leaf node since $u$ (resp. $v$) has only one neighbor which is visited before $u$ (resp. $v$). Hence, $u$ (resp. $v$) is unbesieged (resp. besieged). ■

*Lemma 6.3:* For the vertex $u \in MIS$ with two neighbors $v$ and $w$, if there is an edge between $v$ and $w$, $u$ is unbesieged unless the update operation is deleting $u$.

*Proof:* If the update operation is not deleting $u$, $u$ must be a leaf node since only one of the two neighbors $v$ and $w$ can be used and visited before $u$. Hence, $u$ is unbesieged. ■

The unbesieged and besieged vertices satisfying Lemma 6.2 and Lemma 6.3 are special cases of conditional unbesieged and besieged. Actually, the besieging is transitive according to Definition 6.2 and Definition 6.3. That is, the besieging status of a vertex may be determined based on the besieging status of its neighbors. For instance, vertex $u_8$ is conditional unbesieged since its neighbor $u_7$ is unbesieged. Based on besieging status we can determine whether a vertex $u \in MIS$ (resp. $v \in non\text{-}MIS$) is removable (resp. insertable).

*Theorem 6.1:* A vertex $u$ is (resp. not) removable or insertable if there is a besieging set $S \subseteq u.Y()$ (resp. $S \subseteq u.N()$) such that each vertex in $S$ is not visited before $u$.

*Proof:* If there is a besieging set $S \subseteq u.Y()$ (resp. $S \subseteq u.N()$) such that each vertex in $S$ is not visited before $u$, the vertex $u$ is besieged (resp. unbesieged), which leads to the conclusion that $u$ is removable/insertable or not. ■

*Example 2:* The besieging status of each vertex in the running example is listed in Table II. Assume the update operation is deleting $u_5$. There are four optional vertices to expand, i.e., $u_{10}$, $u_{11}$, $u_{12}$ and $u_{13}$. Since the vertex $u_{16} \in u_{10}.N()$ is not visited, we can rule out $u_{10}$ safely based on Theorem 6.1

**Algorithm 7** *UnbesiegedComputation*($G$)

**Input:** input graph $G$;
**Output:** the unbesieging states of each vertex.
1: $G' \leftarrow G$, $Que \leftarrow \emptyset$
2: **for** each vertex $u \in MIS$ **do**
3: **if** $u$ satisfies Lemma 6.2 **then**
4: $u.N() \leftarrow Nei(u)/Nei'(u)$
5: add $u$ into $v.N()$// $v$ is the only vertex of $u$
6: delete $u$ and its incident edges from $G'$
7: add $v$ into $Que$
8: **if** $u$ satisfies Lemma 6.3 **then**
9: $u.N() \leftarrow Nei(u)/Nei'(u)$
10: add $u$ into $v.N()$, add $u$ into $w.N()$
11: delete $u$ and its incident edges from $G'$
12: add $v$ and $w$ into $Que$
13: delete vertices in $Que$ and their incident edges from $G'$

**Algorithm 8** *BesiegedComputation*($G$)

**Input:** input graph $G$;
**Output:** the besieging states of each vertex.
1: $G' \leftarrow G$, $Que \leftarrow \emptyset$
2: **for** each vertex $v \in non\text{-}MIS$ **do**
3: **if** $u$ satisfies Lemma 6.2 **then**
4: $v.Y() \leftarrow Nei(v)/Nei'(v)$
5: add $v$ into $u.Y()$// $u$ is the only vertex of $v$
6: delete $v$ and its incident edges from $G'$
7: add $u$ into $Que$
8: delete vertices in $Que$ and their adjacent edges from $G'$

instead of extending $u_{10}$ and generating the whole search tree. Similarly, vertex $u_{13}$ can be filtered out as well. Thus, it can reduce the time cost significantly.

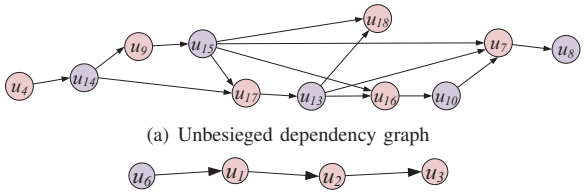
*2) Computation of Conditional Besieging:* Algorithms 7-8 outline the process of computing unbesieging and besieging status, respectively. To compute the "unbesieged" status, we consider two cases: One is the *MIS* vertex of degree 1 (lines 2-7). The other one is the *MIS* vertex with two neighbors $v$ and $w$, where $v$ is connected with $w$ (lines 8-12). Note that it needs to apply Algorithm 7 (resp. Algorithm 8) repeatedly until no vertex is newly labeled as "unbesieged" ("besieged").

**Time Complexity**. In Algorithms 7-8, a vertex and its adjacent edges will be removed when it is labeled as "unbesieged" or "besieged". Hence, each edge will be visited once at most. The time complexity of computing the besieging status is $O(E(G))$.

*3) Update of Conditional Besieging:* To efficiently maintain these conditional besieged or unbesieged vertices, we devise dependency graphs here.

*Definition 6.4:* (Besieged/Unbesieged dependency graph). The source node of a besieged/unbesieged dependency graph is a vertex satisfying Lemma 6.2 or Lemma 6.3. The child nodes (except the tail node) of each node $u$ are the vertices that are conditional besieged/unbesieged by $u$.

Figure 4 presents the besieging dependency graphs for the running example. Since adding a single vertex does affect the besieging status, we just consider the other three update operations. If the update operation results in new vertices of degree 1 or 2, Algorithm 7 will be invoked to compute the besieging status. (1) Deleting a vertex $u$ (Algorithm 9):



(a) Unbesieged dependency graph

(b) Besieged dependency graph

Fig. 4. Besieging dependency graphs

First, we delete $u$ from the unbesieged dependency graph that contains $u$. If $u \in MIS$, we check the child $v$ of $u$. If $v.N() = u$, we need to update the besieging status of each descendant of $u$. Otherwise, we just remove $u$ from $v.N()$. If $u$ is contained in a besieged dependency graph, we check the child $v$ of $u$. If $v.Y() = u$, the besieging status of each descendant of $u$ is recomputed. Otherwise, $u$ is removed from $v.Y()$. Similarly, we can deal with the other two operations. (2) Adding an edge $(u, v)$: If $u \in MIS$, $v \in non\text{-}MIS$, and $u$ is conditional unbesieged, we add $u$ into $v.N()$ and recompute the besieging status starting from vertex $v$. If $v$ is conditional besieged, we add $v$ into $u.Y()$ and recompute the besieging status starting from $u$. If $u \in non\text{-}MIS$ and $v \in non\text{-}MIS$ or ($u \in MIS$ and $v \in MIS$), we delete $u$, $v$ and their descendants from the corresponding besieged dependency graph. (3) Deleting an edge $(u, v)$: If $u \in MIS$ and $u \in v.N()$, we recompute the besieging status of $v$ and its descendants. If $u \in non\text{-}MIS$ and $u \in v.Y()$, we need to recompute the besieging status of $v$ and its descendants. If $u \in non\text{-}MIS$ and $v \in non\text{-}MIS$, we remove $w$ from $u.N()$ and $v.N()$, and recompute the besieging status of $u$, $v$, and their descendants.

**Time Complexity**. There are two dominant procedures in Algorithm 9, i.e., invoking Algorithm 7 and updating the besieging status. Hence, the time complexity of is $O(E(G))$.

### C. A Petal-based Pruning

In the search, the exploration of some vertices may be expensive. However, it may contribute nothing to find a solution even though the exploration is conducted exhaustedly.

**Algorithm 9** *BesiegingUpdate-VertexDeletion*($G, o$)

**Input:** input graph $G$, and vertex deletion operation $o$;
**Output:** the updated besieging states of each vertex.
1: **if** $o$ results in new vertices of degree 1 or 2 **then**
2: invoke Algorithm 7
3: **for** each unbesieged dependency graph containing $u$ **do**
4: **if** $u \in MIS$ **then**
5: **for** each child of $u$ **do**
6: **if** $v.N() = u$ **then**
7: update the besieging status of $u$'s descendant
8: **else**
9: remove $u$ from $v.N()$
10: delete $v$, $u$ and their incident edges from $G'$
11: **for** each besieged dependency graph containing $u$ **do**
12: **for** each child of $u$ **do**
13: **if** $v.Y() = u$ **then**
14: update the besieging status of $u$'s descendants
15: **else**
16: remove $u$ from $v.Y()$

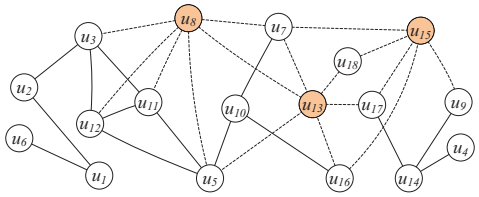



Fig. 5. 3-petal for the graph in Figure 1.

Hence, the underlying heuristic is: we can avoid the costly exploration of the unpromising vertices. Generally, the high-degree vertices are less likely to be in a maximum independent set [13]. Following the similar intuition, we propose two concepts, petal and pistil, as follows.

*Definition 6.5:* (k-Petal and k-Pistil). The $k$-petals of a graph are the structures that are left by removing some vertices such that the degree of each vertex in the petals is not larger than $k$. The removed vertices are called k-pistil.

The sum of vertices in all $k$-petals is the size of $k$-petals. Figure 5 shows an example of 3-petal for the running example graph in Figure 1, where the dashed vertices ($u_8$, $u_{13}$ and $u_{15}$) are the pistils, and the other vertices and edges constitute the 3-petal. The petals may be disconnected, that is, the graph may be decomposed into multiple small subgraphs.

*Theorem 6.2:* Computing $k$-petals of the largest size (i.e., computing the minimum k-pistil) is an NP-hard problem.

*Proof:* The proof can be achieved by considering a special case that $k = 0$. Then the minimum vertex cover problem, which is a known NP-hard problem [27], can be reduced to the problem of computing the minimum *k-pistil*. According to the definition of $k$-petal and *k-pistil*, when $k = 0$, the *k-pistil* is a vertex cover exactly. If there exists a polynomial algorithm to solve the minimum *k-pistil*, the minimum vertex cover problem can be solved by invoking this algorithm as well. Thus, the proof is achieved. ■

For the ease of presentation, we assume that petals have been computed in this subsection. Generally, the search space extended by the pistil vertices will be too large since they often have a large number of neighbors. Moreover, the statuses of the pistil vertices (being in the $MIS$ or not) are less likely to change. In other words, lots of efforts may be wasted to create little benefit. Hence, we do not consider the pistil vertices in the search process. In particular, we can add some constraints in validity check in Algorithm 6. Specifically, if the vertex $v \in Nei(u)$ (line 5 in Algorithm 6) is a pistil vertex, it will be abandoned. Similarly, if the vertex $w$ (in line 9) is a pistil vertex, it will not be pushed into the queue $Q$.

**Computation of k-Petals.** As shown in Theorem 6.2, computing petals of the largest size is non-trivial. Hence, we devise a heuristic method. The main idea is that: we first select some vertices that must be in the *k-petal* and then try to add more vertices to the selected set in a greedy manner.

*Lemma 6.4:* The vertices of degree no larger than $k$ must belong to the k-petal.

*Proof:* According to Definition 6.5, it is straightforward to obtain this lemma. ■

As shown in Lemma 6.4, we can add the vertices whose degrees are no larger than $k$ to the *k-petal* and remove them from the input graph $G$. For each vertex $v$ in the remaining graph $G'$, we assign a counter, denoted by $\gamma(v)$, to record the number of neighborhood vertices in the current *k-petal*. If $\gamma(v)$ is larger than $k$, i.e., $\gamma(v) > k$, the vertex $v$ cannot be in the *k-petal*. Thus it can be removed from $G'$ and added into the *k-pistil*. Then we repeatedly remove the vertex with the largest degree until there is no vertex of degree larger than $k - \gamma(v)$. The removed vertices are added into the *k-pistil* and the remaining vertices are added into the *k-petal*. Algorithm 10 outlines the details of the processing.

**Time complexity**. We need to identify the vertices that must be in the *k-petal* as shown in Algorithm 10 (lines 2-4). It is required to explore the vertices with the time complexity $O(V(G))$. Then we screen out the vertices that cannot belong to the *k-petal* (lines 5-8). As we need to compute $PN(v)$ by exploring the neighborhood of $v$, the time cost is $O(d \cdot V(G))$, where $d$ is the average vertex degree. To determine whether there are vertices of degree larger than $(k - PN(v))$ and select the vertex with the largest degree, we can maintain the vertex in a decreasing order. Hence, the overall time complexity is $max\{O(V(G) \cdot d), O(V(G) \cdot logV(G))\}$.

## VII. Experiments

In this section, we evaluate the effectiveness, i.e., the quality (size) of the independent set returned by these algorithms, and efficiency of our proposed algorithms over extensive graphs.

### A. Experimental Setting

**Dataset.** In the experiments, we use 14 real graphs which are downloaded from the Stanford Network Analysis Platform [28]. Table III presents the statistics of these graphs. To simulate the update operations, we randomly add/remove $m$ vertices/edges.

**Algorithms.** In this paper, we consider two state-of-the-art competitors VCSolver [10] and NearLinear [21], where VCSolver can compute a maximum independent set and NearLinear is designed to find a large independent set. We obtain the source code and executable files from the authors. Each time an update operation arrives, VCSolver or NearLinear is

**Algorithm 10** *GreedyPetal*$(G, k)$

**Input:** $G$: the input original graph $G$;
**Output:** *Petal*, a *k-petal*
1: $Petal \leftarrow \emptyset$
2: **for** each $v \in V(G)$ **do**
3: **if** $d(v) \leq k$ **then**
4: $Petal \leftarrow Petal \cup \{v\}$, $G \leftarrow G/v$
5: **for** each $v \in V(G)$ **do**
6: compute $\gamma(v)$
7: **if** $\gamma(v) > k$ **then**
8: $G \leftarrow G/v$
9: **while** $G$ contains vertices of degree larger than $(k - \gamma(v))$ **do**
10: remove the vertex with the largest degree from $G$
11: $Petal \leftarrow Petal \cup V(G)$
12: **return** *Petal*

TABLE III
GRAPHS AND THEIR CHARACTERISTICS

| Graph | $\|V(G)\|$ | $\|E(G)\|$ | $\bar{d}$ |
|---|---|---|---|
| QrGc | 5,242 | 14,484 | 5.53 |
| wiki-Note | 7,115 | 201,524 | 28.32 |
| HepPh | 12,008 | 236,978 | 19.74 |
| AstroPh | 18,772 | 198,050 | 21.1 |
| CondMat | 23,133 | 93,439 | 8.08 |
| loc-brightkite | 58,228 | 428,156 | 7.35 |
| Epinions | 75,879 | 405,740 | 10.69 |
| email | 265,214 | 364,481 | 2.75 |
| BerkStan | 685,230 | 6,649,470 | 19.41 |
| as-skitter | 1,696,415 | 11,095,398 | 13.08 |
| soc-pokec | 1,632,803 | 44,602,928 | 27.32 |
| wiki-Talk | 2,394,385 | 4,659,565 | 3.89 |
| LiveJ | 4,847,571 | 42,851,237 | 17.68 |
| com-orkut | 3,072,441 | 234,370,166 | 76.28 |

invoked to compute the *MIS* or independent set, respectively. We implement the following algorithms and evaluate them with the competitors systematically.

- ExactUpdate: our proposed algorithm to compute the exact *MIS* for dealing with updates (see Section IV).
- LSTwo: our local search algorithm with the time complexity $O(d^2)$ (see Section V).
- LazySearch: our proposed lazy search algorithm with the time complexity $O(E(G))$ (see Section VI-A).
- LazySearch$^+$: the lazy search equipped with besieging and petal-based pruning (see Sections VI-B, VI-C).
- LSTwo+LazySearch: Perform LSTwo first and then conduct LazySearch.
- LSTwo+LazySearch$^+$: Perform LSTwo first and then conduct LazySearch$^+$.

All the algorithms above are implemented in C++ and the experiments are conducted on a Windows Server 2008 with an Intel(R) Xeon(R) CPU E5504 @ 2.00GHz and 300G RAM.

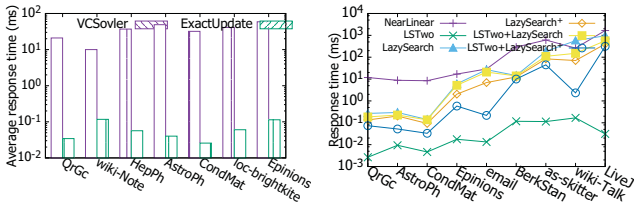
**Metrics.** *Quality*: the size of independent set returned by an algorithm. The larger the independent set, the better the algorithm. *Efficiency*: the response time from an update operation occurring over the graph $G_i$ to returning the solution on graph $G_{i+1}$. We report the average time of $X$ update operations on each graph, where $X$ ranges from 500 to 10,000.

### B. Experimental Results

We evaluate our algorithms and compare them with the state-of-the-art methods through extensive experiments.

*1) Evaluate Exact Algorithms:* We compare our proposed *ExactUpdate* with the state-of-the-art approach VCSolver. Figure 6(a) gives the average response time of the two methods w.r.t. 1,000 update operations. We can see that *ExactUpdate* outperforms VCSolver on all graphs by around two orders of magnitude. The main reason is that VCSolver computes *MIS* from scratch each time an update arrives.

*2) Evaluate Greedy Algorithms:* **Quality of the greedy algorithms**. To evaluate the effectiveness of the proposed greedy algorithms, we compare them with the latest algorithm *NearLinear* [21]. For the greedy algorithms, we use the *MIS* as the initial input. Table IV presents the average gap of the




(a) Exact algorithms (b) Greedy algorithms

Fig. 6. Efficiency test (average response time)

reported independent set size to independence number (i.e., the size of the *MIS*) over 1,000 update operations. Clearly, our proposed algorithms outperform *NearLinear* significantly. The main reason is that *NearLinear* computes independent sets from scratch without considering the solution on the previous graph as well as the update operation. In contrast, our algorithms compute the independent sets based on the former one. *LazySearch* improves *LSTwo* greatly by using the lazy search strategy to expand the search space. Besides the lazy search strategy, *LazySearch*$^+$ adopts the *k-petal* based technique and increases the size of independent set. By combining the two methods *LSTwo* and *LazySearch* (resp. *LazySearch*$^+$), the method *LSTwo+LazySearch* (resp. *LSTwo+LazySearch*$^+$) improves the size of independent set slightly.

**Efficiency of the greedy algorithms**. Figure 6(b) describes the average time consumed of algorithms for 1,000 update operations. It is obvious that our proposed methods outperform the competitor *NearLinear*. *LSTwo* is the fastest among all the algorithms as it only considers the vertices within 2 hops at most. *LazySearch* enlarges the search space and consumes more time. By adopting the conditional besieging and *k-petal* based techniques, the search space is reduced. Hence, *LazySearch*$^+$ improves the time efficiency in comparison with *LazySearch*. Combining *LSTwo* and *LazySearch* leads to slight improvements since it will invoke *LazySearch* only if *LSTwo* does not find a solution. Similarly, *LSTwo+LazySearch*$^+$ outperforms *LazySearch*$^+$ by integrating *LSTwo*. In summary, *LSTwo+LazySearch*$^+$ produces independent sets of the best quality and runs very fast (just slower than the constant time algorithm *LSTwo*).

**Memory cost evaluation**. The right part of Table IV presents the memory usage of the methods *LSTwo, LazySearch, LazySearch*$^+$. *LSTwo* consumes the least memory since it does not require any extra information and only considers the neighborhood vertices within two hops. *LazySearch*$^+$ demands more space than *LazySearch* because it maintains the besieging and *k-pistil* vertices. Nevertheless, both of them take up comparable storage as *LSTwo*.

**Effect of the initial independent set**. In some cases, e.g., the graph is too large to deal with, it is difficult to attain the exact initial *MIS*. Therefore, we perform an inexact algorithm first and take its output independent set as input. Table V illustrates the average gap to the independence number of the proposed algorithms w.r.t. 10,000 update operations. Comparing with the results that are generated based on the exact input as shown

TABLE IV
THE AVERAGE GAP TO THE INDEPENDENCE NUMBER AND THE MEMORY USAGE

| Graphs | Average Gap to the Independence Number | | | | | | Memory Usage (MB) | | |
|---|---|---|---|---|---|---|---|---|---|
| | NearLinear | LSTwo | LazySearch | LazySearch$^+$ | LSTwo+LazySearch | LSTwo+LazySearch$^+$ | LSTwo | LazySearch | LazySearch$^+$ |
| QrGc | 18.6 | 0.219 | 0.152 | 0.044 | 0.059 | 0.039 | 12.932 | 13.076 | 13.732 |
| AstroPh | 23.4 | 0.082 | 0.052 | 0.019 | 0.016 | 0.016 | 22.924 | 23.236 | 23.426 |
| CondMat | 35.3 | 0.141 | 0.078 | 0.018 | 0.016 | 0.015 | 17.668 | 18.052 | 18.145 |
| Epinions | 112.6 | 0.171 | 0.049 | 0.029 | 0.029 | 0.021 | 35.716 | 36.616 | 36.764 |
| email | 47.4 | 0.012 | 0 | 0 | 0 | 0 | 40.248 | 43.628 | 43.888 |
| BerkStan | 398.1 | 0.702 | 0.312 | 0.209 | 0.208 | 0.202 | 379.08 | 387.792 | 388.464 |
| as-skitter | 74.2 | 0.218 | 0.097 | 0.054 | 0.054 | 0.046 | 643.868 | 665.412 | 667.072 |
| wiki-Talk | 8.9 | 0.101 | 0 | 0 | 0 | 0 | 336.564 | 366.972 | 369.312 |
| LiveJ | 232.8 | 0.227 | 0.173 | 0.087 | 0.087 | 0.081 | 2,392.06 | 2,453.604 | 2,458.34 |
| soc-pokec | 30.4 | 0.014 | 0.002 | 0.008 | 0.002 | 0 | 1,219.992 | 1,240.728 | 1,242.324 |
| com-orkut | 94.7 | 0.010 | 0 | 0.011 | 0 | 0.009 | 6,163.012 | 6,202.028 | 6,205.032 |

TABLE V
EFFECT OF THE INITIAL INDEPENDENT SET

| Graphs | Average Gap to the Independence Number | | | | |
|---|---|---|---|---|---|
| | Initial Gap | LSTwo | LazySearch | LazySearch$^+$ | LSTwo+LazySearch$^+$ |
| QrGc | 5 | 0.295 | 0.174 | 0.065 | 0.047 |
| AstroPh | 34 | 0.092 | 0.085 | 0.031 | 0.022 |
| CondMat | 11 | 0.167 | 0.093 | 0.029 | 0.019 |
| Epinions | 153 | 0.198 | 0.061 | 0.045 | 0.033 |
| email | 70 | 0.017 | 0.009 | 0.007 | 0.007 |
| BerkStan | 456 | 0.783 | 0.394 | 0.269 | 0.237 |
| as-skitter | 39 | 0.281 | 0.133 | 0.076 | 0.058 |
| wiki-Talk | 670 | 0.164 | 0.011 | 0.008 | 0.008 |
| LiveJ | 36 | 0.256 | 0.198 | 0.107 | 0.095 |

TABLE VI
EFFECT OF $|T|$ ON THE GAP TO THE INDEPENDENCE NUMBER

| Graphs | $\|T\| = 500$ | $\|T\| = 1,000$ | $\|T\| = 2,000$ | $\|T\| = 5,000$ | $\|T\| = 10,000$ |
|---|---|---|---|---|---|
| QrGc | 1 | 1 | 4 | 32 | 59 |
| AstroPh | 0 | 2 | 2 | 2 | 16 |
| CondMat | 0 | 0 | 0 | 10 | 15 |
| Epinions | 0 | 1 | 2 | 11 | 21 |
| email | 0 | 0 | 0 | 1 | 0 |
| BerkStan | 8 | 23 | 36 | 99 | 202 |
| as-skitter | 3 | 8 | 13 | 23 | 46 |
| wiki-Talk | 0 | 0 | 0 | 1 | 0 |
| LiveJ | 6 | 3 | 18 | 43 | 81 |

in Table IV, we can find that they are very close. It shows that these methods are insensitive to the quality of the initial independent set and have good robustness.

*3) Evaluate Besieging and Petal-based Pruning:* As shown in Table IV, equipped with the besieging and petal-based techniques *LazySearch*$^+$ and *LSTwo+LazySearch*$^+$ outperform *LazySearch* and *LSTwo+LazySearch* on the gap to the independence number, respectively. *LazySearch* disables vertices in a heuristic way, which may generate false positives. Different from that, the besieging pruning is accurate. The petal-based pruning may enlarge the search space by releasing some promising vertices. Moreover, the response time is also reduced as depicted in Figure 6(b). *LSTwo+LazySearch*$^+$ outperforms *LSTwo+LazySeacrh* by around two orders of magnitude on the graphs "email" and "wiki-Talk".

We further study the effect of $k$ (the size of petal). Figure 7(a) shows the gap to independence number by varying $k$ from 1 to 8 for *LSTwo+LazySearch*$^+$ (on 5,000 update operations). It gains little improvement when $k$ is larger than 6 on most graphs. The reason is that it has limited effect to extend the search space if $k$ is small. On the contrary, if $k$ gets larger, the probability of releasing promising vertices will decrease as the size of $k$-petal will be smaller, which is depicted in Figure 7(c). The corresponding average time consumed w.r.t. $k$ is reported in Figure 7(b). We can find that the average response time increases slowly with the growth of $k$, which demonstrates the efficiency of the approach.

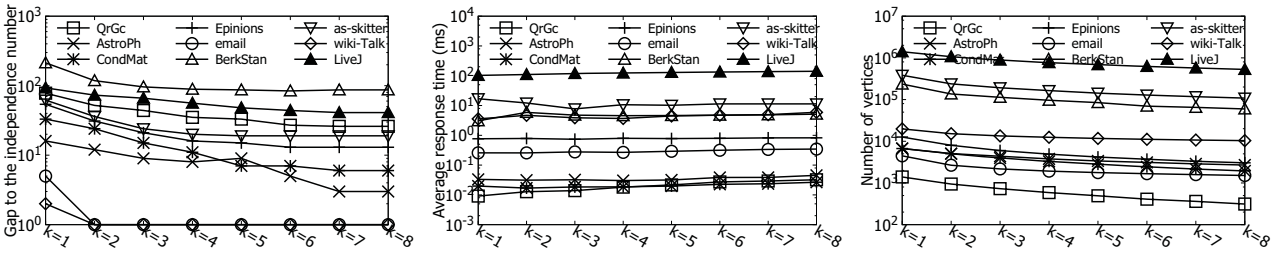
*4) Evaluate The Update Operations:* We study the effect of update operations in this section.

**Effect of different update operations**. Figure 8(a) and Figure 8(b) depict the quality and time efficiency of *LSTwo+LazySearch*$^+$ with respect to 10,000 update operations, respectively. Edge removal has the highest quality in most cases as removing a vertex requires deleting the incident edges as well. Furthermore, edge removal is the most efficient to deal with. And vertex removal consumes the most time as shown in Figure 8(b).

**Scalability test**. We also test the scalability by varying the number of update operations (denoted by $|T|$) from 500 to 10,000. Table VI reports the gap to the independence number of the algorithm *LSTwo+LazySearch*$^+$ over the graphs. It shows that the gap grows as we increase the number of update operations on most graphs. Moreover, the growing ratio is near linear to the amount of update operations. Figure 8(c) presents the effect of $|T|$ on the response time. It is clear that it will consume more time to deal with more update operations. The gap between two adjacent lines keeps nearly stable, which indicates that our method has good scalability.

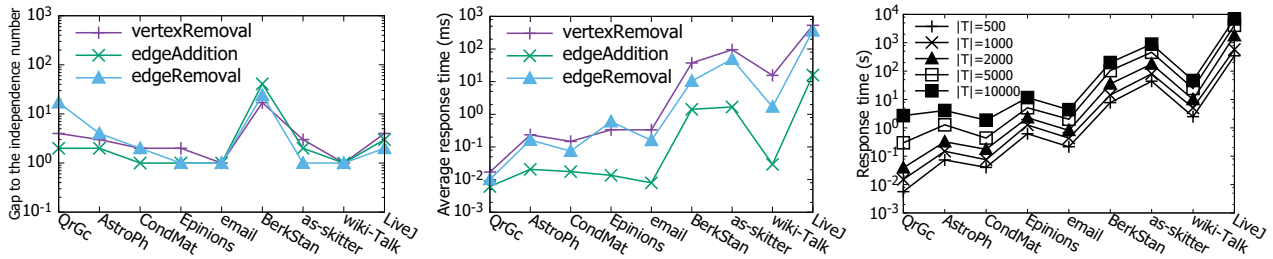
## VIII. CONCLUSIONS

Computing *MIS* is an interesting fundamental problem in graph theory. However, finding the (or maximum) independent set over evolving graphs brings new challenges and little attention has ever been paid so far. By exploiting the preceding independent set rather than computing from scratch, we propose




(a) Gap to the independence number (b) Average response time (c) The size of $k$-petal

Fig. 7. Effect of $k$




(a) Gap to the independence number (b) Average response time (c) Scalability test: time efficiency

Fig. 8. Effect of the update operations

systematic methods including the exact and greedy methods (*LSTwo, LazySearch and LazySearch*$^+$) to compute the exact or approximate *MIS*, respectively. Extensive experiments over a wide range of graphs demonstrate that our methods are very efficient to find high-quality independent sets.

## Acknowledgment

This work was substantially supported by grants from the Research Grant Council of the Hong Kong Special Administrative Region, China [Project No.: CUHK 14205617] and [Project No.: CUHK 14221716]. Lei Zou was supported by NSFC under grant 61622201 and 61532010.

## References

[1] Y. Chen and Y. Chen, "An efficient algorithm for answering graph reachability queries," in *ICDE*, 2008, pp. 893–902.

[2] E. P. F. Chan and H. Lim, "Optimization and evaluation of shortest path queries," *VLDB J.*, vol. 16, no. 3, pp. 343–369, 2007.

[3] P. Zhao and J. Han, "On graph query optimization in large networks," *PVLDB*, vol. 3, no. 1, pp. 340–351, 2010.

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

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

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

[7] T. A. Feo, M. G. C. Resende, and S. H. Smith, "A greedy randomized adaptive search procedure for maximum independent set," *Operations Research*, vol. 42, no. 5, pp. 860–878, 1994.

[8] A. Gemsa, M. Nöllenburg, and I. Rutter, "Evaluation of labeling strategies for rotating maps," in *Experimental Algorithms - 13th International Symposium, SEA 2014*, 2014, pp. 235–246.

[9] T. Kieritz, D. Luxen, P. Sanders, and C. Vetter, "Distributed time-dependent contraction hierarchies," in *Experimental Algorithms, 9th International Symposium*, 2010, pp. 83–93.

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

[11] F. V. Fomin, F. Grandoni, and D. Kratsch, "A measure & conquer approach for the analysis of exact algorithms," *J. ACM*, vol. 56, no. 5, pp. 25:1–25:32, 2009.

[12] R. E. Tarjan and A. E. Trojanowski, "Finding a maximum independent set," *SIAM J. Comput.*, vol. 6, no. 3, pp. 537–546, 1977.

[13] J. Dahlum, S. Lamm, P. Sanders, C. Schulz, D. Strash, and R. F. Werneck, "Accelerating local search for the maximum independent set problem," in *Experimental Algorithms - 15th International Symposium, SEA 2016*, 2016, pp. 118–133.

[14] M. Xiao and H. Nagamochi, "Exact algorithms for maximum independent set," in *ISAAC*, 2013, pp. 328–338.

[15] J. Håstad, "Clique is hard to approximate within $n^{1-\text{epsilon}}$," in *Symposium on Foundations of Computer Science*, 1996, pp. 627–636.

[16] M. M. Halldórsson and J. Radhakrishnan, "Greed is good: approximating independent sets in sparse and bounded-degree graphs," in *Proceedings of ACM Symposium on Theory of Computing*, 1994, pp. 439–448.

[17] G. L. Nemhauser and L. E. T. Jr., "Vertex packings: Structural properties and algorithms," *Math. Program.*, vol. 8, no. 1, pp. 232–248, 1975.

[18] U. Feige, "Approximating maximum clique by removing subgraphs," *SIAM J. Discrete Math.*, vol. 18, no. 2, pp. 219–225, 2004.

[19] S. Lamm, P. Sanders, C. Schulz, D. Strash, and R. F. Werneck, "Finding near-optimal independent sets at scale," in *ALENEX*, 2016, pp. 138–150.

[20] Y. Liu, J. Lu, H. Yang, X. Xiao, and Z. Wei, "Towards maximum independent sets on massive graphs," *PVLDB*, vol. 8, no. 13, pp. 2122–2133, 2015.

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

[22] S. Wasserman and K. Faust, *Social Network Analysis: Methods and Applications*. Structural Analysis in the Social Sciences, 2009.

[23] V. Boginski, S. Butenko, and P. M. Pardalos, *On Structural Properties of the Market Graph*. Edward Elgar Publishers, 2003.

[24] P. M. Pardalos and D. Du, *Handbook of Combinatorial Optimization: Supplement*. Kluwer Academic Publishers Boston Ma, 2005.

[25] V. Boginski, S. Butenko, and P. M. Pardalos, "Statistical analysis of financial networks," *Computational Statistics & Data Analysis*, vol. 48, no. 2, pp. 431–443, 2005.

[26] S. Khanna, R. Motwani, M. Sudan, and U. V. Vazirani, "On syntactic versus computational views of approximability," *SIAM J. Comput.*, vol. 28, no. 1, pp. 164–191, 1998.

[27] G. Karakostas, "A better approximation ratio for the vertex cover problem," *ACM Trans. Algorithms*, vol. 5, no. 4, pp. 41:1–41:8, 2009.

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