2019 IEEE 35th International Conference on Data Engineering (ICDE)

Computing A Near-Maximum Independent Set In Dynamic Graphs

Weiguo Zheng[†], Chengzhi Piao[‡], Hong Cheng[‡], Jeffrey Xu Yu[‡]
[†] *School of Data Science, Fudan University, China*
[‡] *Department of Systems Engineering and Engineering Management, The Chinese University of Hong Kong*
zhengweiguo@fudan.edu.cn, {czpiao,hcheng,yu}@se.cuhk.edu.hk

***Abstract*—As a fundamental NP-hard problem in graph theory, the maximum independent set (MIS) has attracted a lot of efforts to improve the time efficiency. However, most graphs in real scenarios are usually changing over time. But the previous studies take the stationary graphs as input, the computation of MIS in dynamic graphs receives little attention. Since computing the exact MIS is intractable, we compute the high-quality (large-size) independent set for dynamic graphs in this paper, where 4 graph updating operations are allowed: adding or deleting a vertex or an edge. Based on two state-of-the-art reduction rules that are designed for static graphs, we propose a novel scheme, i.e., dependency graph based independent set computation, which can support computing the high-quality independent set on the basis of the previous result rather than calculating from scratch. Moreover, a dynamic searching strategy is devised to improve time efficiency. In order to make it more useful in practical applications, we devise an effective yet efficient method to deal with the batch update. To confirm the effectiveness and efficiency of the proposed methods, we conduct extensive experiments over both real and synthetic datasets.**

I. Introduction

Since more and more data in real-world scenarios, such as bioinformatics, semantic web, social networks, and software engineering, can be modeled as graphs, researches on graph data have attracted a lot of attention these years, e.g., shortest path query [1], reachability computation [2], subgraph search [3], and similarity query [4], [5]. The *maximum independent set* (*MIS*) problem is a classic NP-hard problem in graph theory [6]. Given an input graph G, it finds the largest independent set $MIS(G)$, where each two vertices $u \in MIS(G)$ and $v \in MIS(G)$ have no edges in G. As discussed in the existing work, the *MIS* problem is closely related to two well-known problems, i.e., the minimum vertex cover problem and the maximum clique problem [7], [8]. For example, to compute the maximum clique of a graph equals finding the *MIS* in its complementary graph with edges between all non-neighbors in the original graph.

A. Applications

The *MIS* problem has a wide spectrum of applications, such as wireless network [9], macromolecular docking [10], collusion detection [11], association rule mining [12], automated map labeling [13] and road network routing [14].

Note that the traditional *MIS* problem only considers the stationary graphs without any updates. However, in many real-world applications the graphs are changing (i.e., adding or removing vertices/edges) continuously. For example, in a social network two users may establish a friendship, then an edge is added between the two vertices that correspond to the two users. Similarly, a user can also remove the edges between his neighbors and himself. Thus maintaining high-quality independent sets over the dynamic social network is very useful for studying cohesive subgraphs since the clique is a foundational idea in the task and computing cliques equals computing independent sets in the complementary graph as discussed above [15]. In the market graph that models the stock price, the financial instruments (i.e., stocks) can be represented as vertices, the cross-correlation between the price fluctuations can be represented as edges [16], [17]. In stock investments, a wise strategy is portfolio diversification [18]. For instance, one may buy several stocks that are not binding together, where "binding" can be defined as two stocks rise or drop simultaneously over a time window (i.e., the edges in the market graph). Then the independent set can be computed for the market graph. Since the "correlations" may change over time, providing the real-time independent set is useful for investors. It is clear that studying the *MIS* problem over dynamic graphs is very interesting and has high practical significance.

B. MIS Computation Over Static Graphs

There have been many efforts for the *MIS* (approximate) computation. They can be divided into three categories.

Exact Algorithms. Due to the NP-hardness of the *MIS* problem, it takes exponential time in terms of the number of vertices in G. The branch-and-bound framework with advanced vertex reordering strategies and pruning is widely used in the existing algorithms [19], [20]. To reduce the searching space, Tarjan et al. and Dahlum et al. apply reduction rules to remove the vertices that are definitely (or not) contained in a certain *MIS* [21], [22]. Much research has been devoted to reducing the base of the exponent. Based on the measure and conquer analysis, Fomin et al. propose an algorithm with the time complexity $O(1.2201^n)$, where n is the number of vertices in G [20]. Xiao et al. improve the time complexity to $O(1.1996^n n^{O(1)})$ by using the rule of 'branching on edges" [23]. Even with these state-of-the-art algorithms, it remains intractable to handle large graphs with millions of vertices.

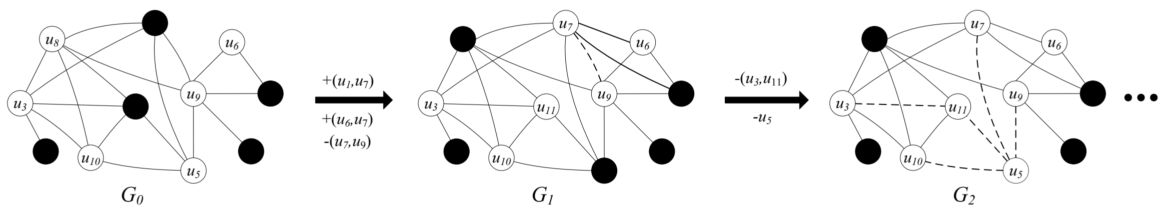


Fig. 1. A running example.

Approximation Algorithms. It has been proved that there does not exist a constant approximation factor for general graphs [24], [25]. Moreover, computing the *MIS* with the approximation ratio $n^{1-\epsilon}$ or less is NP-hard, where $0 < \epsilon < 1$, as shown in [26]. Nevertheless, there are a lot of efforts devoted to deriving some approximation ratios [27], [28], [29].
Greedy Algorithms. Instead of computing the exact *MIS* or the approximation ratios, many algorithms adopt heuristics to compute high-quality (may not be the *MIS*) independent sets [30], [31], [7], [32], [33]. Plateau search is a widely used technique that is typically achieved through node swaps, i.e., replacing a node by one of its neighbors. It only accepts the moves that do not change the objective function. In addition to plateau search, Grosso et al. design various diversification operations and restart rules [31]. Two local search routines are introduced by Andrade et al. to determine whether a maximal solution can be improved by replacing a single vertex with two others or replacing two solution vertices with three vertices [7]. Lamm et al. develop an advanced evolutionary algorithm, which incorporates kernelization techniques to compute large independent sets in huge sparse networks [33]. A major drawback of the kernelization-based algorithms is that they require significant preprocessing overhead. To handle the problem, Dahlum et al. propose to perform simple kernelization techniques in an online fashion. Moreover, they also show that cutting the high-degree vertices can boost local search performance [22]. When the main memory can accommodate all vertices of the graph but not all edges, Liu et al. present a greedy algorithm and a vertex-swap framework [32]. A state-of-the-art method is proposed by Li et al. recently. It iteratively applies reduction rules on vertices and removes the vertex with the highest degree when no reduction rules can be applied [34].

C. MIS Over Evolving Graphs

As discussed above, many real-life graphs are dynamically changing. Generally, there are 4 kinds of updating operations: vertex/edge addition and vertex/edge removal. However, most of the existing algorithms take static graphs as input. Hence, they cannot be used for computing the *MIS* over evolving graphs directly. A naive approach is computing the *MIS* by invoking the algorithms that are designed for static graphs. Each time the solution is computed from scratch, which is costly for frequently updated large-scale graphs.

Recently, Zheng et al. propose a lazy search strategy to enable the *MIS* computation over dynamic graphs [8]. The main idea is that an exact *MIS* is delivered if a solution is found in the search; Otherwise, some visited vertices will not be explored further. However, when the initial solution is not optimal, the quality of the returned answers is not satisfying after a few rounds of updates. Furthermore, multiple updating operations may occur at a time in the real-life scenario. However, only one updating operation is allowed each time by the existing algorithms. Hence, they are not efficient enough.
Challenges and Our Approaches. The first challenge is to maintain the quality of a solution. Since it is necessary to compute the *MIS* based on the previous result for dynamic graphs and the solution in each time point may be inaccurate, a challenging problem is to maintain or improve the solution quality rather than decrease the quality significantly. Essentially, to maintain the quality of solutions, a critical task is to find the complementary vertices, denoted by C, and add them into the solution when some vertices, denoted by R, are moved out of the solution (caused by the updating operations) such that $|C| \geq |R|$. To the end, we propose a novel framework for dynamic *MIS* computation. An effective index, named as the dependency graph, is devised to support the computation, which is easy to construct by applying the state-of-the-art reduction rules including the degree-one reduction rule [20] and degree-two reduction rules [34]. Based on the index, it can find the complementary vertices in an easy fashion. More importantly, the quality of solutions almost does not decrease regardless of how large the gap between the initial input and the optimal one as shown through extensive experiments. In other words, it is insensitive to the quality of the initial input.

As proved in [8], it is NP-hard to compute $MIS(G_{i+1})$ based on $MIS(G_i)$ since there is no local property for the evolving *MIS*. Therefore, the second challenge is improving the time efficiency as much as possible without sacrificing the quality of solutions. The dependency graph can guide finding the complementary vertices easily by reducing the search space significantly. When the dependency graph is built based on the degree-two reduction rules, the average time complexity is $O(d)$ and the worst-case time complexity is $O(|V(G)| \times |E(G)|)$. To guarantee the efficiency, we devise a bottom-up algorithm and design a dynamic searching strategy, which produces an algorithm with the time complexity of $O(|E(G)|)$ in the worst case. Furthermore, the existing method assumes that only one updating operation occurs each time, which may be unrealistic in real-life application. Therefore, we present a method to deal with the batch updates rather than consider them individually.
Contributions. In summary, we make the following contributions.

- We propose a novel framework that is insensitive to the quality of initial input for computing the evolving *MIS* based on the state-of-the-art reduction rules.

- We develop an algorithm with the average time complexity $O(d)$ to perform valid checking. Following the dynamic searching strategy, we devise a bottom-up algorithm to guarantee the worst-case time complexity $O(|E(G)|)$.
- An efficient approach is designed to deal with the batch update for computing the *MIS* over evolving graphs.
- We conduct extensive experimental studies over a bunch of large-scale real graphs. As confirmed in the experiments, the proposed methods are both effective and efficient to deliver high-quality solutions.

II. Problem Definition

In this section, we introduce the basic concepts and define the task of the paper. Without loss of generality, we focus on *undirected graphs*. Note that our technique is easy to extend to handle directed graphs. Let G denote a graph consisting of $(V(G), E(G))$, where $V(G)$ and $E(G)$ are vertex and edge sets.

Definition 2.1: (Independent Set). An independent set, denoted by $IS(G)$, is the set of vertices in G such that each two vertices $u \in IS(G)$ and $v \in IS(G)$ have no edge in G.

Definition 2.2: (Maximal Independent Set). A maximal independent set, denoted by $LIS(G)$, is the independent set that is not a subset of any other independent set.

Definition 2.3: (Maximum Independent Set). Given a graph G, a maximum independent set, denoted by $MIS(G)$, is the largest independent set of G.

Note that $IS(G)$, $LIS(G)$, and $MIS(G)$ refer to only one independent set, maximal independent set, and maximum independent set, respectively. Following conventions, four updating operations are allowed to modify a graph, i.e., vertex addition, vertex deletion, edge addition, and edge deletion. Given a maximum independent set $MIS(G_i)$ of graph G_i at time t_i and an updating operation o_i over G_i at time t_{i+1}, computing $MIS(G_{i+1})$ of G_{i+1} at time t_{i+1} based on $MIS(G_i)$ has already proven to be NP-hard [8]. As discussed above, there may be a bunch of updating operations rather than just one updating operation each time.

Definition 2.4: (Individual Update and Batch Update). Individual update refers to the case that only one updating operation is allowed each time. Batch update refers to the case that a set of updating operations are allowed each time.

Example 1: Fig. 1 presents a running example, where the grey vertices constitute an *MIS* in each graph G_i. In the first update, there are three updating operations, i.e., adding the edges (u_1, u_7) and (u_6, u_7), and removing the edge (u_7, u_9). Thus we can obtain G_1 whose *MIS* is $\{u_1, u_2, u_4, u_5, u_8\}$. The second update consists of two updating operations, i.e., removing the edge (u_3, u_{11}) and deleting the vertex u_5. Correspondingly, the maximum independent set of G_2 is $\{u_1, u_2, u_4, u_8\}$.

Due to the computational hardness, it is costly to compute the exact *MIS* in real time when the graph changes frequently especially for a very large graph. Moreover, computing a high-quality independent set (not necessary to be the maximum one) is acceptable and interesting in real applications [8]. Therefore, we do not strive for the exact solutions in this paper. Formally, we can define the problem as follows.

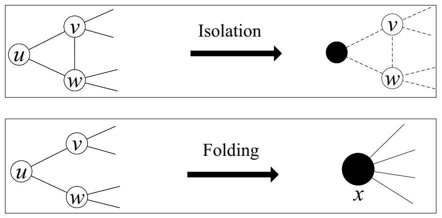


Fig. 2. Degree-two reductions.

Problem Statement. *Given an independent set $IS(G_i)$ of graph G_i at time t_i and a set of updating operations O 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}.*

A high-quality maximal independent set involves two aspects: (1) It should be a maximal independent set; (2) The independent set is as large as possible.

III. A Novel Framework for Maintenance

In this section, we focus on individual updates and develop a novel framework for maintaining a high-quality independent set. We handle the batch updates in Section V.

A. Preliminary–Algorithms on Static Graphs

Reducing-and-peeling is a state-of-the-art framework for computing the independent set over a static graph [34]. It iteratively applies reduction rules to reduce the graph size by determining whether or not the vertices with degree 1 or 2 should be included in a maximum independent set. If no reduction rules can be applied on the current graph it removes the vertex with the highest degree. Let $G \backslash S$ represent the graph obtained by removing the vertices in S and their adjacent edges from G, where S is a set of vertices.

Lemma 3.1: (Degree-one Reduction [20]) If a vertex u has only one neighbor v in the graph G, there exists a maximum independent set of G that includes u; thus, v can be removed from G, and $MIS(G) = MIS(G \setminus \{v\})$.

BDOne [34] handles degree-one vertices with the observation that there must be a maximum independent set including the vertex u of degree one [20] as depicted in Lemma 3.1. Specifically, it iteratively deletes the neighbor v of a degree-one vertex u from G and updates the degree of each neighbor w of v in G. If there is no degree-one vertex, it deletes the vertex u with the highest degree from G and updates the degree of each neighbor w of u. Finally, it extends the independent set to be a maximal one.

If $Nei(u) = \{v, w\}$, we use $G/\{u, v, w\}$ to represent the graph obtained by replacing u, v, and w with a new vertex x whose neighbors consist of $Nei(v) \cup Nei(w)$, where $Nei(v)$ (resp. $Nei(w)$) is the set of neighbors of v (resp. w) excluding u.

Lemma 3.2: (Degree-two Reductions [34]) Considering a degree-two vertex u in G with two neighbors v and w:
(1) Isolation [22]: $(v, w) \in E(G)$. There exists a maximum independent set of G that includes u; thus, the vertices v and w can be removed from G, and $MIS(G) = MIS(G \setminus \{v, w\})$.
(2) Folding [20]: $(v, w) \notin E(G)$. If $x \in MIS(G/\{u, v, w\})$, then $MIS(G/\{u, v, w\}) \setminus \{x\} \cup \{v, w\}$ is an MIS of G; otherwise, $MIS(G/\{u, v, w\}) \cup \{u\}$ is an MIS of G. u is a folding vertex.

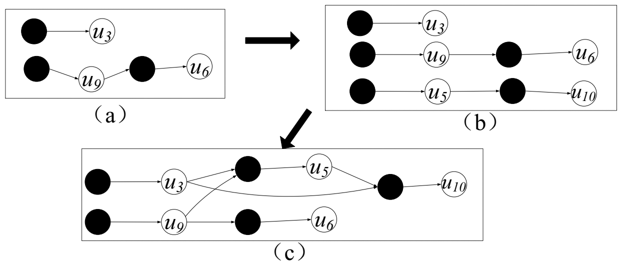


Fig. 3. DG-one construction.

Example 2: Fig. 2 gives examples for Degree-two reductions "Isolation" and "Folding", respectively. "Isolation" add the degree-two vertex u into the maximum independent set directly, while "Folding" constructs a supervertex x "uvw" by contracting the vertices u, v, and w.

B. Dependency Graph

Before introducing the dependency graph, we first define reducing vertex, dependent vertex, and peeling vertex.

Definition 3.1: (Reducing Vertex and Dependent Vertex). In the reducing-and-peeling framework, a vertex is called a reducing vertex if it is certain to be included in a maximum independent set. A vertex is called a dependent vertex if it is removed due to the finding that its neighbor belongs to a maximum independent set.

Definition 3.2: (Peeling Vertex). A peeling vertex is the vertex that is deleted when no reduction rules can be applied in the reducing-and-peeling framework.

We can build a dependency graph which is used to maintain a high-quality independent set efficiently.

Definition 3.3: (Dependency Graph). A dependency graph, denoted by DG, is a directed graph that is generated in the reducing-and-peeling procedure. Each time a reducing vertex u and its dependent vertices are removed from G and added into DG. An edge is added starting from u to each of its dependent vertices. The procedure proceeds until G is empty. An edge is added from each dependent vertex v to the reducing vertex w, if $w \in Nei(v)$ and w is not the parent of v.

Obviously, applying different reduction rules will lead to diverse dependency graphs. To make it clear, we introduce the dependency graphs generated based on Reduction 1 and Reduction 2 in Section III-B1 and Section III-B2, respectively. According to the dependency graphs, we can compute the independent sets in an efficient way in Section III-C.

1) Degree-one Reduction Based DG: Let *DG-one* denote the dependency graph that is generated based on the degree-one reduction. The degree-one reduction in Lemma 3.1 is very simple. Thus the corresponding dependency graph is easy to construct as shown in Algorithm 1.

First, we compute two sets of vertices of degree one and at least two in G, denoted by $V_{=1}$ and $V_{\geq 2}$, respectively. For each vertex $u \in V_{=1}$ if any, we add u and its neighbor v into DG, where u is a reducing vertex and v is u's dependent vertex. At the same time, a directed edge from u to v is added in DG (lines 4-5). Then we need to delete u and v from G and update the vertex degrees of the neighbors of v by invoking the procedure DeleteVertex(v) (lines 17-23). If $V_{=1} = \emptyset$, we need to delete the vertex u with the highest degree in G (lines 9-10). After obtaining the independent set, we need to add the edges between the non-reducing vertex u in DG and the reducing vertices that are u's neighbors in G as shown in lines 11-15. Actually, during the construction of a dependency graph, we can also obtain the independent set $IS(G)$ that is generated by applying the Degree-one Reduction.

Algorithm 1 *DG-one Construction*

```
Input:  The input graph $G$;
Output: Dependency graph $DG$ and independent set $IS(G)$.
 1: Let $V_{=1}$ and $V_{\geq 2}$ be the sets of vertices of degree one and at least
    two in $G$, respectively
 2: while $V_{=1} \neq \emptyset$ or $V_{\geq 2} \neq \emptyset$ do
 3:   if $V_{=1} \neq \emptyset$ then
 4:     add a vertex $u \in V_{=1}$ and its neighbor $v$ into $DG$
 5:     add an edge from $u$ to $v$ in $DG$
 6:     DeleteVertex($v$)     /* invoking Procedure*/
 7:     add $u$ into $IS(G)$ and remove $u$ from $G$, $V_{=1}$
 8:   else
 9:     $u \leftarrow$ the vertex $u$ with the highest degree in $G$
10:     DeleteVertex($u$)     /* invoking Procedure*/
11: for each vertex $u \in DG$ do
12:   if $u \notin IS(G)$ then
13:     for each neighbor $v$ of $u$ in $G$ do
14:       if $v \in IS(G)$ and $(v, u) \notin DG$ then
15:         add an edge from $u$ to $v$ in $DG$
16: return $DG$ and $IS(G)$
    Procedure DeleteVertex($v$)
17: for each neighbor $w$ of $v$ do
18:   $d(w) \leftarrow d(w) - 1$
19:   if $d(w) = 1$ then
20:     remove $w$ from $V_{\geq 2}$ and add $w$ into $V_{=1}$
21:   else if $d(w) = 0$ then
22:     remove $w$ from $V_{=1}$ and add $w$ into $IS(G)$
23: remove $v$ from $G$, $V_{=1}$ and $V_{\geq 2}$
```

Example 3: Let us consider the graph G_0 in Fig. 1. In the first phase, there are three Degree-one reductions that can be applied over G_0. Thus we can obtain the dependency graph as shown in Fig. 3(a). Then we select u_8 as the peeling vertex. A new path $u_7 \rightarrow u_5 \rightarrow u_{11} \rightarrow u_{10}$ is added into the dependency graph as depicted in Fig. 3(b). The set of vertices $\{u_1, u_2, u_4, u_7, u_{11}\}$ forms the corresponding independent set of G_0. Finally, we check the reducing vertices that are neighbors (in G) of the non-reducing vertex u in DG. Since u_7 (reducing vertex) is a neighbor of u_3 (dependency vertex), *DG-one* is constructed as presented in Fig. 3(c).

The reducing vertices constitute the independent set $IS(G)$. Moreover, the generated *DG-one* may be disconnected and there is no circle in *DG-one*.

Time complexity. The first part (lines 1-10) of Algorithm 1 mainly generates an independent set by applying the Degree-one reduction. Meanwhile, we add the reducing vertices and dependent vertices into the dependency graph. The corresponding time cost is $O(|E(G)|)$. The second part adds the edges from dependent vertices to reducing vertices as shown in lines 11-15. Therefore, the overall time complexity of Algorithm 1 is $O(|E(G)|)$.

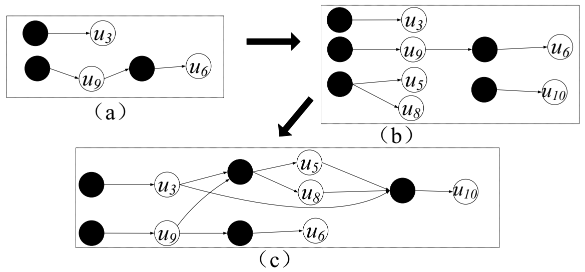


Fig. 4. DG-two construction (fictitious edges).

2) Degree-two Reductions Based DG: The dependency graph that is generated based on the degree-two reductions is denoted by *DG-two*. Different from the Degree-one reduction rule, Degree-two reductions contain two cases as shown in Lemma 3.2. We deal with them individually next.

Isolation reduction. We first deal with the case that the degree-two vertex u has two neighbors v and w and there is an edge between v and w (u is called an isolation vertex). Similar to the Degree-one reduction, we can add u into $IS(G)$ directly. The difference lies in that two vertices v and w will be deleted from G. Thus u may have two dependent vertices[1].

Folding reduction. It needs to construct a super vertex by contracting the vertices u, v, and w together to obtain a graph G'. The upper bound of the time complexity is $O(|V(G)| \times |E(G)|)$ as discussed in [34]. Moreover, it is difficult to build the dependency graph as u, v, and w are not determined to be included in $MIS(G)$ or not until $MIS(G')$ is known.

Adding fictitious edges. To make it consistent, we devise a novel method to deal with the folding reduction. Specifically, when there are no degree-one vertices and isolation vertices, we add an edge between the two neighbors of a folding vertex u. The newly added edge is called a fictitious edge. Thus we can transform the folding vertex to an isolation vertex. Finally, we remove the fictitious edges from the dependency graph. The benefits are two-fold: (1) The Degree-one reduction, isolation reduction, and folding reduction can be incorporated into a unified framework; (2) The time cost can be reduced to $\max\{O(|E(G)|), O(\delta)\}$, where $O(\delta)$ denotes the cost of computing independent sets for edge removal updates. Note that $\delta = |E(G)|$ with the method proposed in the paper (as shown in Section IV-C).

Algorithm 2 presents the details of constructing the dependency graph *DG-two* by using the Degree-one and isolation reductions. Let ES denote the set of fictitious edges added into G. First, we adopt the Degree-one reduction if degree-one vertices exist, which is similar to the construction of *DG-one*. Then, we perform the isolation reduction (lines 8-12), where V_{iso} represents the set of isolation vertices. The vertices $u \in V_{iso}$, $v \in Nei(u)$, $w \in Nei(u)$ and the two edges (u, v) and (u, w) are added into DG. Correspondingly, the vertices v and w together with their neighbors are removed from G by invoking the procedure "DeleteVertex(v)". For a folding vertex $u \in V_{fol}$, we add an edge (v, w) between v and w in

[1]Since the Degree-one reduction is applied before the isolation reduction, a reducing vertex may have only one dependent vertex.

Algorithm 2 *DG-two Construction*

```
Input:   The input graph $G$;
Output:  $DG$, independent set $IS(G)$, and edge set $ES$.
 1: Let $V_{=1}$, $V_{iso}$, $V_{fol}$, and $V_{\geq 3}$ be the sets of degree-one, isolation,
    folding, and degree at least three vertices, respectively
 2: while $V_{=1} \neq \emptyset$ or $V_{iso} \neq \emptyset$ or $V_{fol} \neq \emptyset$ or $V_{\geq 3} \neq \emptyset$ do
 3:   if $V_{=1} \neq \emptyset$ then
 4:     add a vertex $u \in V_{=1}$ and its neighbor $v$ into $DG$
 5:     add an edge from $u$ to $v$ in $DG$
 6:     add $u$ into $IS(G)$ and remove $u$ from $G$, $V_{=1}$
 7:     DeleteVertex($v$)        /* invoking Procedure*/
 8:   else if $V_{iso} \neq \emptyset$ then
 9:     add a vertex $u \in V_{iso}$ and its neighbors $v$ and $w$ into $DG$
10:     add two edges from $u$ to $v$ and $w$ into $DG$, respectively
11:     add $u$ into $IS(G)$ and remove $u$ from $G$, $V_{=2}$
12:     DeleteVertex($v$), DeleteVertex($w$)
13:   else if $V_{fol} \neq \emptyset$ then
14:     add an edge $(v, w)$ between two neighbors $v$ and $w$ of a
        vertex $u \in V_{fol}$ into $G$ and $ES$
15:     move $u$ from $V_{fol}$ to $V_{iso}$
16:   else
17:     $u \leftarrow$ the vertex $u$ with the highest degree in $G$
18:     DeleteVertex($u$)        /* invoking Procedure*/
19: for each vertex $u \in DG$ do
20:   if $u \notin IS(G)$ then
21:     for each neighbor $v$ of $u$ in $G$ do
22:       if $v \in IS(G)$ and $(v, u) \notin DG$ then
23:         add an edge from $u$ to $v$ in $DG$
24: return $DG$, $IS(G)$, $ES$
    Procedure DeleteVertex($v$)
25: for each neighbor $w$ of $v$ do
26:   $d(w) \leftarrow d(w) - 1$
27:   if $w$ is an isolation vertex then
28:     remove $w$ from $V_{\geq 3}$ and add $w$ into $V_{iso}$
29:   else if $w$ is a folding vertex then
30:     remove $w$ from $V_{\geq 3}$ and add $w$ into $V_{fol}$
31:   else if $d(w) = 1$ then
32:     remove $w$ from $V_{iso}$, $V_{fol}$ and add $w$ into $V_{=1}$
33:   else if $d(w) = 0$ then
34:     remove $w$ from $V_{=1}$ and add $w$ into $IS(G)$
35: remove $v$ from $G$, $V_{=1}$, $V_{=2}$ and $V_{\geq 3}$
```

G, and add (v, w) to the edge set ES (lines 13-14). Then the vertex is moved from V_{fol} to V_{iso} as it is transformed to an isolation vertex (line 15). When no reduction rule can be applied, a vertex of the highest degree is deleted. In the procedure "DeleteVertex(v)", for each degree-two vertex w, we check whether there is an edge between w's two neighbors x and y to determine w is an isolation vertex or a folding vertex.

Example 4: Let us consider the graph G_0 in Fig. 1. Similar to Example 3, we can obtain the dependency graph as shown in Fig. 4(a) by applying the Degree-one reduction. Then a fictitious edge (u_5, u_8) is added to generate an isolation vertex u_7. So we do not need to delete any vertex. Finally, the dependency graph returned by Algorithm 2 is presented in Fig. 4(c), where u_7 has two dependent vertices.

Remark. Notice that the fictitious edges ES are used to support the construction of $DG(G)$. However, the DG returned by Algorithm 2 may not be the dependency graph for G. To obtain the real $DG(G)$, we need to remove the edges in ES by invoking the edge removal updates in Section III-D.

Time complexity. As shown in lines 2-18 in Algorithm 2, each edge in the graph G is visited once. Hence, the time cost of Algorithm 2 is $O(|E(G)|)$.

We can see that a reducing vertex has two dependent vertices at most in the dependency graph, while a dependent vertex has just one in-neighbor reducing vertex. As discussed above, we adopt the technique of adding fictitious edges to perform the degree-two reductions, which can be used to compute the independent set of a static graph. Clearly, it significantly reduces the time cost.

C. DG-based Independent Set Computation

To maintain a high-quality independent set, it is better to include a set of new vertices when some vertices are removed from the independent set. Generally, if a vertex is added into (resp. moved out from) the independent set $IS(G)$, all its neighbors should be moved out from (resp. added into) $IS(G)$. The process proceeds recursively until all combinations have been considered. As discussed in [8], it is clear that the time complexity is $O(d^{|V(G)|})$, where d is average degree of G. In order to improve time efficiency, we employ the dependency graph to guide the searching progress.

Intuition: A key operation of dealing with updates is to determine whether a vertex $u \in IS(G)$ can be moved out from $IS(G)$ without decreasing the independent set. In other words, we need to find a swap of two sets of vertices V_{in} and V_{out} such that $|V_{in}| \geq |V_{out}|$, where V_{in} and V_{out} denote the sets of vertices that are added into and removed out of $|IS(G)|$, respectively. To find such a swap based on the dependency graph $DG(G)$, we can check the out-neighbors in $DG(G)$. Since the reducing vertices and dependent vertices alternate in $DG(G)$, it is straightforward to determine the validity of a swap by exploring these vertices recursively.

Definition 3.4: (Valid Swap). In a valid swap, all the vertices removed out from $IS(G)$ or added into $IS(G)$ are valid.
(1) The vertex $u \in IS(G)$ is valid: one dependent vertex v of u's out-neighbors in the dependent graph $DG(G)$ is valid.
(2) The vertex $u \notin IS(G)$ is valid: u has no out-neighbors in $DG(G)$ or all u's out-neighborhood reducing vertices in $DG(G)$ are valid, and none of u's non-independent set neighbors in G is used in the swap.

Example 5: Let us consider DG-one in Fig. 3. u_1 is valid since its out-neighbor dependent vertex u_6 is valid, which indicates that we can replace u_1 with u_6 to maintain the size of the independent set. Then u_9 and u_4 are both valid.

Algorithm 3 gives the processing of performing the valid checking. If the vertex u is a reducing vertex, i.e., $u \in IS(G)$, where $IS(G)$ is the current independent set of G, we check each dependent vertex v in the out-neighbors of u in the dependency graph, denoted by $out\text{-}Nei_{DG}(u)$. To avoid adding two adjacent vertices into $IS(G)$, v should not have any neighbors in V_{in}, i.e., $Nei_G(v) \cap V_{in} = \emptyset$, where $Nei_G(v)$ represents the neighbors of v in G. If v is not used (i.e., $v \notin V_{in}$) and valid, we can conclude that u is valid. If we cannot find such a vertex v, u is invalid (lines 1-6). When u is a dependent vertex in DG, it needs to check all the reducing neighbors

Algorithm 3 *Valid-Checking*(u, DG, G)

```
Input:  The input graph $G$, dependency graph $DG$, vertex $u$;
Output: True ($u$ is valid) or False ($u$ is not valid).
 1: if $u \in IS(G)$ then
 2:     for each vertex $v \in out\text{-}Nei_{DG}(u)$ and $Nei_G(v) \cap V_{in} = \emptyset$ do
 3:         if $v \notin V_{in}$ and $Valid\text{-}Checking(v, DG, G)$ = True then
 4:             $V_{in} \leftarrow V_{in} \cup v$
 5:             return True
 6:     return False
 7: else if $u \notin IS(G)$ then
 8:     for each reducing vertex $v \in Nei_{DG}(u)$ do
 9:         if $v \notin V_{out}$ and $Valid\text{-}Checking(v, DG, G)$ = False then
10:             $V_{out} \leftarrow V_{out} / V_{temp}$
11:             return False
12:         $V_{temp} \leftarrow V_{temp} \cup v$
13:     $V_{out} \leftarrow V_{out} \cup V_{temp}$
14:     return True
```

$Nei_{DG}(u)$ of u in the dependency graph. If one unused vertex of $Nei_{DG}(u)$ is not valid, u is not valid (lines 7-12).
Time complexity. DG-one: As a reducing vertex in *DG-one* has just one out-neighbor dependent vertex at most, each edge is visited once. Thus the time cost is $O(|E(G)|)$ in the worst case. DG-two: A reducing vertex u may have two dependent vertices, either of which may lead to the conclusion that u is valid. Thus the time cost is $O(|V(G)| \times |E(G)|)$ in the worst case.

Let p_0, p_1, and p_2 denote the probabilities that a reducing vertex u has 0, 1, and 2 out neighbors, respectively. Let d denote the average number of out neighbors for each dependent vertex v in *DG-two*. We define X and Y as the average number of vertices visited for checking u and v, respectively. When we check a vertex $v \notin IS(G)$, v is invalid only if one of its d out neighbors has no out neighbors. Thus we can check the out neighbors of v by visiting the d vertices. If each of v's out neighbors has one out neighbor at least, we need to check them one by one. Therefore, we have the following equation.

$$\begin{aligned} Y &= 1 + d + (1-p_0)^d \cdot d \cdot (Y \cdot \frac{p_1}{p_1+p_2} + 2 \cdot Y \cdot \frac{p_2}{p_1+p_2}) \\ &= \frac{1+d}{1-(p_1+p_2)^{d-1} \cdot (p_1+2p_2) \cdot d} \end{aligned}$$

If we take p_0, p_1, and p_2 as three constants, it holds that $O(Y) = O(d)$. Since the reducing vertex has two out neighbors at most, we have $X = 1 + p_1 \cdot Y + 2p_2 \cdot Y$. Thus it holds that $O(X) = O(d)$. In conclusion, the average time complexity of Algorithm 3 is $O(d)$.

D. Handling Updates

Based on Algorithm 3 that determines the status (valid or not) of a vertex, it is easy to deal with the edge addition, edge deletion, and vertex deletion.

Edge addition. It contains three cases when adding an edge between two vertices u and v: (1) Neither u nor v is not the independent-set vertex, i.e., $u \notin IS(G)$ and $v \notin IS(G)$; (2) Only one of u and v is an independent-set vertex, i.e., $u \in IS(G)$ and $v \notin IS(G)$, or $u \notin IS(G)$ and $v \in IS(G)$; (3) Both u and v are the independent-set vertices, i.e., $u \in IS(G)$ and $v \in IS(G)$. Clearly, the first two cases do not affect the size of the independent set. So we just need to deal with the third one. Specifically, we check each of the two vertices u and v by invoking Algorithm 3. If one of them is valid, we can obtain a new independent set $IS(G_1)$ whose size equals $|IS(G_0)|$, i.e., $|IS(G_1)| = |IS(G_0)|$, where $G_1 = G_0 \cup (u, v)$.

Although the second case does not affect the independent set, we need to add the edge from the dependent vertex to the reducing vertex. For the third case, we just need to reverse the status (*reducing* or *dependent*) of the vertices in V_{in} and V_{out}, where V_{in} and V_{out} represent the vertices that are swapped in and out from $IS(G)$, respectively (returned by Algorithm 3). Meanwhile, the edges between these vertices are also inverted. Specially, when u has two dependent vertices v and w, and there is an edge (v, w) in G, we need to revise the in-neighbor reducing vertex of w from u to v if $u \in V_{out}$ and $v \in V_{in}$.

Edge deletion. There are two cases of deleting an edge (u, v): (1) One of u and v is an independent-set vertex, i.e., $u \in IS(G)$ and $v \notin IS(G)$; (2) Neither u nor v is not the independent-set vertex, i.e., $u \notin IS(G)$ and $v \notin IS(G)$. In the first case, we determine whether v is valid or not by invoking Algorithm 3. If v is valid, the size of the independent set will increase 1. In the second case, both u and v will be checked. If u or v is valid, the corresponding dependency graph will be updated by reversing the status of the vertices in V_{in} and V_{out}, which is similar to the process of adding an edge. Otherwise, we just delete the edge between u and v from the dependency graph directly for the first case.

Vertex deletion. Two cases of deleting a vertex u should be considered: (1) u does not belong to the independent set; (2) u is an independent-size vertex. In the first case, the independent set will not be affected by u's removal. Meanwhile, the dependency graph DG should be updated by removing u from DG if it belongs to DG. In the second case, we need to check each neighbor of u. If one of them is valid, we can maintain the size of the independent set by swapping the vertices in V_{out} and V_{in}. Then DG is updated by reversing (1) the status of vertices in V_{out} and V_{in}, (2) and the edges between them.

DG vs. the besieged/unbesieged dependency graph [8]. They have different structures in terms of both vertices and relations, and they are used for completely different purposes. *DG* defined in the paper is generated by applying the reduction rules in the process of computing the independent set. It captures the removal relations among the reducing vertices and the dependent vertices, and specifies the search space of finding a valid swap. In contrast, the besieged/unbesiedged dependency graph describes the conditional besieged/unbesieged relations. It is used to accelerate the determination of the besieging status of some vertices in the state expanding tree.

Algorithm 2 vs. BDTwo [34]. Algorithm 2 transforms the folding vertex to an isolation vertex rather than constructing the super vertex. Hence, its time complexity is $O(E(G))$. However, the upper bound of the time complexity for the degree-two reduction based algorithm, BDTwo [34], is $O(V(G) \times E(G))$. Moreover, the construction of the proposed dependency graph is not subject to BDOne or BDTwo. Generally, it can be built upon any given independent set as discussed in Section IV.

IV. Oracle-based Dependency Graph And Dynamic Search

In this section, we first present how to construct the dependency graph based on an exact *MIS*, and then propose a bottom-up dynamic searching algorithm to check vertices.

A. Oracle-based Dependency Graph

As shown in Section III-B, we use Degree-one and Degree-two reductions to generate the initial dependency graph. Since some vertices of the highest degree are deleted when no reduction rules can be applied, the discovered independent set may not be the maximum one and the corresponding generated dependency graph may not be optimal as well. In this section, we assume that we are given a maximum independent set (named as an oracle) and construct the dependency graph.

The underlying rationale is to apply the Degree-one and Degree-two reductions following the procedure that is similar to Algorithm 2. Suppose that the original maximum independent set of G is $MIS(G)$. Let *DG-oracle* denote the dependency graph that is constructed according to $MIS(G)$.

(1) We apply the degree-one reduction first. For the degree-one vertex u, if $u \in MIS(G)$, u is a reducing vertex and its neighbor v is its dependent vertex, which is the same as Algorithm 2. Otherwise, when $u \notin MIS(G)$, its neighbor v must be in $MIS(G)$. In this case, we replace v with u, which does not decrease the size of $MIS(G)$. Then the dependency graph can be constructed similar to the first case.

(2) When there is no degree-one vertex, we check each isolation vertex u, whose two neighbors are v and w. If $u \notin MIS(G)$, v or w must be in $MIS(G)$. Without loss of generality, suppose v belongs to $MIS(G)$. Then we replace v with u as it will not affect the size of $MIS(G)$. In other words, we can guarantee that u belongs to an MIS. Thus the DG is constructed following the procedure in Algorithm 2.

(3) If there are no degree-one/isolation vertices, we delete the vertex of the highest degree that is not in $MIS(G)$. The process proceeds until degree-one or isolation vertices are generated.

Clearly, the maximum independent set can guide the construction of the dependency graph by "telling" us which vertex should be deleted when no reduction rules can be applied.

B. Bottom-up Search

We can perform a bottom-up search to determine whether a vertex is valid or not. Then each edge is visited just once.

Definition 4.1: (Terminal Vertex). A vertex is a terminal vertex if it has no out-neighbors in the dependency graph.

The main idea is to check each terminal vertex first and then determine its in-neighbor status. Algorithm 4 presents the details, which consists of three steps, i.e., examination, infection, and propagation. First of all, we add the peeling vertices and the edges between them and the reducing vertices into the DG. Initially, the status of each vertex is "unknown".

(1) Examination (lines 5-8): For the unknown terminal vertex u, (a) If $u \notin IS(G)$, u is valid; (b) If $u \in IS(G)$ and u has no peeling neighbors, u is invalid.

(2) Infection (lines 9-12): For the terminal vertex u, (a) If u is a valid dependent vertex, we assign its in-neighbor reducing

Algorithm 4 *Bottom-up Search*

```
Input:  The input graph $G$, dependency graph $DG$;
Output: The status $\lambda(u)$ of each vertex $u$ in $DG$, updated $DG$.
1: adding peeling vertices and the edges between them and the reducing vertices into $DG$
2: for each vertex $u$ in $DG$ do
3:     $\lambda(u) \leftarrow$ "unknown"
4: while $DG$ contains terminal vertices do
5:     if the unknown terminal vertex $u \notin IS(G)$ then
6:         $\lambda(u) \leftarrow$ "valid"
7:     else if the unknown terminal vertex $u \in IS(G)$ then
8:         $\lambda(u) \leftarrow$ "invalid"
9:     if $u$ is a valid dependent vertex then
10:        set $u$'s in-neighbor reducing vertex $v$ as "valid"
11:    else if $u$ is an invalid reducing vertex then
12:        set $u$'s in-neighbor dependent vertices as "invalid"
13:    delete the valid/invalid terminal vertices from $DG$
14:    for each remaining valid/invalid vertex $u$ do
15:        delete the edges between $u$ and its out-neighbors in $DG$
16: for each vertex $u \in DG$ do
17:     $\lambda(u) \leftarrow$ "valid"
18: return $DG$, $\{\lambda(u)\}$
```

vertex as "valid"; (b) If u is an invalid reducing vertex, we assign its in-neighbor dependent vertices as "invalid".
(3) Propagation (lines 13-15): We delete the valid/invalid terminal vertices from the *DG*. For each remaining valid/invalid vertex u, the edges between u and its out-neighbors in *DG* are deleted. During the propagation, the status of a vertex cannot be revised once it is assigned with "valid" or "invalid". This procedure proceeds until there are no terminal vertices.

The remaining dependency graph consists of circles. Due to the nature of *DG*, the dependent vertices are no less than the reducing vertices. Then we have the following theorem.

Theorem 4.1: Each remaining vertex in the *DG* that is returned by Algorithm 4 is valid.

Proof: The updated *DG* that is returned by Algorithm 4 consists of circles. When the circles are independent to each other, each reducing vertex is followed by a dependent vertex. Hence, the number of reducing vertices, denoted by α_r, equals the number of dependent vertices, denoted by α_d. Then we consider the circles that share some common vertices. Since the dependent vertex has only one in-neighbor reducing vertex, it cannot be the starting vertex in the common path of two circles. If the length of the common path is even, $\alpha_d = \alpha_r$; Otherwise, $\alpha_d > \alpha_r$. Hence, it holds that $\alpha_d \geq \alpha_r$, which means that these vertices can be swapped and they are valid. ■

To determine that u is really valid or not, we need to check whether there is an edge between the valid dependent vertices of u's descendants in G. Since only one target vertex u is examined each time, it only needs to perform the edge existence checking for u. Thus the time cost is $O(|E(G)|)$.

C. Dynamic Searching Strategy

Algorithm 4 improves the worst-case time complexity. Nevertheless, it is outperformed by Algorithm 3 as the average time complexity of Algorithm 3 is $O(d)$, and it is lower than $O(|E(G)|)$, the average time complexity of Algorithm 4.

To guarantee the performance, we propose a dynamic searching strategy. Specifically, Algorithm 3 is conducted first. The number of edges that have been visited is counted as α. When α achieves $|E(G)|$, Algorithm 3 terminates and we use Algorithm 4 to determine the status of the target vertex. The algorithm terminates as soon as the status of the target vertex is obtained instead of computing the status of other vertices.

V. Batch Update

The previous algorithm demands that there is only one updating operation each time, which is not efficient enough in real-life scenario since multiple updating operations may arrive at the same time point. To make it more useful in practice, we propose an efficient algorithm for batch updates in this section.

A. Affected Subgraph

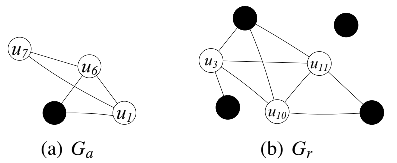
Without loss of generality, we suppose that there are k operations $o_1, o_2, \ldots, o_k$ over graph G_i at time t_i and the updated graph is denoted as G_{i+1}. The basic idea of the proposed method is: we partition the graph G_{i+1} into two subgraphs G_a and G_r, where G_a is the subgraph related to the updates and $G_r = G_{i+1}/G_a$. Then the constraint independent set of G_a is computed, based on which the independent set of G_{i+1} is recovered efficiently. The benefits are two folds: (1) It can improve the time efficiency since the updates are considered together instead of dealing with them one by one; (2) The quality of solutions can be improved as these updating operations are handled together to avoid the accumulated quality degradation. For example, if an edge is added in o_i and then deleted in o_j, where $i < j$, the quality may degrade when dealing with them individually because the exact solution is not guaranteed to be found for each updating operation.

Affected subgraph construction. We say a vertex u is affected by an updating operation if (1) u is added or deleted; or (2) One of its incident edges is added or deleted; or (3) One of its neighbors is deleted. First, we collect all the affected vertices by the k updating operations into a set V_a and then retrieve the subgraph G_a of G_{i+1} that is induced by these vertices[2]. The induced subgraph G_a is called the affected subgraph. Notice that the affected subgraph G_a may not be connected.

Let G_r denote the remaining subgraph that is induced by $V(G_i)\backslash V_a$. It is clear that there may be some crossing edges $(u, v) \in E(G_i)$ between G_a and G_r, where $u \in V(G_a)$ and $v \in V(G_r)$. Then we define the active vertex.

Definition 5.1: (Active Vertex). A vertex u in G_a is active if it is incident to a crossing edge (u, v) such that $v \in IS(G_i)$ and $v \in G_r$, where $IS(G_i)$ is the initial independent set of G_i.

[2]If $e(u, v) \in E(G_i)$, $e(u, v)$ is included in the subgraph G_a, where $u, v \in V_a$.



(a) G_a (b) G_r

Fig. 5. Affected subgraph of G_1 in Fig. 1

Example 6: The affected subgraph of G_1 in the running example in Fig. 1 is shown in Fig. 5(a), where u_9 is an active vertex. The remaining graph G_r is presented in Fig. 5(b).

To compute the independent set of G_{i+1}, we resort to the results $IS(G_r)$ and $IS(G_a)$ of G_r and G_a, respectively. After obtaining their results, we need to integrate them together. Ideally, the two sets have no intersection so that they can be combined together straightforwardly. However, the two sets may ***conflict with*** each other, i.e., there is at least an edge $(u, v) \in G_{i+1}$, where $u \in IS(G_r)$ and $v \in IS(G_a)$.

B. Maximum Constraint Independent Set

Note that $IS(G_i) \cap V(G_r)$ is an independent set of G_r, i.e., $IS(G_i) \cap V(G_r) = IS(G_r)$, based on which we can obtain a set of active vertices in G_a, denoted by V_c. To reduce conflicting vertices, we define the maximum constraint independent set.

Definition 5.2: (Maximum Constraint Independent Set). Given a graph G and a set of active vertices V_c, its maximum constraint independent set, denoted by $MCIS(G)$, is the largest independent set of G that includes the fewest active vertices.

Generally, a graph G may contain multiple maximum independent sets, each of which includes a subset of the active vertices. For instance, the $MCIS$ of G_a in Fig. 5(a) is $\{u_7, u_9\}$.

Theorem 5.1: Computing the maximum constraint independent set for the a graph G is NP-hard.

Proof: When the set of active vertices is empty, i.e., $V_c = \emptyset$, it becomes the traditional maximum independent set, which is a well-known NP-hard problem. ■

Assume that there is an existing algorithm designed for computing the *MIS*, denoted by *Alg-MIS*. A naive method is to remove x active vertices and their incident edges, and compute the *MIS* for the remaining graph by invoking *Alg-MIS*, where x is exhausted from 0 to $|V_c|$. Finally, the *MIS* of the largest size that includes the fewest active vertices is returned. This method is inefficient as it invokes *Alg-MIS* $2^{|V_c|}$ times in total. So we devise a more efficient algorithm next.

Algorithm 5 outlines the procedure. Initially, we perform *Alg-MIS* over G_a to obtain $MIS(G_a)$, which may contain t $(0 \leq t \leq |V_c|)$ active vertices. Then t is an *upper bound* of the number of active vertices for the *MCIS*. If $t = 0$, the *MCIS* is found luckily. Otherwise, we need to perform the exhausted search by computing the *MIS* for the remaining graph after removing x active vertices and their incident edges, where x ranges from $|V_c|$ to $|V_c| - t + 1$ in the decreasing order (lines 5-10). If the maximum independent set obtained in a certain step equals $MIS(G_a)$, the *MCIS* is found. Hence, the time efficiency will be improved since the searching space can be reduced without exploring all the possible removals of active vertices. As the affected subgraph is very small, we can perform the exact algorithm, e.g., VCSolver [19], to compute the *MIS*.

Algorithm 5 *MCIS-Computation*

```
Input:   The affected subgraph $G_a$, active vertices $V_c$;
Output:  $MCIS(G_a)$.
 1: $MIS(G_a) \leftarrow$ computing the $MIS$ of $G_a$
 2: $t \leftarrow |MIS(G_a) \cap V_c|$
 3: if $t = 0$ then
 4:     return $MIS(G_a)$
 5: for $x$ from $|V_c|$ to $|V_c| - t + 1$ do
 6:     for each $x$ active vertices $V_x$ in $V_c$ do
 7:         $G'_a \leftarrow$ remove $V_x$ and their incident edges from $G_a$
 8:         $MIS(G'_a) \leftarrow$ computing the $MIS$ of $G'_a$
 9:         if $|MIS(G_a)| = MIS(G'_a)$ then
10:             return $MIS(G'_a)$
11: return $MIS(G_a)$
```

Actually, computing the exact *MCIS* suffers from high time complexity and may not be necessary in a real scenario. Therefore, we propose an efficient greedy algorithm. Similar to the exact algorithm above, we perform *Alg-MIS* over G_a to obtain $MIS(G_a)$. Then we try to "delete"[3] each active vertex u in $MIS(G_a)$ by invoking the algorithm in Section III-D. If we can obtain a new $MIS(G_a)'$ ($|MIS(G_a)'| = |MIS(G_a)|$) that includes less active vertices, a better *MCIS* is identified. The procedure proceeds until no better *MCIS* is generated.

C. Independent Set Integration

After obtaining $IS(G_a)$ or $MCIS(G_a)$, we need to integrate it with the independent set $IS(G_r)$. For ease of presentation, we only use $IS(G_a)$ to explain the integration. However, it also applies to $MCIS(G_a)$. If $IS(G_a) \cap V_c = \emptyset$, the two sets $IS(G_a)$ and $IS(G_r)$ can be combined together as the result of G_{i+1}, i.e., $IS(G_{i+1}) = IS(G_a) \cup IS(G_r)$; Otherwise, it indicates that there are some ***conflicting edges***, i.e., the edge $(u, v) \in G_{i+1}$, where $u \in IS(G_a) \cap V_c$ and $v \in IS(G_r)$. Note that some conflicting edges may be generated even for $MCIS(G_a)$ as there is no guarantee that $MCIS(G_a)$ does not contain any active vertices.

For each conflicting edge (u, v), we try to remove v out of $IS(G_r)$ by invoking the algorithm in Section III-D that deals with vertex removals. If we can find some vertices to replace the conflicting vertices in $IS(G_r)$, thus an independent set $IS(G_{i+1})$ of size $|IS(G_a)| + |IS(G_r)|$ will be delivered. Otherwise, we just remove the conflicting vertices from $IS(G_r)$.

In addition to the improvement on the solution quality by reducing the errors accumulated over single updates, batch update can reduce the time cost. Specifically, the time complexity

[3]After the searching, we need to recover the affected subgraph G_a by adding the deleted vertex and the corresponding edges to G_a.

TABLE I
Statistics of graphs

ID	Graph	$\|V(G)\|$	$\|E(G)\|$	$\bar{d}$
G01	Wiki-Vote	7,115	10,0762	28.32
G02	CA-HepPh	12,006	118,489	19.74
G03	AstroPh	18,771	198,050	21.1
G04	Brightkite	58,228	214,078	7.35
G05	Epinions	75,879	405,740	10.69
G06	Slashdot	82,168	504,230	12.27
G07	com-dblp	317,080	1,049,866	6.62
G08	com-amazon	334,863	925,872	5.53
G09	BerkStan	685,230	6,649,470	19.41
G10	web-Google	875,713	4,322,051	9.87
G11	soc-pokec	1,632,803	22,301,964	27.32
G12	as-skitter	1,696,415	11,095,298	13.08
G13	wiki-tomcats	1,791,489	25,444,207	28.41
G14	Wiki-Talk	2,394,385	4,659,565	3.89
G15	com-orkut	3,072,441	117,185,083	76.28
G16	cit-Patents	3,774,768	16,518,947	8.75
G17	com-lj	3,997,962	34,681,189	17.35
G18	LiveJournal	4,846,609	42,851,237	17.68

TABLE II
THE GAPS CAUSED BY 1000 UPDATES AND THE MEMORY USAGE

Graphs	*LSTwo+LazySearch*$^+$		*DGOneDIS*		*DGTwoDIS*		*DGOracleDIS*		*BatchUE*		*BatchUN*		Memory Usage (MB)		
	AGap	*RGap*	*AGap*	*RGap*	*AGap*	*RGap*	*AGap*	*RGap*	*AGap*	*RGap*	*AGap*	*RGap*	*DGOneDIS*	*DGTwoDIS*	*BatchUE*
Wiki-Vote	23	23	1	0	1	1	1	1	1	0	1	1	2.48	2.51	2.52
CA-HepPh	62	62	10	9	6	6	6	6	5	2	4	-5	2.99	3.04	3.76
AstroPh	35	35	12	11	5	5	3	3	7	7	5	5	4.98	5.05	5.14
Brightkite	17	17	2	1	1	1	1	1	1	0	1	1	6.29	6.51	6.76
Epinions	19	19	1	1	0	0	0	0	1	0	1	0	11.09	11.39	11.48
Slashdot	26	26	3	1	2	0	0	0	1	1	2	1	13.49	13.81	14.04
com-dblp	72	72	22	10	6	5	5	5	5	4	6	4	31.59	32.81	32.93
com-amazon	89	89	150	-2	21	-2	0	0	3	-5	4	-5	29.18	30.45	30.86
BerkStan	838	838	1074	5	761	4	18	18	15	0	12	3	168.53	171.15	172.28
web-Google	306	306	314	18	173	7	10	10	11	5	16	5	119.8	123.14	124.21
as-skitter	371	371	393	-2	295	0	0	0	7	0	14	-1	294.39	300.86	302.38
Wiki-Talk	74	74	1	0	1	0	0	0	1	0	1	1	163.74	172.87	173.52
LiveJournal	414	414	1458	4	388	-2	2	2	10	-2	18	-2	1096.34	1114.83	1115.12

of batch update is $O(\delta(G_a)+\gamma\cdot E(G))$, where $\delta(G_a)$ denotes the cost of computing $MIS(G_a)$ and γ is the number of conflicting edges. The time complexity of single updates is $O(k \cdot E(G))$, where k is the number of updating operations in a batch. Note that G_a is very small especially when the batch of updating operations occur over a small subgraph. Generally, k is much larger than γ as γ is related to G_a. Therefore, the time cost of batch update is less than that of single updates.

VI. EXPERIMENTS

A. Experimental Setting

Dataset. 18 real graphs are used to evaluate the algorithms. All of the graphs are downloaded from the Stanford Network Analysis Platform [35]. The statistics are summarized in Table I. Similar to [8], we randomly add/remove m vertices/edges to simulate the updating operations.

Algorithms. In this paper, we compare our algorithm with the state-of-the-art method *LSTwo+LazySearch*$^+$ [8] that computes independent sets for dynamic graphs. We implement the following algorithms and compare them with the competitor.

- *DGOneDIS*, *DGTwoDIS*, and *DGOracleDIS*: The *DG-one*, *DG-two*, and *DG-oracle* based algorithms that compute independent sets for dynamic graphs, respectively.
- *DGOneDY*, *DGTwoDY*, and *DGOracleDY*: The algorithms *DGOneDIS*, *DGTwoDIS*, and *DGOracleDIS* equipped with the dynamic searching strategy, respectively.
- *BatchUE* and *BatchUN* : The batch updating algorithm equipped with the *MCIS-Exact* and *MCIS-Near* algorithms.

All the experiments are conducted on a Windows Server 2008. The programmes above are implemented in C++.

B. Experimental Results

1) Comparison With State-of-the-art Algorithms:

Evaluate Solution Quality. Table II gives the results of these algorithms for handling 1000 updating operations, where *AGap* denotes the absolute gap, i.e., the gap between the output of an algorithm and the independence number (the size of the maximum independent set), and *RGap* denotes the relative gap, i.e., the gap between the output of an algorithm A and the result of the static algorithm that adopts the same reduction rules as A. For example, the *RGap* of *DGOneDIS* is *RGap*=*RESULT*[*BDOne*]-*RESULT*[*DGOneDIS*] as *BDOne* is the static algorithm that adopts the degree-one reduction rule as *DGOneDIS*, where *RESULT*[A] represents the size of independent set returned by algorithm A. Note that *AGap* = *RGap* for *LSTwo+LazySearch*$^+$ since it takes the exact maximum independent set as the input and does not adopt any reduction rules. It is clear that the proposed *DGOracleDIS* outperforms the competitor *LSTwo+LazySearch*$^+$ significantly in terms of both *AGap* and *RGap*. Although *DGOneDIS* and *DGTwoDIS* take the inexact independent sets as inputs, they can find solutions of higher quality than *LSTwo+LazySearch*$^+$ on most datasets. Moreover, we can find that *DGTwoDIS* is more effective than *DGOneDIS* because it adopts both degree-one and degree-two reduction rules. More interestingly, *BatchUE* and *BatchUN* find larger independent sets than the other algorithms, which indicates that taking the updating operations as a whole is very promising to improve the solution quality, but they show similar performance. On some datasets *BatchUN* is better than *BatchUE*. Hence, it may not be necessary to find the exact $MCIS$.

The relative precision of the solution can be computed as $RGap/\lambda$, where λ represents the size of the independent set returned by the corresponding static algorithm. The relative precisions of the proposed algorithms over all the datasets are all larger than 99.46%. In other words, the proposed algorithms can maintain the solution quality very well. Furthermore, *RGap* is negative on some datasets, e.g., com-amazon, as-skitter, and LiveJournal, which illustrates that the quality of the solutions returned by the proposed algorithms may be higher than the solution quality of the static algorithms.

Evaluate Time Efficiency. To study the time efficiency of these methods, we report the response time consumed by handling 1000 updating operations as shown in Table III. All the proposed methods in this paper outperform the previous algorithm *LSTwo+LazySearch*$^+$ by almost two orders of magnitude. *DGOneDIS* is more efficient than *DGTwoDIS* as its dependency graph is not larger than that of *DGTwoDIS* and the searching space is smaller correspondingly. Clearly,

TABLE III
Visited vertices on average and response time consumed by 1000 updates

Graphs	*LSTwo+LazySearch*$^+$	*DGOneDIS*		*DGTwoDIS*		*DGOracleDIS*		*BatchUE*		*BatchUN*		*DGOneDY*		*DGTwoDY*		*DGOracleDY*	
	ms	#	*ms*	#	*ms*	#	*ms*	#	*ms*	#	*ms*	#	*ms*	#	*ms*	#	*ms*
Wiki-Vote	19.3	1.1	1.3	5.6	7.2	3.6	4.5	1.3	1.5	0.28	0.3	1.3	1.5	5.5	7.1	3.9	4.6
CA-HepPh	36.6	2.7	3.2	3.7	4.6	4.4	5.4	1.7	1.9	0.78	0.9	2.6	3.3	4.2	4.8	4.9	5.7
AstroPh	63.7	1.4	1.5	2.8	3.3	5.8	6.9	0.76	0.9	0.31	0.4	1.3	1.5	3.1	3.7	6.7	7.1
Brightkite	104.6	3.6	4.1	3.9	4.5	7.3	8.3	1.8	2.2	0.91	1.1	4.0	4.6	3.8	4.5	6.9	7.7
Epinions	72.1	0.92	1.2	1.4	1.7	5.2	6.6	0.71	0.85	0.28	0.33	1.1	1.4	1.2	1.8	5.1	6.3
Slashdot	166.8	1.8	2.3	2.3	2.8	7.5	9.3	0.79	0.9	0.17	0.2	1.9	2.1	2.2	2.7	7.4	8.4
com-dblp	88.7	1.1	1.4	1.7	1.9	3.2	4.2	0.56	0.66	0.29	0.41	1.3	1.6	2.1	2.5	3.1	4.2
com-amazon	1209.3	2.1	2.6	2.2	2.7	4.3	5.1	1.2	1.4	0.41	0.52	2.2	2.8	2.3	2.9	4.3	5.6
BerkStan	2723.4	6.4	7.7	7.5	8.6	8.9	10.5	2.8	3.3	1.3	1.5	5.8	7.2	6.1	7.8	8.2	9.6
web-Google	4045.8	4.7	5.4	5.6	6.3	7.7	9.3	2.2	2.6	0.64	0.72	4.3	5.5	5.2	6.5	8.6	9.7
as-skitter	3659.3	7.2	8.6	8.2	9.8	9.6	11.5	2.8	3.4	0.26	0.29	7.1	8.2	7.9	9.1	8.9	10.8
Wiki-Talk	2325.5	0.9	1.1	1.4	1.6	4.7	5.8	0.55	0.62	0.1	0.11	1.2	1.5	1.4	1.8	5.6	6.4
LiveJournal	7521.4	4.2	5.3	12.9	15.2	10.1	12.4	1.1	1.6	0.29	0.39	4.6	5.5	12.9	14.8	9.5	11.2

instead of dealing with the updating operations individually, the two algorithms *BatchUE* and *BatchUN* reduce the time cost by adopting the batch-mode technique. Computing the exact maximum constraint independent set takes more time. Hence, *BatchUN* is the most efficient among these algorithms. The number of visited vertices on average is also reported, which confirms that Algorithm 3 is very efficient.

Evaluate Memory Cost. The memory consumed by the index of each proposed algorithm is listed in Table II. *DGTwoDIS* takes more space than *DGOneDIS*. The reason is that *DGTwoDIS* maintains more information (vertices and edges) than *DGOneDIS*. The space cost of *BatchUE* is almost the same as *BatchUN* (we only present the space cost of *BatchUE* due to the space limitation), and both of them demand a little more space than *DGTwoDIS* to compute the exact or near *MCIS*.

Effect of Initial Input Quality. We vary the initial solution quality, based on which the dependency graphs are constructed. Generally, following the constructing procedure as described in Section IV-A we can build a dependency graph according to any given independent set. Let θ represent the quality of the initial solution $|IS(G_0)|$, i.e., $\theta = |IS(G_0)|/|MIS(G_0)|$. Table IV depicts the absolute precision (i.e., $AGap/MIS(G)$) of *DGOneDIS* and *DGTwoDIS* for dealing with 1000 updates, where θ is varied from 50% to 90%. It is clear that the quality of the delivered solutions increases slightly when the quality of the initial input gets better. Notice that the answer quality remains high enough (above 99.9% on most datasets) even if θ = 50%. Thus the proposed algorithm is insensitive to the initial input quality.

2) Evaluate Dynamic Search: To guarantee the time cost of checking whether a vertex is valid or not within $O(|E(G)|)$ in the worst case, a dynamic searching strategy based on bottom-up search is devised in Section IV. In order to evaluate the effect of the dynamic searching strategy, we incorporate it into the algorithms *DGOneDIS*, *DGTwoDIS*, *DGOracleDIS*, and *BatchUN* (they are also called the top-down algorithms) that take Algorithm 3 as the core to form the algorithms *DGOneDY*, *DGTwoDY*, and *DGOracleDY*. As presented in Table III, the algorithms equipped with the dynamic searching strategy do not outperform the top-down algorithms on most datasets, which confirms that Algorithm 3 is very efficient in the real scenario and it has little chance to traverse the whole

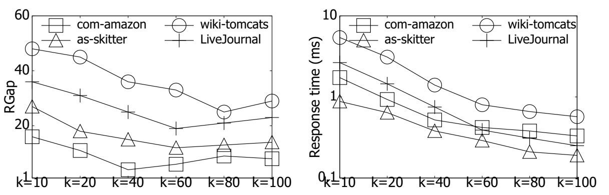
TABLE IV
Effect of initial input quality (*DGOneDIS* and *DGTwoDIS*)

Graphs	*DGOneDIS*			*DGTwoDIS*		
	θ=0.50	θ=0.75	θ=0.90	θ=0.50	θ=0.75	θ=0.90
Wiki-Vote	99.91%	99.94%	99.97%	99.97%	99.97%	99.98%
AstroPh	99.86%	99.93%	99.95%	99.95%	99.95%	99.96%
Brightkite	99.46%	99.86%	99.99%	99.99%	99.99%	99.99%
Slashdot	99.94%	99.97%	99.99%	99.99%	99.99%	100%
com-amazon	99.86%	99.94%	99.97%	99.97%	99.99%	99.99%
BerkStan	98.67%	98.93%	99.21%	98.75%	99.33%	99.44%
web-Google	99.73%	99.88%	99.93%	99.91%	99.98%	99.99%
as-skitter	98.96%	99.63%	99.79%	99.67%	99.86%	99.93%
Wiki-Talk	99.91%	99.94%	100%	100%	100%	100%
LiveJournal	99.89%	99.92%	99.96%	99.98%	99.99%	99.99%

graph G as analyzed in Section III-C.

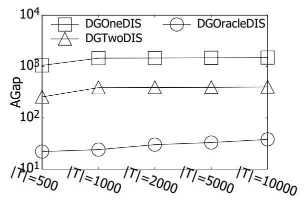
3) Evaluate Batch Updates: In this section, we evaluate the effect of the number of updating operations in each batch, denoted by k. Fig. 6(a) and Fig. 6(b) describe the effect on the solution quality and time efficiency, respectively, where k is varied from 10 to 100. As shown in Fig. 6(a), with the growth of k, *RGap* decreases first and then increases when k becomes larger. That is because more active vertices will be generated if there are more updating operations in each round of update. Fig. 6(b) shows that the response time decreases as k grows, which confirms the effectiveness of the batch-mode algorithm.

4) Evaluate Scalability: To study the scalability of the proposed methods, the number of updating operations (denoted by $|T|$) is varied from 500 to 10,000. Fig. 7(a) and Fig. 7(b) show the effect of $|T|$ on the quality and time efficiency (over the graph LiveJournal), respectively. It is clear that decreasing rate of the performance is near linear to the amount of update operations. Fig. 7(c) and Fig. 7(d) depict the results of *DGTwoDIS* over different datasets. It shows that the quality

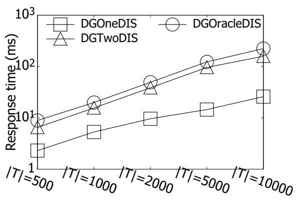


(a) *RGap vs. k* (b) Response time *vs. k*

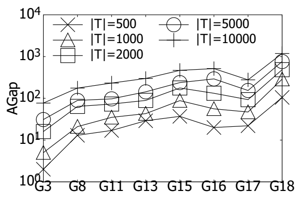
Fig. 6. Effect of the batch size



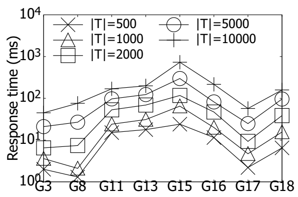
(a) *AGap vs.* $|T|$



(b) Response time *vs.* $|T|$



(c) *DGTwoDIS*: *AGap*



(d) *DGTwoDIS*: time efficiency

Fig. 7. Scalability evaluation

decreases as we increase the number of update operations. Moreover, the decreasing ratio is near linear to the number of updating operations. As presented in Fig. 7(d) more time is consumed to handle more updates. The algorithm performs good scalability as it takes less than 1 second on each dataset even for 10,000 updates.

VII. Conclusions

In this paper, we study the problem of computing the high-quality (may not be the maximum) independent sets over the graphs that are dynamically changing. Based on the two widely used degree-one and degree-two reduction rules, we devise a very effective index, i.e., dependency graph, that can facilitate the computation of independent sets. In order to guarantee the time complexity $O(|E(G)|)$, a dynamic search algorithm is proposed based on the bottom-up searching strategy. Moreover, we present an novel method to handle batch updates for computing independent sets over evolving graphs. Extensive experiments over a wide range of graphs confirm that the proposed methods are both effective and 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], [Project No.: CUHK 14221716] and [Project No.: CUHK 14203618].

References

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

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

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

[4] L. Sun, R. Cheng, X. Li, D. W. Cheung, and J. Han, "On link-based similarity join," *PVLDB*, vol. 4, no. 11, pp. 714–725, 2011.

[5] W. Zheng, L. Zou, L. Chen, and D. Zhao, "Efficient simrank-based similarity join," *ACM TODS*, vol. 42, no. 3, pp. 16:1–16:37, 2017.

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

[7] 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.

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

[9] P. Wan, B. Xu, L. Wang, S. Ji, and O. Frieder, "A new paradigm for multiflow in wireless networks: Theory and applications," in *2015 IEEE Conference on Computer Communications, INFOCOM 2015, Kowloon, Hong Kong, April 26 - May 1, 2015*, 2015, pp. 1706–1714.

[10] E. J. Gardiner, P. Willett, and P. J. Artymiuk, "Graph-theoretic techniques for macromolecular docking," *Journal of Chemical Information & Computer Sciences*, vol. 40, no. 2, p. 273, 2000.

[11] 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.*, pp. 1356–1366, 2011.

[12] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li, "New algorithms for fast discovery of association rules," in *International Conference on Knowledge Discovery and Data Mining*, 1997, pp. 283–286.

[13] 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.

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

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

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

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

[18] 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.

[19] 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.

[20] 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.

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

[22] 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 -SEA*, 2016, pp. 118–133.

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

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

[25] M. M. Halldórsson and J. Radhakrishnan, "Greed is good: Approximating independent sets in sparse and bounded-degree graphs," *Algorithmica*, vol. 18, no. 1, pp. 145–163, 1997.

[26] J. Håstad, "Clique is hard to approximate within $n^{1\text{-epsilon}}$," in *FOCS*, 1996, pp. 627–636.

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

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

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

[30] 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.

[31] A. Grosso, M. Locatelli, and W. J. Pullan, "Simple ingredients leading to very efficient heuristics for the maximum clique problem," *J. Heuristics*, vol. 14, no. 6, pp. 587–612, 2008.

[32] 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.

[33] 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.

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

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