Maximizing the Reduction Ability for Near-maximum Independent Set Computation

Chengzhi Piao[†], Weiguo Zheng[‡], Yu Rong[§], Hong Cheng[†]
[†]The Chinese University of Hong Kong
[‡]Fudan University
[§]Tencent AI Lab
[†] {czpiao,hcheng}@se.cuhk.edu.hk, [‡] zhengweiguo@fudan.edu.cn, [§] yu.rong@hotmail.com

ABSTRACT

Finding the maximum independent set is a fundamental NP-hard problem in graph theory. Recent studies have paid much attention to designing efficient algorithms that find a maximal independent set of good quality (the more vertices the better). Kernelization is a widely used technique that applies rich reduction rules to determine the vertices that definitely belong to the maximum independent set. When no reduction rules can be applied anymore, greedy strategies including vertex addition or vertex deletion are employed to break the tie. It remains an open problem that how to apply these reduction rules and determine the greedy strategy to optimize the overall performance including both solution quality and time efficiency. Thus we propose a scheduling framework that dynamically determines the reduction rules and greedy strategies rather than applying them in a fixed order. As an important reduction rule, degree-two reduction exhibits powerful pruning ability but suffers from high time complexity $O(nm)$, where n and m denote the number of vertices and edges respectively. We propose a novel data structure called representative graph, based on which the worst-case time complexity of degree-two reduction is reduced to $O(m \log n)$. Moreover, we enrich the naive vertex addition strategy by considering the graph topology and develop efficient methods (active vertex index and lazy update mechanism) to improve the time efficiency. Extensive experiments are conducted on both large real networks and various types of synthetic graphs to confirm the effectiveness, efficiency and robustness of our algorithms.

PVLDB Reference Format:
Chengzhi Piao, Weiguo Zheng, Yu Rong, and Hong Cheng. Maximizing the Reduction Ability for Near-maximum Independent Set Computation. *PVLDB*, 13(11): 2466-2478, 2020.
DOI: https://doi.org/10.14778/3407790.3407838

1. INTRODUCTION

A wide variety of real-world data can be represented as graphs, such as drug molecule [14], social networks and knowledge graphs. Graph problems have attracted much attention these years, e.g., shortest path queries [2], reachability computation [22], subgraph matching [5] and maximum clique [16]. Finding a maximum independent set (MIS in short) is a fundamental NP-hard problem in graph theory [11]. An independent set of a graph G denotes a vertex set in which every two vertices are not adjacent, i.e., vertices u and v cannot be included in the same independent set if G contains the edge (u, v). The MIS is an independent set that has the largest number of vertices. It is extremely challenging since finding an approximate solution within $n^{1-\varepsilon}$ for any constant $0 < \varepsilon < 1$ is NP-hard as well [12].

Applications. The maximum independent set can be applied in many real-world applications, e.g., collusion detection [3], social network coverage computation [19], wireless network organization [4] and association rule mining [18]. It is also commonly used as one step of other graph algorithms, e.g., clustering [9] and k-hop reachability query [7]. A recent study [24] has proposed a novel potential application – eliminating duplicate paths on a condensed representation of graph. Assume that there are two tables $TaughtCourse(\underline{tid, cid})$ and $TookCourse(\underline{sid, cid})$ in a relational database where *tid*, *cid* and *sid* denote the ID of teachers, courses and students respectively. The target is to construct a bipartite graph between teachers and students where each edge represents a teaching relationship, i.e., a teacher and a student are in the same course. It can be formulated as a SQL query: SELECT tid, sid FROM TaughtCourse, TookCourse WHERE TaughtCourse.cid = TookCourse.cid. Let us consider the example as shown in Figure 1. Compared to the target Teacher-Student graph, the 3-layer Teacher-Course-Student graph contains enough information but fewer edges, since each course represents a full connection between its teachers and students on the target graph. Therefore, the 3-layer graph is regarded as a condensed representation. However, it may still contain duplicate edges. For example, the two green paths in the 3-layer graph correspond to the same edge in the target bipartite graph. To eliminate such duplication, some of the middle vertices (courses) must be expanded back to the full connection, resulting in a combination of the 3-layer graph and the bipartite graph. In other words, two middle vertices cannot be kept at the same time if they share a pair of teacher and student, which corresponds to the independent set problem in essence.

Existing Algorithms. The existing algorithms are categorized as exact algorithms [23, 1, 10] and inexact (including approximate and greedy) algorithms [6]. The former aims

Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407838

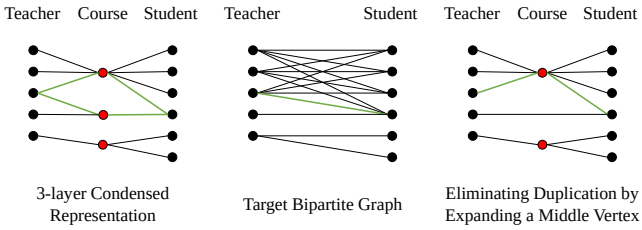

Figure 1: A Scenario of MIS Computation

to find an exact MIS. The latter delivers a near-maximum one (within an approximation ratio or not) but is required to consume much less time. In the latter case, the size of an independent set is used to measure the solution quality, the larger the better. Hence, each MIS is an optimal solution. The existing techniques can be divided into three categories:
1) Kernelization. Construct a smaller kernel graph based on some reduction rules, as long as the MIS on the original and kernel graphs is equivalent. Therefore, it helps reduce the problem size without sacrificing the solution quality. Many useful reduction rules have been proposed in previous studies, such as the folding and mirroring rule in [10] and the degree-two path reduction rule in [6].
2) Branch and Bound. Branch and bound methods obtain exact solutions by exploring all possible cases. Each vertex u can be either included in the independent set or not [10]. As for an edge (u, v), there are three cases, picking u, v or neither of them [23].
3) Greedy Strategy. Based on the fact that including a vertex into the independent set leads to discarding all its neighbors, a vertex that has more neighbors is less likely to be included in an MIS. Hence, selecting the smallest-degree vertex [15] and discarding the largest-degree vertex [6] are two commonly used greedy strategies.

Actually, most of the existing algorithms can be regarded as the combination of the above techniques. The state-of-the-art exact algorithm [23] runs in $O(1.1996^n n^{O(1)})$ time and polynomial space consumption by branching on edges and largest-degree vertices. The state-of-the-art inexact algorithm [6] applies the reducing and peeling framework, which combines kernelization with greedy vertex deletion to find a near-maximum independent set in near-linear time. However, two questions have not been addressed:
Q1: How to evaluate and apply these reduction rules to maximize the reduction ability? Currently, there have been many reduction rules, e.g., degree-one reduction [10], degree-two reduction [8, 10], and dominance reduction [6]. Generally, it is difficult to conclude that reduction R_1 is better than R_2 even if R_2 is dominated by R_1, and vice versa. A reduction is more efficient to compute but may exhibit a limited reduction ability. For instance, the degree-two reduction rule repeatedly contracts the neighbors of a degree-two vertex, which results in the time cost $O(nm)$ in the worst case. It may be unacceptable on large graphs. The degree-two path reduction rule proposed in [6] is actually a special case of the degree-two reduction rule, but it is extremely useful due to its linear time complexity. Traditionally, these reduction rules are applied in a fixed order: the simple reductions are adopted first. However, it may degrade the time efficiency since a useless rule can be repeatedly examined.

Instead of applying these reduction rules in a fixed order, we propose a novel scheduling framework to dynamically apply the reduction rules based on the current graph.

Q2: How to determine the greedy strategy and cooperate with the reduction rules? When no reduction rules can be applied, the greedy strategies are invoked to break the tie. It is known that there have been two kinds of greedy strategies available – vertex addition and vertex deletion. The existing addition strategy adds the vertex of the smallest degree into the independent set. However, it may degrade the solution quality, since the large number of smallest-degree vertices makes it hard to include the correct one.

Therefore, we propose a more powerful vertex addition strategy. Furthermore, these greedy strategies are expected to be determined through the scheduling framework rather than roughly specified.

In summary, we make four contributions in this paper:
1) A novel scheduling framework is devised to dynamically apply different reduction rules and greedy strategies. In this framework, we evaluate the reduction rules by a cost-and-benefit estimation and determine the appropriate greedy strategy according to the structure of the current graph.
2) We propose a novel notion **representative graph**, based on which the worst-case time complexity of degree-two reduction is reduced from $O(nm)$ to $O(m \log n)$.
3) We design a more powerful vertex addition strategy that takes the degree of neighbors into consideration. Two techniques **active vertex index** and **lazy update mechanism** are developed to improve the time efficiency.
4) Extensive experiments are conducted on both large real networks and various types of synthetic graphs. The results confirm that our proposed algorithms **Sch-standard** and **Sch-extension** outperform the state-of-the-art near-linear time algorithm **NearLinear** [6] in terms of both solution quality and time efficiency on most graphs.

2. PRELIMINARY AND OVERVIEW

2.1 Problem Definition

In this paper, we focus on finding a near-maximum independent set on an unweighted undirected graph $G = (V, E)$, where V and E denote the vertex set and edge set respectively. $N(u)$ denotes vertex u's neighbor set and $N[u] = N(u) \cup \{u\}$. We use $d(u)$ to denote the degree of vertex u, and we have $d(u) = |N(u)|$. Moreover, n and m are used to denote $|V|$ and $|E|$ in the time complexity analysis.

Definition 1. **Independent Set.** Given a graph $G = (V, E)$, a set of vertices $I \subset V$ forms an independent set of G if it holds that $\forall u, v \in I, (u, v) \notin E$.

Definition 2. **Maximum Independent Set (MIS).** An independent set I that has the largest number of vertices among all independent sets of G.

Generally, a graph may contain many independent sets, among which the one with the largest number of vertices is called the maximum independent set.

Problem Statement. Given a graph $G = (V, E)$, we aim to find a near-maximum independent set (Near-MIS for short) as large as possible at very low time cost.

2.2 Reduction and Greedy Algorithms

As computing an MIS over a large graph suffers from expensive computation cost, several reduction rules [10, 1, 8] have been proposed to reduce the size of the graph without sacrificing the solution quality.

2.2.1 Reduction Rules

If there is a vertex u whose degree is at most two in the graph G, then one of the following rules works.

Definition 3. **Degree-one Reduction** [10] ($d \leq 1$).

$$|MIS(G \backslash N[u]) \cup \{u\}| = |MIS(G)|$$

Definition 4. **Degree-two Reductions** [8, 10] ($d = 2$).

- Isolation Rule: If the vertices in $N[u]$ form a triangle, $|MIS(G \backslash N[u]) \cup \{u\}| = |MIS(G)|$;
- Folding Rule: If the vertices in $N[u]$ do not form a triangle, we could contract $N[u]$ as a vertex x and denote the resulting graph as G'. Then it holds that when $x \in MIS(G')$, $|(MIS(G') - \{x\}) \cup N(u)| = |MIS(G)|$; otherwise, $|MIS(G') \cup \{u\}| = |MIS(G)|$.

These reductions consider vertices whose degree is smaller than 3. Thus they are called low-degree reductions. Different from them, another reduction rule, vertex dominance reduction, is more likely to discard high-degree vertices.

Definition 5. **Vertex dominance**. For two arbitrary different vertices u and v, u *dominates* $v \iff N[u] \subseteq N[v]$.

Definition 6. **Dominance Reduction** [6]. If v is dominated by any other vertex, then $|MIS(G \backslash \{v\})| = |MIS(G)|$.

2.2.2 Reduction Algorithms

Based on these reduction rules, immediately we have the following two reduction algorithms.

Algorithm 1 simply tries to apply the dominance reduction rule on each vertex in the non-ascending order of degree. In line 3, actually we could skip the neighbors whose degree is larger than $d(u)$. Therefore, the time complexity of Algorithm 1 is $O(\sum_{(u,v)\in E} \min\{d(u), d(v)\})$. This algorithm cannot guarantee to delete all dominated vertices, as it does not apply the dominance reduction rule recursively.

The low-degree reduction algorithm is presented in Algorithm 2. To check whether there exists a degree-one or degree-two vertex, there is no need to go through all vertices in each loop. Instead, we could use two queues Q_1 and Q_2 to maintain vertices whose degree is no greater than 1 and equal to 2 respectively. Therefore, when we update the graph (e.g., deleting a vertex), it is necessary to push the new low-degree vertices into Q_1 and Q_2 immediately.

The time complexity of Algorithm 2 is $O(nm)$ according to the analysis in [6], and it also proves a lower bound, $\Omega(n \log n + m)$. Using a traditional linear data structure, e.g., adjacency list, the contraction operation takes the time cost $O(d(u) + d(v))$ to traverse $N(u)$ and $N(v)$. A huge gap still remains between $\Omega(n \log n + m)$ and $O(nm)$, which brings us two challenging questions:
1) Can we reduce the time complexity of contraction to $O(\min\{d(u), d(v)\})$ without affecting other operations?
2) If yes, can we derive an upper bound better than $O(nm)$?
These questions will be addressed in Section 3.

Algorithm 1: Dominance Reduction

Input : a graph $G = (V, E)$
Output: the remaining graph G after reduction

```
1 Sort all vertices in non-ascending order of degree;
2 for each vertex u ∈ V in the above order do
3 |   if u is dominated by any neighbor then
4 |   |   delete u from G;
5 return G
```

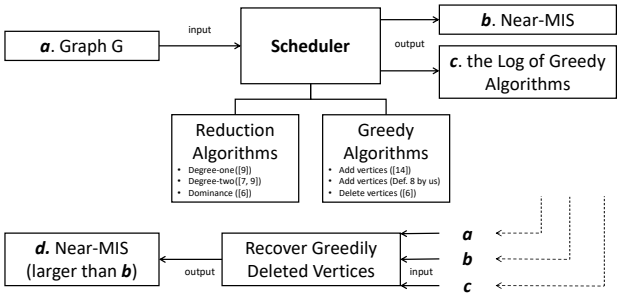

Figure 2: Overview of Our Approach

2.2.3 Greedy Algorithms

When no reduction rules can be applied, the greedy algorithms will be invoked until the reduction rules can be used. There are mainly two strategies, adding the vertex of the smallest degree into the independent set [15] and deleting the vertex of the largest degree [6].

2.3 Overview of Our Approach

Figure 2 shows the overview of our scheduler framework. Basically, it consists of two parts, i.e., Near-MIS computation (scheduler) and Near-MIS recovery.

Different from existing algorithms that apply the reduction rules sequentially and adopt one of the greedy strate-

Algorithm 2: Low-degree Reduction

Input : a graph $G = (V, E)$
Output: independent set I, the remaining graph G

```
1  for each vertex u ∈ V do
2  |   if d(u) ≤ 1 then Q1.push(u);
3  |   else if d(u) = 2 then Q2.push(u);
4  while True do
5  |   if !Q1.empty() then          /* d1 Reduction */
6  |   |   pop a vertex u from Q1;
7  |   |   if !del[u] and d(u) ≤ 1 then
8  |   |   |   add u into I and then delete N[u] from G;
9  |   else if !Q2.empty() then    /* d2 Reduction */
10 |   |   pop a vertex u from Q2;
11 |   |   if del[u] or d(u) ≠ 2 then continue;
12 |   |   let v and w be the two neighbors of u;
13 |   |   if (v, w) ∈ E then      /* Isolation Rule */
14 |   |   |   add u into I and then delete N[u] from G;
15 |   |   else                    /* Folding Rule */
16 |   |   |   temporarily add u into I;
17 |   |   |   delete u from G;
18 |   |   |   contract v and w as a super vertex with
   |   |   |     the contraction information (u, v, w);
19 |   else break;
20 trace back all super vertices:
21 if a super vertex (u, v, w) is in I then
22 |   replace it by v and w, and remove u from I;
23 return I and G
```

Table 1: RGraph vs. Adjacency List

Time Complexity	add an edge (u, v)	delete a vertex u	contract vertices u, v	Low-degree Reduction Algorithm
RGraph	$O(1)$	$O(d(u))$	$O(\min\{d(u), d(v)\})$	$\Theta(m \log n)$
Adjacency List	$O(1)$	$O(d(u))$	$O(d(u) + d(v))$	$O(nm)$

gies, we devise a scheduler to determine the techniques including both reduction rules and greedy strategies dynamically according to the graph in each step. Specifically, the scheduler receives a graph G as input (as shown in block a), and then dynamically schedules the reduction and greedy algorithms. After iteratively applying these algorithms, the size of graph G gradually decreases to zero and we get a Near-MIS as output (as shown in block b). As greedy algorithms take full responsibility for the gap between the output Near-MIS and an MIS, the scheduler also records the log of using greedy algorithms (as shown in block c). Combining blocks a, b and c, we try to debug the mistakes made by greedy algorithms and finally recover a much larger Near-MIS (as shown in block d).

3. REPRESENTATIVE GRAPH

In this section, we introduce a new graph data structure, representative graph (RGraph in short), to speed up the low-degree reduction algorithm.

Adjacency lists do not support contraction efficiently as duplicate vertices may be generated if two linked lists $N(u)$ and $N(v)$ are spliced directly. Here, duplicate vertices denote the co-neighbors of u and v as each co-neighbor appears twice in the splicing list. To remove these duplicate vertices, it usually takes $O(d(u) + d(v))$ time to traverse both adjacency lists. Otherwise, many graph operations are restricted since the degree of a vertex is not even known. Nevertheless, in the low-degree reduction algorithm the exact degree of a vertex is useless unless it decreases to 2 or less. This property motivates us to deal with duplicate vertices in a more flexible way so as to accelerate the contraction operation without degrading other operations.

Specifically, we design a novel structure RGraph, where each vertex's neighbors are divided into two parts, **representative neighbors** and **backup neighbors**. A major difference between them is that the latter may contain duplicate vertices but the former not. When a vertex u is deleted, denoted by **deleted vertex**, it is inefficient to remove it directly from each of its neighbor's adjacency lists. One better method is using a Boolean array $del[n]$ to record which vertices have been deleted, i.e., $del[v] = True \iff v$ has been deleted. With the support of this array, deleted vertices are allowed to stay in the backup neighbor lists.

Algorithm 3 shows the abstract data type definition of RGraph. Instead of being involved in the low-degree reduction algorithm directly, it provides interfaces in the graph operation level. Table 1 shows the time complexity comparison between operations on RGraph and adjacency list.

3.1 Component and Property

For each vertex u, $N(u)$ is divided into **representative neighbors** (R-neighbors in short) and **backup neighbors** (Bak-neighbors in short), stored in two linked lists and denoted by $N_r(u)$ and $N_{bak}(u)$ respectively.

$N_r(u)$ is a subset of $N(u)$ and it holds that:

- **Size Constraint:** Each vertex has no more than three R-neighbors, i.e., $|N_r(u)| \leq 3$.
- **Clean:** For each vertex u, $N_r(u)$ does not contain duplicate or deleted vertices.
- **Maximal:** Each vertex should have R-neighbors as many as possible within the size constraint.

Lemma 1 reveals the relation between $|N_r(u)|$ and $d(u)$. If a vertex u has less than 3 neighbors, $N_r(u) = N(u)$ holds; Otherwise, $N_r(u)$ consists of 3 distinct vertices arbitrarily chosen from $N(u)$.

LEMMA 1. *If $|N_r(u)| = 3$, it indicates that $d(u) \geq 3$; Otherwise, $d(u) = |N_r(u)|$.*

In contrast, $N_{bak}(u)$ stores the remaining neighbors of u in a **lazy** way. Specifically, it may contain duplicate vertices and deleted vertices. When a neighbor v of u is deleted, we need to remove v from $N_r(u)$ at once if it is a R-neighbor; Otherwise, no operation is required.

3.2 Core Operation of RGraph: Touch

As aforementioned, RGraph consists of two linked lists for each vertex. One major challenge is how to efficiently update the graph for the reduction or peeling operations while maintaining the above properties of R-neighbors.

As shown in Algorithm 3 we propose a **touch** operation to connect high level graph operations and the bottom implementation of linked lists. Specifically, **touch(u)** is in charge of keeping $N_r(u)$ clean and maximal after the updating operations (e.g., deleting a vertex). It first checks whether any R-neighbor is deleted and removes the deleted ones (lines 6-7

Algorithm 3: ADT Definition of RGraph

Vertex_Type: (N_r, N_{bak})

```
1  Procedure init(u)
2      for v ∈ N(u) do
3          if |N_r(u)| < 3 then add v into N_r(u);
4          else add v into N_bak(u);
5  Procedure touch(u)
6      for v ∈ N_r(u) do
7          if del[v] then delete v from N_r(u);
8      while |N_r(u)| < 3 and |N_bak(u)| > 0 do
9          pop a vertex v from N_bak(u);
10         if !del[v] and v ∉ N_r(u) then
11             add v into N_r(u)
12 Procedure add_edge(u, v)
13     add u into N_bak(v) and then touch(v);
14     add v into N_bak(u) and then touch(u);
15 Procedure delete(u)
16     del[u] ← True;
17     for v ∈ N_r(u) ∪ N_bak(u) do
18         if !del[v] then touch(v);
19 Procedure contract(u, v)
20     if |N_bak(u)| < |N_bak(v)| then swap u and v;
21     del[v] ← True;
22     for w ∈ N_r(v) ∪ N_bak(v) do
23         if !del[w] then add_edge(u, w);
```

in Algorithm 3). Then it tries to extend $N_r(u)$ to be maximal by promoting some Bak-neighbors (lines 8-11). Notice that we just promote Bak-neighbors which meet the clean requirement and discard the others (do not put back to $N_{bak}(u)$). Theorem 1 shows the time complexity.

THEOREM 1. *If the function touch() is called k times in the low-degree reduction algorithm, the total time complexity of the touch operations is $O(k+m)$.*

The remaining part of Algorithm 3 presents three graph operations based on the touch operation.

Lines 12-14: Add an edge between u and v. Add u and v into each other's Bak-neighbors and then touch them to maintain the index. The time cost is $O(1)$.

Lines 15-18: Delete u and update $N(u)$. When u is deleted, it is necessary to touch each neighbor of u to find all newly generated low-degree vertices in $O(d(u))$ time.

Lines 19-23: Contract two vertices u and v. Let v denote the vertex with less Bak-neighbors. If not, swap u and v first in line 20. To contract u and v as a super vertex, we delete v and pass $N(v)$ to u (i.e., add edges between u and v's neighbors), so that u becomes the super vertex. It is worth noting that the set of newly generated low-degree vertices is a subset of $N(u) \cap N(v)$. Hence, updating either $N(u)$ or $N(v)$ is enough. $N(v)$ is chosen to ensure the time complexity is $O(d(v)) = O(\min\{d(u), d(v)\})$, and each neighbor w of v is touched in **add_edge**(u, w).

In conclusion, the contraction operation can be conducted at the cost of $O(\min\{d(u), d(v)\})$ by using RGraph. Moreover, the time complexity of other updating operations remains unchanged.

3.3 Time Complexity Analysis

THEOREM 2. *The worst-case time complexity of RGraph-based low-degree reduction algorithm is $\Theta(m \log n)$.*

Chang et al. [6] present a low-degree reduction algorithm with the time complexity $\Omega(n \log n + m)$, where Ω denotes a lower bound of the time complexity. It constructs a graph instance with only $\Theta(n)$ edges on which the algorithm runs in $\Theta(n \log n)$ time. Actually, it is quite easy to make an extension: if we construct the graph instance with $\Theta(m)$ edges, the algorithm will run in $\Theta(m \log n)$ time. It means that we find a tighter lower bound $\Omega(m \log n)$.

In the following, we prove that the upper bound is also $O(m \log n)$. One basic observation is that the total time cost on deleting vertices can be covered by $O(m)$. As the contraction operation dominates the algorithm, we just need to consider vertex contraction rather than vertex deletion.

Simple Merge Model. Assume that there are m apples in n boxes. A non-negative integer d_i ($i \in \{0, 1, \ldots, n-1\}$) denotes the number of apples in box_i. We try to put all apples in one box in $n-1$ steps. In each step, we can arbitrarily choose two boxes box_i and box_j to merge. Then the merged box contains $d_i + d_j$ apples and the total number of boxes decreases by one. After $n-1$ steps, there is only one box left which contains all of the m apples. Moreover, we define the cost of each step as $\min\{d_i, d_j\}$ and use $f(n, m)$ to represent the total cost.

LEMMA 2. $f(n, m) = O(m \log n)$.

PROOF. Each apple in the box containing fewer apples results in a cost of 1. In other words, if an apple contributes a cost of 1, the number of apples in its box will at least double. Therefore, the cost resulted from a box of x apples is at most $x \lfloor \log_2 \frac{m}{x} \rfloor$. Immediately, we have:

$$f(n, m) \leq \sum_{i=0}^{n-1} d_i \left\lfloor \log_2 \frac{m}{d_i} \right\rfloor \leq \sum_{i=0}^{n-1} d_i \log_2 \frac{m}{d_i}$$

It is easy to prove that the right side gets maximized when all d_i's are equal, i.e., $d_0 = d_1 = \ldots = d_{n-1} = \frac{m}{n}$. Then we have $f(n, m) \leq m \log_2 n = O(m \log n)$. □

Now we can use the simple merge model to analyse the low-degree reduction algorithm.

1) Each vertex u is regarded as a box. The number of apples in the box is set as u's degree $d(u)$.

2) When box_u and box_v are merged, the merged box contains $d(u)+d(v)$ apples at the cost of $\min\{d(u), d(v)\}$. It exactly fits the situation of contracting two vertices on RGraph.

3) Although duplicate and deleted neighbors are allowed in N_{bak} due to the lazy storage manner, some of them may be removed during the update. By comparison, the setting of the simple merge model considers the worst case, that no apple is thrown away.

The simple merge model describes the worst case of running low-degree reduction algorithm on RGraph. According to Lemma 2, $f(n, m)$ is an upper bound of the time complexity. Therefore, the correctness of Theorem 2 is proved.

3.4 RGraph and Adjacency Lists

Although Algorithm 3 does not maintain the exact degree of a vertex unless its degree is less than 3, the low-degree reduction algorithm is not affected at all. Moreover, it is easy to build the RGraph and restore the adjacency lists based on RGraph.

1) Build the RGraph based on Adjacency Lists. A naive method is to apply the **init()** operation provided by Algorithm 3 on each vertex. The time cost is $\Theta(m)$. A more elegant way is that the RGraph itself can be used as adjacency lists. Since each node of RGraph contains two linked lists N_r and N_{bak}, we can just let N_r be empty and use N_{bak} as adjacency lists. In this way, applying **touch()** instead of **init()** on each node is enough to initialize the RGraph. Hence, the time cost reduces to $\Theta(n)$.

2) Restore Adjacency Lists based on the RGraph. When all contractions complete, a simple linear scan can remove all duplicate and deleted vertices from the RGraph. It takes $\Theta(m)$ time to restore the RGraph to normal adjacency lists.

Furthermore, the time complexity can be reduced to the size of the subgraph affected. To initialize the RGraph, there is no need to touch all vertices at the beginning but flexibly touch a vertex when it is visited by the algorithm. As for restoring, it just needs to traverse the neighbor lists of all merged vertices as well. Obviously, the time cost of building and restoring is always dominated by the low-degree reduction algorithm's time complexity.

4. TIE BREAKING STRATEGY

As discussed above, reduction rules can reduce the graph size by including vertices that definitely belong to the MIS. Generally, not all the vertices can be reduced through these reduction rules. When no reduction rules can be applied, it is not easy to determine whether a vertex belongs to any MIS of the remaining graph or not.

4.1 Vertex Addition Framework

One straightforward idea of tie breaking is to choose a candidate vertex u and add it into the independent set. Then vertex u and its neighbors will be deleted from the graph. If there exists an MIS that contains the candidate vertex, this greedy addition is exact and does not degrade the solution quality. Thus it is very critical to determine how likely one vertex could be in an MIS and then conduct selection under this metric greedily. Specifically, the metric can be regarded as a function which maps a vertex to an integer score representing the likelihood. Without loss of generality, assume that a higher score indicates a vertex is more likely to be included in an MIS.

Algorithm 4 shows the pseudo code of the vertex addition framework. Once there is no exact reduction rule to use, the algorithm greedily chooses a vertex that has the largest score as the candidate vertex and conducts reduction mentioned above to break the tie. The key issue is that line 4 must be implemented in a smart way, instead of enumerating all vertices to select the best one. Therefore, we need to use an efficient data structure to maintain the score, which supports updating and finding the largest score efficiently. At the same time, deleting a vertex (line 5) not only removes it from the graph structure, but also updates the scores of all affected vertices and maintains the related indexes.

Actually, many advanced data structures (e.g., balanced binary search tree) can handle updates and maximum (minimum) queries with $O(\log N)$ time complexity where N represents the number of elements. Ignoring the details of implementation, we define the abstract data type BST in Algorithm 5. Except $size()$ which takes $O(1)$ time in line 1, all member functions take $O(\log N)$ time. At the same time, this structure only takes a linear space complexity $O(N)$.

4.2 Vertex Addition Metric

As used in many previous studies, choosing the vertex of the smallest degree is a natural idea. It means that we can simply use the opposite of degree as the score of a vertex.

Definition 7. **Naive Addition Metric**

$$\forall u \in V, u.Score = -d(u)$$

The intuition behind Definition 7 is that the vertex with a lower degree is more likely to be included in an MIS. When a vertex u is deleted from G, we only need to increase the score of each neighbor by one, which takes $O(d(u))$ time and can be covered by $O(m)$ time complexity in total.

However, the demerit is also obvious. At the beginning, there may be a large number of vertices with the lowest degree (e.g., 3). Based on the naive addition metric, what we can do is just selecting one of them randomly, which is basically useless. Therefore, we need to enrich this metric by further selection.

Algorithm 4: Vertex Addition Framework

Input : a graph $G = (V, E)$
Output: a maximal independent set I

1 Initialize the score of each vertex;
2 **while** *G is not empty* **do**
3 **if** *there is no exact reduction to use* **then**
4 Choose a vertex u with the maximum score;
5 Add u into I and then delete $N[u]$ from G;
6 **else** apply exact reduction;
7 **return** I

Algorithm 5: ADT Definition of BST

Element_Type: $(key, value)$

1 int size(): return the number of elements;
2 void insert(k, v): insert an element (k, v);
3 void delete(k): delete the element with key k;
4 void update(k, v'): $(k, v) \leftarrow (k, v')$;
5 Element_Type max(): return an element with the maximum key value;

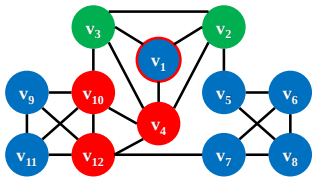

Figure 3: An Example of Combined Addition Metric

Definition 8. **Combined Addition Metric**

$$\forall u \in V,\ u.Score_1 = -d(u),\ u.Score_2 = \min_{v \in N(u)} d(v)$$

In Definition 8, the minimum degree of neighbors is defined as the second key of the score. In other words, if there is more than one vertex that has the maximum $Score_1$ value, we will choose the one whose $Score_2$ value is the highest among them. Based on this definition, all neighbors of the chosen vertex have high degree, which implies that they are unlikely to be in the maximum independent set.

Figure 3 gives an example, where different colors represent vertices of different degree. Blue, green and red denote vertices of degree from 3 to 5 respectively. In this graph, vertex v_1 has the largest $Score_2$ ($v_1.Score_2 = 4$) among all blue vertices. Therefore, we add vertex v_1 into the independent set and delete v_1 and its neighbors from the graph.

4.3 Score Maintenance

In this subsection, we discuss how to efficiently maintain the score of each vertex under the combined addition metric (Definition 8).

Table 2 summarizes the impact on the score by deleting a vertex. When u is deleted, one direct impact is that each neighbor's $Score_1$ increases by 1. Recall that $Score_1$ is defined as $-degree$. Another direct impact is that for each $v \in N(u)$, if u was the minimum degree neighbor of v, v needs to find a new one now and therefore $v.Score_2$ may get larger. Assume that there is a path $u-v-w$, the deletion of u may have an indirect impact on w due to $d(v)$'s decrease. If v becomes the minimum degree neighbor of w, $w.Score_2$ decreases by 1 as well. One more thing, a vertex v can receive both direct and indirect impacts if $v \in N(u) \cap N^2(u)$, where $N^2(u)$ denotes u's 2-hop neighbor set.

Algorithm 6 shows the pseudo code of a brute implementation which simply updates all affected vertices' scores. To be more specific, it first updates $Score_1$ of 1-hop neighbors in lines 1-3, and then updates $Score_2$ of 1-hop and 2-hop neighbors in lines 4-9. As the update operation on BST takes $O(\log n)$ time, the time complexity of deleting vertex u is $O(d(u)^2 \log n)$.

Table 2: Impact on Score when Deleting Vertex u

a path: u - v - w	$Score_1$ (degree)	$Score_2$
Direct Impact on $v \in N(u)$	+1 (-1)	may get larger if $d(u) = v.Score_2$, otherwise stays unchanged
Indirect Impact on $w \in N(v)$	stays unchanged	-1 if the latest $d(v) < w.Score_2$, otherwise stays unchanged

Active Vertex Index. It is easy to see that updating $Score_2$ is the bottleneck of score maintenance. As $Score_2$ is the second key, actually only the vertices with the highest $Score_1$ matter. Therefore, just inserting **vertices with the minimum degree** into BST is a direct idea to reduce the update cost. Specifically, we could regard them as **active vertices** and only maintain them in BST instead of the whole V. Moreover, there is no need to repeatedly update the $Score_2$ value of any inactive vertex. Instead, it merely takes $d(u)$ time to calculate $u.Score_2$ once u becomes active.

As we only delete but not add vertices in this framework, the degree of active vertices has no chance to increase. However, deleting an active vertex may produce one with a much lower degree, which means that the minimum degree goes down. In this circumstance, active vertices could become inactive again, resulting in a waste of time on deleting them from BST. To make sure that each vertex is inserted into BST at most once, in Definition 9 we define the threshold value as the largest minimum degree in history, which is not decreasing. Moreover, in Lemma 3 we devise an upper bound of the threshold value based on the coreness of graph.

Definition 9. **Active Vertex**. A vertex is active if and only if its degree is no greater than a threshold value.

Here $G_i = (V_i, E_i)$ denotes the graph in the i-th while loop of Algorithm 4, so G_0 and G_{now} are the initial and current graphs respectively. Then the threshold value is defined as $\max_{0 \le i \le now}\{\min_{u \in V_i} d(u)\}$.

LEMMA 3. *The threshold value of active vertex is never greater than the biggest coreness value K of the initial graph.*

PROOF. Without loss of generality, we assume that the final threshold value is k, the minimum degree of G_i. Obviously each maximal connected subgraph of G_i is a k-core. As G_i is also a subgraph of G_0, we can say that G_0 contains at least one k-core. Therefore, the biggest coreness value of G_0 is no less than k. □

Algorithm 6: Deleting a Vertex - Brute Method

Input : a graph $G = (V, E)$ and a vertex u
Target: maintain score while deleting vertex u

```
// Array A stores affected vertices.
1  for each vertex v ∈ N(u) do
2  |  d(v) decreases by 1 (v.Score_1 increases by 1);
3  |  add v into A;
4  for each vertex v ∈ N(u) do
5  |  v.Score_2 ← N;
6  |  for each vertex w ∈ N(v) and w ≠ u do
7  |  |  v.Score_2 ← min(v.Score_2, d(w));
8  |  |  w.Score_2 ← min(w.Score_2, d(v));
9  |  |  add w into A;
10 update the score of each v ∈ A on BST;
11 remove u from the graph structure;
```

Lazy Update Mechanism. The lazy update mechanism is commonly used in many traditional scenarios such as deleting an element from a priority queue. Actually a priority queue can only pop the best element but not support deleting an arbitrary element. Under the lazy update mechanism, deleting an element u does not update the priority queue but simply marks u as deleted in constant time. An extra job is that the original pop operation is replaced by a loop, repeatedly popping the best element until it is not deleted. In other words, we put off deleting an element until it is popped. Back to score maintenance, the way we delay updates on BST under the lazy update mechanism is similar, but the circumstance is much more complicated due to various types of updates. When BST reports a vertex u that has the largest score, we need to double check whether the score of u stored in BST is the latest. If not, delete it from BST and insert the latest score into BST again. Otherwise, u has the largest score indeed. However, this inference is based on an assumption that all delayed updates do not make score increase. If the score increases, we must update it on BST at once. Otherwise, this update can be omitted.

Time Complexity Analysis. Algorithm 7 shows the pseudo code of an advanced implementation equipped with both accelerating techniques above. All indirect impact (Table 2) is ignored (lazy update mechanism) and only the active vertices' scores are updated on BST (active vertex index). Noted that line 5 includes a case that v is a newly active vertex due to u's deletion. Whatever, it merely takes $O(K)$ time (Lemma 3) to calculate $v.Score_2$ and $\log n$ time to update it on BST. Therefore, the total time complexity on deleting vertices is $O(m(K + \log n))$.

5. ALGORITHM SCHEDULER

Although there have been many reduction rules and tie breaking strategies, it is required to integrate them to maximize the solution quality and minimize the overall time cost.

5.1 Scheduling Reduction Rules

Instead of invoking reduction algorithms in a fixed order, we introduce the novel framework which dynamically schedules reduction algorithms based on a cost-and-benefit model. Specifically, the scheduler iteratively conducts the

Algorithm 7: Deleting a Vertex - Advanced Method

Input : a graph $G = (V, E)$ and a vertex u
Target: maintain score while deleting vertex u

```
1 for each vertex v ∈ N(u) do
2 |  d(v) decreases by 1 (v.Score_1 increases by 1);
3 for each vertex v ∈ N(u) and v is active do
4 |  recalculate v.Score_2 by traversing N(v);
5 |  update the score of v on BST;
6 remove u from the graph structure;
```

following two operations until the graph is empty. **1)** Estimate the cost and benefit of each reduction rule, and then choose the best one; **2)** If the benefit is overwhelmed by the corresponding cost (the reduction cannot even reduce any vertex in the worst case), greedy algorithms are invoked to break the tie.

Algorithm 8 shows the pseudo code. In each iteration of the while loop (lines 1-8), it first estimates the benefit density of each reduction rule (line 2). If the best one (denoted as ra) still has a poor performance, the scheduler chooses to apply greedy algorithms (line 4). Otherwise, it applies ra on the graph and updates G and I accordingly (line 6). In order to avoid an infinite loop, the scheduler applies greedy algorithms as well when the actual benefit of ra is poor (line 8). The procedure of applying greedy algorithms is described in lines 10-12.

Formally, we define the **benefit density** to evaluate the performance of each reduction rule, and set a threshold on the **benefit density** to determine poor performance. In our experiments, the reduction rule has a poor performance if it reduces less than 100 vertices per millisecond.

Definition 10. **Benefit Density**

$$BD = \frac{reducing\ ability}{processing\ time}$$

where **reducing ability** represents the decrease of graph size after applying the reduction rule. Specifically, it can be defined as the number of reduced vertices, edges or a trade-off. **Processing time** refers to the actual time consumed by running the corresponding reduction algorithm. Thus, it is critical to conduct fast and effective estimation of the reducing ability and processing time of each reduction rule.

Estimation of Processing Time. We assume that the processing time of an algorithm is linear to the graph size, since the whole algorithm is supposed to run on extremely large graphs and cannot contain any high time complexity (e.g., $O(n^2)$) reduction algorithm. Similar to the definition of reducing ability, the size of graph could be either $|V|$ or $|E|$ which depends on the characteristic of the specific reduction algorithm. For a reduction algorithm ra, let t_i and $size_i$ denote the processing time and graph size of the i-th running of ra. Then, we have the following equation:

$$\forall i \geq 1, speed_{ra} = \frac{size_i}{t_i}$$

Initially, we assign the same large value to each $speed_{ra}$. Then, we can use the above equation to estimate t_i. After the i-th running of ra, we can use the actual t_i to update $speed_{ra}$. Under this mechanism, the scheduler tends to apply all reduction algorithms in turn at the beginning, and then each $speed_{ra}$ will fall back to a normal range. Moreover, appropriately reducing the initial speed of a reduction algorithm can make it applied much later. It is a useful method to avoid applying a time-consuming reduction algorithm to the initial large graph.

Estimation of Reducing Ability. In practice, we design specific estimation methods for each reduction algorithm. The number of degree-one and degree-two vertices is easy to maintain and it gives a rough estimate on degree-one and degree-two reduction respectively. As for dominance reduction, random sampling is an efficient and robust method. Each time we randomly select a constant number (denoted as c) of vertices and check whether each of them is dominated by a neighbor. Assume that c_{dom} vertices of them are dominated and the current graph contains n_{now} vertices, then the total number of vertices that are dominated in the current graph is estimated to be $n_{now} \times c_{dom}/c$. The average time complexity of estimating degree-one, degree-two and dominance reduction is $O(1)$, $O(1)$ and $O(c\overline{d}^2)$ respectively, where $\overline{d}$ denotes the average degree of vertices.

Algorithm 8: Scheduling Framework

Input : a graph $G = (V, E)$, reduction algorithm set RA and greedy algorithm set GA
Output: an independent set I

```
1  while G is not empty do
2  |  Estimate each reduction algorithm's benefit
   |   density, then choose the best one ra ∈ RA;
3  |  if ra's estimated performance is poor then
4  |  |  Apply_Greedy_Algorithm()
5  |  else
6  |  |  (G, I) ← ra(G, I);
7  |  |  if ra's actual performance is poor then
8  |  |  |  Apply_Greedy_Algorithm()
9  return I
10 Procedure Apply_Greedy_Algorithm()
11 |  Choose a greedy algorithm ga ∈ GA which fits
   |   the current graph best;
12 |  Repeatedly apply ga until the size of G reduces
   |   significantly;
```

5.2 Scheduling Greedy Algorithms

Trigger Mechanism. Generally speaking, we change the traditional trigger mechanism from "reduction algorithms do not work" [6] to "they do not work well". The traditional method is not good enough due to the diminishing marginal effect, that is the number of vertices reduced in each iteration gradually decreases to zero (i.e., reduction rules do not work) and then greedy algorithms are triggered. Therefore, the reduction algorithms usually obtain quite limited benefits in the last several iterations before triggering greedy algorithms. Naturally, we want to trigger greedy algorithms much earlier to avoid these inefficient iterations. As shown in Algorithm 8, our novel mechanism invokes greedy algorithms when the best reduction algorithm has a poor performance, i.e., its benefit density is less than the given threshold. Under this mechanism, the whole algorithm can keep running efficiently.

Hybrid Tie Breaking Strategy. Currently, there are mainly two kinds of tie breaking strategies: vertex deletion (delete the vertex of the largest degree [6]) and vertex addition (add the vertex of the smallest degree [15]). As shown in the experiments, we find that vertex deletion performs better than vertex addition on most real networks. On the contrary, they just show opposite performance on most ER graphs. Generally, real networks have a power-law degree distribution, which means that there are few high-degree vertices but a large number of low-degree vertices. As for ER graphs, the degree of vertices concentrates around the average degree $\overline{d}$, i.e., both the number of low-degree and high-degree vertices are quite few. In essence, a strategy

has a good performance when it gives a good partition, i.e., the gap between the chosen vertices and the others is large.

Based on the above analysis, we propose a hybrid strategy as shown in Definition 11, which dynamically uses the deletion and addition strategies depending on d_{max} (the maximum degree), d_{min} (the minimum degree) and $\overline{d}$ (the average degree) of the current graph.

Definition 11. **Hybrid Tie Breaking Strategy**. Apply the addition strategy when $\overline{d} - d_{min} \geq d_{max} - \overline{d}$, otherwise apply the deletion strategy.

It is easy to further improve this strategy with the help of detailed degree distribution information. However, we keep this simple version since it clearly shows the idea of hybrid strategy and also avoids overfitting on experimental graphs.

Batch Deletion Technique. Traditionally, once a vertex is greedily deleted from the current graph or added into the solution, the reduction algorithms will be invoked until their benefit density becomes low again. However, the benefit density stays almost unchanged due to several vertices' deletion. The scheduler will apply greedy algorithms once and once again, wasting a large amount of time in meaningless estimation. Hence, we propose the batch deletion technique: each time the greedy algorithm is invoked until a percentage of edges are deleted. In our experiment, we set the percentage to be 2%. Note that batch deletion can improve the time efficiency without degrading the solution quality obviously. The main reason is that if we do not force to invoke greedy algorithms repeatedly, the scheduler will also resort to greedy algorithms since reduction algorithms' estimated benefit density is still low.

Delay Indexing. Note that the greedy algorithms are not always used in the scheduling framework. In the extreme case, a graph can be reduced to empty merely by applying reduction algorithms, so that an exact solution is found. However, when the index for greedy strategy is constructed, there is always some update cost while deleting vertices. Hence, a simple but effective idea is: the construction of index can be delayed until the scheduler applies a greedy algorithm. In this way, there is no cost on maintaining the index when greedy algorithms are never used.

5.3 Recovery Technique

As shown in block b of Figure 2, the scheduler finally outputs a Near-MIS (denoted as I_b). In this subsection, we discuss how to further improve the solution quality.

A naive method proposed in [6] is simply extending the solution independent set to be maximal. In the following we propose an advanced recovery technique which resorts to updating algorithms. Suppose that we remove all vertices deleted by greedy algorithms from the original graph $G = (V, E)$ and then we have $G' = (V', E')$. It is easy to prove that $I_b \cap V'$ is an MIS on G'. Now, for each vertex $u \in V - V'$, we add it back into G' and maintain I_b to be an MIS at the same time. The above method may be a potential exact algorithm of the MIS problem. However, we merely aim to find a Near-MIS within reasonable time in this paper. Therefore, we adopt inexact updating algorithms as well, which maintain a Near-MIS during graph updating operations, such as $LSTwo+LazySearch^+$ in [26] and *DGOracle* in [25]. As shown at the bottom of Figure 2, the updating algorithm receives a, b and c as input, and finally returns a Near-MIS I_d which is larger than I_b.

Table 3: Statistics of Real Networks

Graph	$\|V\|$	$\|E\|$	$\overline{d}$	d_{max}
Wiki-Vote	7,115	100,762	28	1,065
CA-AstroPh	18,771	198,050	21	504
Epinions	75,879	405,740	11	3,044
com-amazon	334,863	925,872	6	549
com-dblp	317,080	1,049,866	7	343
web-Google	875,713	4,322,051	10	6,332
Wiki-Talk	2,394,385	4,659,565	4	100,029
BerkStan	685,230	6,649,470	19	84,230
as-skitter	1,696,415	11,095,298	13	35,455
cit-Patents	3,774,768	16,518,947	9	793
soc-pokec	1,632,803	22,301,964	27	14,854
LiveJournal	4,846,609	42,851,237	18	20,333
com-orkut	3,072,441	117,185,083	76	33,313

Table 4: Statistics of ER and ER* Graphs

Graph	$\|V\|$	$\|E\|$	$\overline{d}$	d_{max}	d^*_{max}
ER-1	100,000	751,263	15	34	37
ER-2	500,000	4,939,792	20	43	42
ER-3	1,000,000	24,585,034	49	88	89
ER-4	3,000,000	114,257,845	76	126	122

6. EXPERIMENTS

6.1 Data Sets

The proposed algorithms are evaluated on 13 real networks and three kinds of synthetic graphs. The statistics of these graphs are listed in Tables 3 and 4, where $|V|$, $|E|$, $\overline{d}$ and d_{max} denote the number of vertices and edges, the average and maximum degree respectively. All of the **real networks** in Table 3 are downloaded from the Stanford Network Analysis Platform [13] and frequently used in previous studies [1, 6, 25].

As shown in Table 4, four synthetic graphs **ER-x** generated by the Erdos-Renyi model are used to evaluate the practical performance of the proposed algorithms in general cases. The Erdos-Renyi (ER in short) model [20] is a classical random graph model for analyzing the average case of graph algorithms [17], where each edge exists with probability p independently. It can also be used to model different graph structures such as communities [21]. In this paper, the ER model receives two parameters, the number of vertices n and expected average degree d. Then, each pair of vertices generates an edge with the same probability $\frac{d}{n-1}$. Obviously, vertices tend to have a similar degree around the average degree in an ER graph.

Since all ER graphs have similar topology, we generate four graphs **ER*-x** by randomly updating 10% edges on the above ER graphs. Specifically, each update operation consists of four steps: 1) Randomly pick an existing edge; 2) Delete this edge; 3) Randomly pick a pair of vertices without an edge; 4) Add an edge between them. In this way, the number of vertices and edges remains unchanged. Hence, a column d^*_{max} is appended to Table 4, denoting the maximum degree of the ER* graphs. Beyond that, six **Dense Graphs** are used to test the robustness. Each of them has 10^4 vertices but varies in the percentage of $|E|$ from 1% to 75%, compared with the complete graph.

6.2 Competitors

For ease of presentation, let **Sch-standard** and **Sch-extension** denote our proposed scheduler-based methods. We compare the following methods.

Real Networks		Wiki-Vote	Epinions	CA-AstroPh	com-amazon	Wiki-Talk	web-Google	com-dblp	LiveJournal	BerkStan	as-skitter	soc-pokec	cit-Patents	com-orkut
GroundTruth(*)		4866	53599	6759	174632	2338222	529138	152131	2630941	408482	1170580	789207*	2089887*	833922*
Gap to GroundTruth(*)	LinearTime	0	0	1	21	0	150	2	379	762	170	200	2522	6007
	NearLinear	0	0	0	3	0	4	0	36	426	39	57	2124	5256
	Sch-standard	0	0	0	1	0	4	0	29	245	31	28	171	1362
	Sch-extension	0	0	0	0	0	4	0	16	221	27	0	0	0

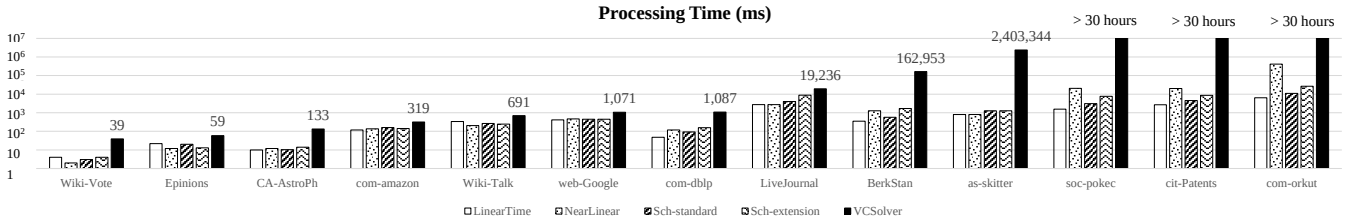

Figure 4: Overall Performance on Real Networks

Sch-standard: The standard version of our scheduling framework. It applies the core technique proposed in this paper, i.e., dynamically scheduling degree-one, degree-two and dominance reduction algorithms based on the cost-and-benefit estimation. It uses the basic tie breaking strategy (i.e., deleting a max-degree vertex) in the batch manner and the naive recovery method.

Sch-extension: The extension version of our scheduling framework equipped with the hybrid tie breaking strategy, delay indexing and advanced recovery technique. Specifically, we choose DGOracle [25] as the updating algorithm for recovery. To improve the efficiency of recovery, we simplify DGOracle by disabling the DFS procedure which aims to find a valid swap.

LinearTime & **NearLinear** [6]: The state-of-the-art methods for finding a Near-MIS in linear and near-linear time respectively.

VCSolver [1]: The state-of-the-art implementation of the exact MIS algorithm. If it can find an MIS within 1 hour, we use the size of this MIS as ground truth.

All experiments are conducted on a 2012 Windows Server. Our algorithms are implemented in C++ and compiled with -O3 option. The existing algorithms' codes are published by their authors and we obtain them from public resources.

6.3 Experimental Results

6.3.1 Comparison with State-of-the-art Algorithms

Figures 4 and 5 present the overall performance on real networks and three kinds of synthetic graphs respectively. Each figure consists of two parts: solution quality and processing time.

Evaluation of Solution Quality. **GroundTruth** in Figure 4 denotes the size of an MIS. The gap between the ground truth and the size of Near-MIS delivered by each algorithm is computed, the smaller the better. $Gap = 0$ means that the algorithm actually finds an MIS, i.e., the optimal solution. **Sch-standard** and **Sch-extension** outperform the competitors. As equipped with the hybrid tie breaking strategy and advanced recovery technique, **Sch-extension** finds larger solutions than **Sch-standard** on most graphs.

To further differentiate the solution quality of these algorithms, they are evaluated on graphs which are too large to compute the ground truth. In the last three columns of Figure 4 and the whole Figure 5, we simply adopt the best result of four algorithms as **GroundTruth*** which is used to compute the gap accordingly. It is clear that the gap gradually decreases to zero from top to bottom on almost all graphs, i.e., **Sch-extension** > **Sch-standard** > **NearLinear** > **LinearTime** in terms of solution quality. Note that $Gap = 0$ indicates that **Sch-extension** outperforms the others but not necessary finds an MIS. **Sch-extension** improves the quality of **LinearTime** and **NearLinear** by $3.7\% - 7.0\%$ on ER and ER* graphs.

Evaluation of Time Efficiency. In the lower part of Figures 4 and 5, the y axis denotes the processing time in millisecond. It is clear that **LinearTime** > **Sch-standard** > **Sch-extension** > **NearLinear** $\gg$ **VCSolver** in terms of time efficiency on most graphs. Concretely, **Sch-standard**, **Sch-extension** and **NearLinear** are at most 2, 5 and 65 times slower than **LinearTime** on real networks respectively, and at most 2, 20, 50 times slower on ER and ER* graphs respectively. **LinearTime** is the most efficient algorithm as it adopts very simple techniques (a subset of others') and consumes $\Theta(n + m)$ time on any graph. Our proposed algorithms are slower than **LinearTime** on large graphs due to the near-linear time complexity, but run much faster than **NearLinear**. **NearLinear** performs especially poor on dense graphs, i.e., thousands of times slower than **LinearTime**. The performance of **Sch-standard** and **Sch-extension** is stable, which confirms the robustness of our scheduler framework. Furthermore, the exact algorithm **VCSolver** is far more time-consuming than the others, and is manually stopped if it does not terminate within 30 and 3 hours on real networks and synthetic graphs respectively.

6.3.2 Evaluation of Scheduling Framework

In order to study the performance of the scheduling framework, we conduct extensive ablation test. The results are presented in Figure 6, where the y axis denotes the time cost and the x axis denotes the gap to the ground truth. Note that **com-orkut** and **ER-4** are hard cases on which the maximum independent sets are not known. We have to use the largest independent sets computed by **Sch-extension** as the ground truth.

All compared algorithms are listed on the left side. Each of **Fixed Order**, **Single Deletion** and **Probability Estimation** replaces dynamic order, batch deletion, and random sample from **Sch-standard**, respectively. Therefore, in the following we focus on the comparison between **Sch-standard** and them to evaluate each specific technique. We also include **Sch-extension**'s results for reference.

Synthetic Graphs		ER Graphs				Randomly Update 10% Edges on ER Graphs				Dense Graphs					
		ER-1	ER-2	ER-3	ER-4	ER*-1	ER*-2	ER*-3	ER*-4	Dense-1%	Dense-5%	Dense-10%	Dense-25%	Dense-50%	Dense-75%
GroundTruth*		24445*	102042*	107382*	230267*	24853*	104145*	114837*	268362*	610*	159*	83*	36*	16*	9*
Gap to GroundTruth*	LinearTime	912	4240	6507	16163	875	3912	5533	13785	41	22	9	7	2	2
	NearLinear	901	4247	6472	16146	872	3931	5520	13795	40	26	9	7	2	2
	Sch-standard	467	2168	4365	11997	383	1811	3331	9663	38	15	11	5	3	2
	Sch-extension	0	0	0	0	0	0	0	0	0	0	0	0	0	0

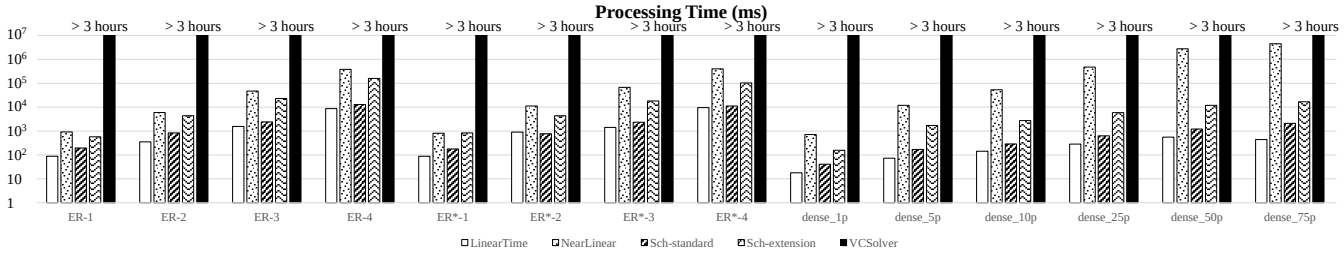

Figure 5: Overall Performance on Synthetic Graphs

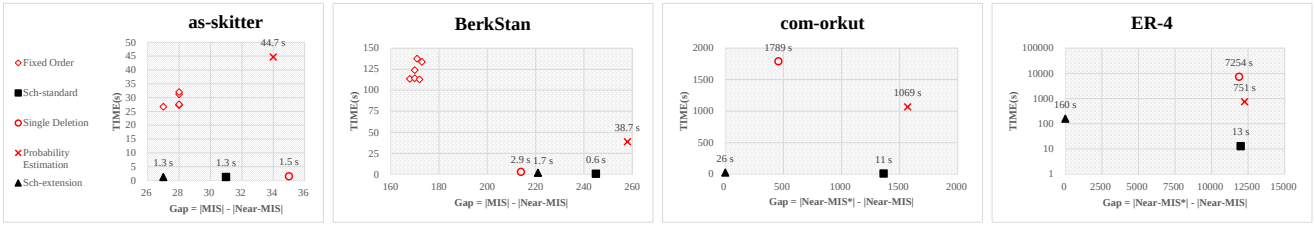

Figure 6: Ablation Test: Evaluate Specific Techniques of the Scheduling Framework

Fixed Order vs. Dynamic Order. Fixed Order represents the traditional method that applies degree-one reduction, degree-two reduction and dominance reduction in a fixed order instead of making a decision by the scheduler. Under **Fixed Order**, there are six different orders to schedule the above three reductions, which are depicted by the six points in Figure 6. However, we do not draw any point of **Fixed Order** in the last two figures because it does not find a result within an hour on **com-orkut** or **ER-4**.

The results show that different orders do lead to different solutions and influence the processing time. However, the difference is not obvious among the fixed orders. By comparison, **Sch-standard** schedules reduction algorithms in a dynamic order and enhances the usage of greedy algorithms, which results in a minor loss of solution quality but saves a substantial amount of time. In other words, there is a better trade-off between solution quality and time efficiency in our scheduling framework.

Single Deletion vs. Batch Deletion. Single Deletion deletes one vertex when applying the greedy algorithm. On the contrary, **Sch-standard** adopts the batch deletion technique. It is clear that **Single Deletion** and **Sch-standard** have almost the same solution quality. As for time efficiency, their performance is still close on small real networks. However, when it comes to more complicated graphs, **Sch-standard** is two orders of magnitude faster than **Single Deletion**. Therefore, we can conclude that the batch deletion technique does not sacrifice solution quality but improves the time efficiency especially in complicated graphs.

Probability Estimation vs. Random Sample. In **Sch-standard**, we use random sampling to estimate the effect of dominance reduction. By comparison, **Probability Estimation** adopts a statistical method which roughly estimates an expected number of reduced vertices in linear time. Figure 6 clearly shows that **Sch-standard** is superior to **Probability Estimation** in terms of both solution quality and time efficiency, which confirms the effectiveness of random sampling.

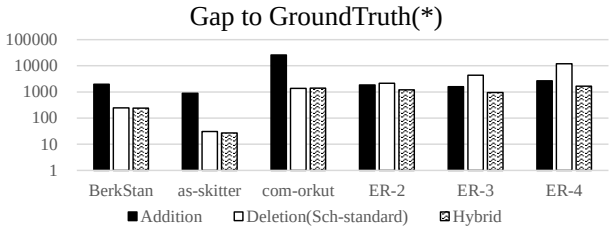

Figure 7: Solution Quality of Three Strategies

6.3.3 *Evaluation of Tie Breaking Strategy*

In this experiment, we still use **Sch-standard** as the control algorithm and vary in tie breaking strategies. Figure 7 depicts the performance of solution quality. We can see that the comparison between **Addition** and **Deletion** is contingent upon the type of graph. The deletion strategy performs excellently on real networks due to the power-law degree distribution. In contrast, the addition strategy performs better on ER graphs because the degree of vertices concentrates

Table 5: Processing Time of Three Strategies

Time(s)	Addition		Deletion	Hybrid
	Brute	Advance		
BerkStan	59.8	1.2	0.6	0.6
as-skitter	59.7	1.6	1.3	1.3
com-orkut	1415.3	14.3	10.9	15.6
ER-2	28.4	3.1	0.8	3.2
ER-3	196.4	22.7	2.4	20.0
ER-4	2240.4	179.1	12.9	147.9

Table 6: Evaluate the Impact of Near-MIS on Kreach Construction

Graph	Near-MIS Computation			Kreach Construction			
	Algorithm	Size	Time(s)	Index Size(GB)	Time(s)		
					$k=2$	$k=3$	$k=4$
BerkStan	LinearTime	407720	0.350	11.928	2003	2020	2816
	Sch-standard	408237	0.571	11.906	1933	1983	2429
	Sch-extension	408261	1.685	11.905	1747	1834	2157
cit-Patents*	LinearTime	447010	0.816	29.330	3876	3734	4439
	Sch-standard	448208	1.273	29.260	3658	3595	4098
	Sch-extension	448313	2.884	29.250	3490	3394	4026

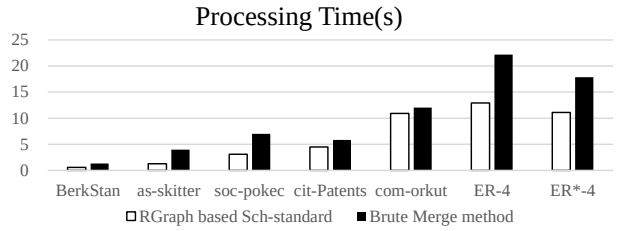

Figure 8: Sch-standard with vs. without RGraph

around $\overline{d}$. **Hybrid** takes both features into consideration and delivers larger independent sets than any single strategy, which is especially obvious on ER graphs. Note that on real networks the improvement is tiny, as the deletion strategy fits this type of graph well and then **Hybrid** almost degenerates into **Deletion**.

Table 5 reports the time efficiency. Compared to **Sch-standard**, the brute implementation of **Addition** is too time-consuming. Nevertheless, the time consumed by the advanced implementation decreases by one order of magnitude. It is normal that **Advance** is still slower than **Sch-standard**, as the deletion strategy is too simple. Moreover, combining these two strategies, **Hybrid** is also efficient enough to be applied to large graphs.

6.3.4 Evaluation of Representative Graph

To evaluate the practical effect of **RGraph**, we compare the performance of **Sch-standard** by enabling and disabling RGraph on large real networks, ER-4 and ER*-4 graphs. Figure 8 reports the efficiency results. It is clear that exploiting RGraph can reduce the time cost on all graphs, e.g., it achieves 2.3x and 1.8x speedup on soc-pokec and ER-4, respectively.

6.3.5 Evaluation of the Impact of Near-MIS on a Downstream Task

In this experiment, we evaluate how the quality of a Near-MIS affects a downstream task, k-hop reachability query [7]. Cheng et al. [7] design an index **Kreach** to efficiently process k-hop reachability queries which ask whether there is a path from a node to another within k hops. The construction of **Kreach** relies on computing the pair-wise distance among nodes in a vertex cover (the complement of an independent set) of a graph, thus the first step of **Kreach** is to compute a vertex cover (or a Near-MIS equivalently). A smaller vertex cover (i.e., a larger Near-MIS) leads to more efficient construction of **Kreach**.

Specifically, the **Kreach** index is constructed based on the Near-MIS generated by different algorithms on two large real networks **Berkstan** and **cit-Patents**, with the hop parameter k varying from 2 to 4. Note that the **Kreach** index on **cit-Patents** is too large and exceeds the memory limit of our server, so we select a connected component whose node size is 1/4 of that of the original graph, denoted as **cit-Patents***, to conduct this experiment.

The experimental results are presented in Table 6. Although the Near-MIS of **Sch-extension** is larger than that of **LinearTime** by only 0.13% and 0.29% on **BerkStan** and **cit-Patents*** respectively, it can significantly accelerate the construction of **Kreach** by up to 23.4% and 10.0% on the two networks respectively. Moreover, the time consumed by Near-MIS computation is marginal compared to that of building the **Kreach** index. For example, constructing **Kreach** based on the Near-MIS of **Sch-extension** on **Berkstan** saves 659 seconds when $k=4$ compared to the Near-MIS of **LinearTime**, but it only costs 1.335 seconds more than **LinearTime** to compute the Near-MIS. These results on the k-hop reachability task demonstrate that it is valuable to further improve the Near-MIS quality by our proposed methods at the expense of mild extra time cost.

7. CONCLUSIONS

In this paper, we study the inexact algorithms of MIS computation, and focus on how to organise and improve existing techniques to obtain a more powerful algorithm. In general, we propose a scheduling framework that dynamically invokes various reduction and greedy algorithms based on the benefit-and-cost estimation. Moreover, a novel data structure called RGraph is devised to reduce the worst-case time complexity of degree-two reduction from $O(nm)$ to $O(m \log n)$. As for greedy algorithms, we design a more powerful vertex addition strategy which takes the degree information of neighbors into consideration, and two techniques active vertex index and lazy update mechanism are developed to improve the time efficiency. Extensive experiments on both large real networks and various types of synthetic graphs confirm that our proposed algorithms **Sch-standard** and **Sch-extension** outperform the state-of-the-art near-linear time algorithm **NearLinear** [6] in terms of both solution quality and time efficiency on most graphs.

8. ACKNOWLEDGMENTS

The work was supported by grants from the Research Grant Council of the Hong Kong Special Administrative Region, China [Project No.: CUHK 14205617] and [Project No.: CUHK 14205618], Tencent AI Lab RhinoBird Focused Research Program GF202005, NSFC Grant No. U1936205. Weiguo Zheng was supported by National Natural Science Foundation of China (Grant No. 61902074) and Science and Technology Committee Shanghai Municipality (Grant No. 19ZR1404900). Weiguo Zheng and Hong Cheng are the corresponding authors.

9. REFERENCES

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

[2] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings of the 2013 ACM International Conference on Management of Data*, pages 349–360, 2013.

[3] F. Araujo, J. Farinha, P. Domingues, G. C. Silaghi, and D. Kondo. A maximum independent set approach for collusion detection in voting pools. *Journal of Parallel and Distributed Computing*, 71(10):1356–1366, 2011.

[4] S. Basagni. Finding a maximal weighted independent set in wireless networks. *Telecommunication Systems*, 18(1–3):155–168, 2001.

[5] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 ACM International Conference on Management of Data*, pages 1199–1214, 2016.

[6] L. Chang, W. Li, and W. Zhang. Computing a near-maximum independent set in linear time by reducing-peeling. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1181–1196, 2017.

[7] J. Cheng, Z. Shang, H. Cheng, H. Wang, and J. X. Yu. K-reach: Who is in your small world. *PVLDB*, 5(11):1292–1303, 2012.

[8] J. Dahlum, S. Lamm, P. Sanders, C. Schulz, D. Strash, and R. F. Werneck. Accelerating local search for the maximum independent set problem. In *Proceedings of the 15th International Symposium on Experimental Algorithms*, pages 118–133, 2016.

[9] A. K. Datta, L. L. Larmore, S. Devismes, K. Heurtefeux, and Y. Rivierre. Competitive self-stabilizing k-clustering. In *Proceedings of the 2012 IEEE 32nd International Conference on Distributed Computing Systems*, pages 476–485, 2012.

[10] F. V. Fomin, F. Grandoni, and D. Kratsch. A measure & conquer approach for the analysis of exact algorithms. *Journal of the ACM*, 56(5):25:1–25:32, 2009.

[11] M. R. Gary and D. S. Johnson. Computers and intractability: A guide to the theory of np-completeness, 1979.

[12] J. Hastad. Clique is hard to approximate within $n^{1-\epsilon}$. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, pages 627–636, 1996.

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

[14] Y. Li, L. Zhang, and Z. Liu. Multi-objective de novo drug design with conditional graph generative model. *Journal of Cheminformatics*, 10(1):33, 2018.

[15] Y. Liu, J. Lu, H. Yang, X. Xiao, and Z. Wei. Towards maximum independent sets on massive graphs. *PVLDB*, 8(13):2122–2133, 2015.

[16] C. Lu, J. X. Yu, H. Wei, and Y. Zhang. Finding the maximum clique in massive graphs. *PVLDB*, 10(11):1538–1549, 2017.

[17] T. McKenzie, H. Mehta, and L. Trevisan. A new algorithm for the robust semi-random independent set problem. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 738–746, 2020.

[18] Z. P. Ogihara, M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, pages 283–286, 1997.

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

[20] A. Rényi. On random graphs I. *Publicationes Mathematicae*, 4:3286–3291, 1959.

[21] C. Seshadhri, T. G. Kolda, and A. Pinar. Community structure and scale-free collections of erdős-rényi graphs. *Physical Review E*, 85(5):056109, 2012.

[22] J. Su, Q. Zhu, H. Wei, and J. X. Yu. Reachability querying: Can it be even faster? *IEEE Transactions on Knowledge and Data Engineering*, 29(3):683–697, 2016.

[23] M. Xiao and H. Nagamochi. Exact algorithms for maximum independent set. *Information and Computation*, 255:126–146, 2017.

[24] K. Xirogiannopoulos and A. Deshpande. Extracting and analyzing hidden graphs from relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 897–912, 2017.

[25] W. Zheng, C. Piao, H. Cheng, and J. X. Yu. Computing a near-maximum independent set in dynamic graphs. In *Proceedings of the 2019 IEEE 35th International Conference on Data Engineering*, pages 76–87, 2019.

[26] W. Zheng, Q. Wang, J. Xu Yu, H. Cheng, and L. Zou. Efficient computation of a near-maximum independent set over evolving graphs. In *Proceedings of the 2018 IEEE 34th International Conference on Data Engineering*, pages 869–880, 2018.