Computing A Near-Maximum Independent Set in Linear Time by Reducing-Peeling

Lijun Chang, Wei Li, Wenjie Zhang
University of New South Wales, Australia
{ljchang,weili,zhangw}@cse.unsw.edu.au

ABSTRACT

This paper studies the problem of efficiently computing a maximum independent set from a large graph, a fundamental problem in graph analysis. Due to the hardness results of computing an exact maximum independent set or an approximate maximum independent set with accuracy guarantee, the existing algorithms resort to heuristic techniques for approximately computing a maximum independent set with good performance in practice but no accuracy guarantee theoretically. Observing that the existing techniques have various limits, in this paper, we aim to develop efficient algorithms (with linear or near-linear time complexity) that can generate a high-quality (large-size) independent set from a graph in practice. In particular, firstly we develop a Reducing-Peeling framework which iteratively reduces the graph size by applying reduction rules on vertices with very low degrees (Reducing) and temporarily removing the vertex with the highest degree (Peeling) if the reduction rules cannot be applied. Secondly, based on our framework we design two baseline algorithms, BDOne and BDTwo, by utilizing the existing reduction rules for handling degree-one and degree-two vertices, respectively. Both algorithms can generate higher-quality (larger-size) independent sets than the existing algorithms. Thirdly, we propose a linear-time algorithm, LinearTime, and a near-linear time algorithm, NearLinear, by designing new reduction rules and developing techniques for efficiently and incrementally applying reduction rules. In practice, LinearTime takes similar time and space to BDOne but computes a higher quality independent set, similar in size to that of an independent set generated by BDTwo. Moreover, in practice NearLinear has a good chance to generate a maximum independent set and it often generates near-maximum independent sets. Fourthly, we extend our techniques to accelerate the existing iterated local search algorithms. Extensive empirical studies show that all our algorithms output much larger independent sets than the existing linear-time algorithms while having a similar running time, as well as achieve significant speedup against the existing iterated local search algorithms.

Keywords

Maximum independent set, minimum vertex cover, power-law graphs, reducing-peeling, reduction rules, linear time

SIGMOD '17, May 14–19, 2017, Chicago, Illinois, USA.
© 2017 ACM. ISBN 978-1-4503-4197-4/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3035918.3035939

1. INTRODUCTION

Graph model has been widely used to represent the relationships among entities in a wide spectrum of applications such as social networks, collaboration networks, communication networks and biological networks. Significant research efforts have been devoted towards many fundamental problems in managing and analysing graph data. In this paper, we study the problem of efficiently computing an approximate maximum independent set of a graph. A subset I of vertices in a graph G is an *independent set* if there is no edge between any two vertices in I, and its *size* is measured by the number of vertices in it. The independent set with the largest size among all independent sets of G is called the *maximum independent set* of G, which is not unique. Consider the graph in Figure 1, $\{v_2, v_5, v_7, v_9\}$ is an independent set of size 4, while $\{v_1, v_4, v_6, v_8, v_{10}\}$ is a maximum independent set of size 5.

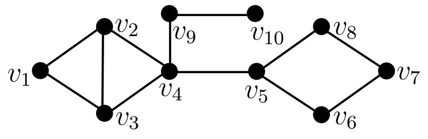


Figure 1: Example

The (maximum) independent set is closely related to the (minimum) vertex cover, where a subset C of vertices in G is a *vertex cover* if it contains at least one end-point for every edge in G. It is well-known [34] that $I \subseteq V$ is a (maximum) independent set of G if and only if $V \backslash I$ is a (minimum) vertex cover of G. For example, in Figure 1, $\{v_1, v_4, v_6, v_8, v_{10}\}$ and its complement $\{v_2, v_3, v_5, v_7, v_9\}$ are the maximum independent set and the minimum vertex cover, respectively. Thus, computing (maximum) independent sets is equivalent to computing (minimum) vertex covers.

Applications. Computing a maximum independent set of a graph is a prominent fundamental optimization problem and has many important applications. For example, it has been used in computing social network coverage and reach [32]; that is, compute a set of vertices covering the graph within their one or several hops neighborhoods. Araujo et al. [3] propose a maximum independent set-based approach for collusion detection in voting pools. Wan et al. [36] reduce the multiflow problem to a sequence of maximum independent set computations to make a trade-off between accuracy and efficiency. Basagni [5] studies the maximum weighted independent set problem in a wireless network to organize the nodes of the network in a hierarchical way. Moreover, computing maximum independent sets (or equivalently, computing minimum vertex covers) is also invoked as a subroutine in effective strategies for other graph processing problems, such as in constructing an effective matching order for subgraph enumeration [8], in build-

Algorithm	Time Complexity	Space Complexity	Exact Reduction Rules Used
BDOne	$O(m)$	$2m + O(n)$	Degree-one reduction [21]
BDTwo	$O(n \times m)$	$6m + O(n)$	Degree-one reduction [21] & Degree-two vertex reductions [21]
LinearTime	$O(m)$	$2m + O(n)$	Degree-one reduction [21] & Degree-two path reduction (this paper)
NearLinear	$O(m \times \Delta)$	$4m + O(n)$	Dominance reduction [21] & Degree-two path reduction (this paper)

Table 1: Overview of our approaches (n: number of vertices, m: number of edges, Δ: maximum vertex degree)

ing indexes for processing shortest path/distance queries [14, 22], in contracting the input graph for external-memory SCC computation [39], and in refining the result of matching two graphs [40].

Existing Algorithms. The existing algorithms for computing maximum independent sets can be categorized as follows.

Exact Algorithms. It is NP-hard to compute a maximum independent set [23]. The state-of-the-art exact algorithms [1, 21] are based on the branch-and-reduce paradigm and have worst-case exponential time complexities regarding n, the number of vertices in G. When considering a vertex u, the algorithm branches on two cases: (i) u is in the independent set, and (ii) u is not. Based on branching rules, reduction rules, and the measure and conquer analysis, the algorithm in [21] runs in $O^*(1.2201^n)$ time, where the notation O^* hides factors polynomial in n. Akiba and Iwata [1] recently show that this paradigm can compute a minimum vertex cover (and a maximum independent set) for many small and medium-sized graphs by developing more branching and reduction rules.

Approximation Algorithms. It is pointed out in [26] that approximately computing a maximum independent set within a factor of $n^{1-\epsilon}$ for any $0 < \epsilon < 1$ (*i.e.*, tighter than n) is NP-hard; that is, maximum independent set is also hard to approximate. As a result, the approximation ratios of the existing techniques depend on either n or Δ, where Δ is the maximum vertex degree of G and is often of the same order as n; for example, the approximation ratios are $O(n(\log\log n)^2/(\log n)^3)$ in [20], $\frac{\Delta+2}{3}$ in [25], and $\frac{\Delta+3}{5}$ in [7].

Heuristic Algorithms. In practice, linear-time algorithms such as Greedy and DU are often adopted to approximately compute a maximum independent set [30]. Both algorithms iteratively add the minimum-degree vertex into an initially empty solution, and at the same time remove the vertex and all its neighbors from the graph. The difference between these two algorithms is that Greedy considers vertex degrees in a static way, while DU considers vertex degrees adaptively in the remaining graph.

Recently, iterative techniques have been proposed to progressively generate larger independent sets until reaching some predefined thresholds; for example, local search algorithms (ARW [2, 30]), evolutionary algorithms (ReduMIS [28]), and algorithms combining local search with simple reduction rules (OnlineMIS [19]).

Motivation. To summarize, due to the hardness results of computing an exact maximum independent set or approximately computing a maximum independent set with accuracy guarantee, the existing algorithms resort to heuristic techniques for approximately computing a maximum independent set without accuracy guarantee. While Greedy and DU have linear time complexities, they are only able to generate an approximate maximum independent set with limited accuracy in practice. On the other hand, although the local search and evolutionary algorithms ARW, ReduMIS, and OnlineMIS can find highly accurate approximate maximum independent sets by iteratively improving the solution regarding its size, they could take excessive long time. These motivate our study in this paper. We aim to design new paradigms and algorithms with linear or near-linear time complexities to compute high-quality (large-size) independent sets for large graphs. Below are our main observations. *(i) Real networks are usually power-law graphs* with many low-degree vertices [4]. *(ii) Reduction rules* (e.g., *degree-one vertex reduction and degree-two vertex reductions) have been effectively used for low-degree vertices* to reduce the graph size while preserving the maximality of independent sets [19]. This is because a vertex with low degree is easier to be determined to be included or excluded in a maximum independent set, especially when the degree is 1 or 2. *(iii) High-degree vertices are less likely to be in a maximum independent set* [19], and removing/peeling high-degree vertices can further sparsify the graph [29].

Our Approaches. Based on the above observations, it is a key to effectively deal with the vertices with very low degrees and very high degrees. Unlike the existing heuristics [30] that iteratively add the vertex with the lowest degree into the solution set with an eye on minimizing the I/O cost, we develop a Reducing-Peeling framework for approximately computing a maximum independent set of a graph. Our framework iteratively applies reduction rules to reduce the graph size (*i.e.*, Reducing) by determining whether or not the vertices with degree 1, 2, or 3 should be included in a maximum independent set, as well as temporarily removes the vertex with the highest degree (*i.e.*, Peeling) if no reduction rules can be applied on the current graph. Note that, a temporarily removed vertex may be added to the solution set at the end of the algorithm.

Secondly, we devise two baseline algorithms, BDOne and BDTwo, by integrating the existing reduction rules (for handling degree-one and degree-two vertices, respectively) in [21] into our new Reducing-Peeling paradigm. The observation behind BDOne is that for a degree-one vertex u, there is a maximum independent set that includes u (*i.e.*, excludes the neighbors of u) [21]; thus, we remove the neighbors of u from the graph. For example, consider the graph in Figure 1, BDOne first removes v_9 from the graph by the degree-one reduction (Reducing). Then, highest-degree vertices v_4 and v_3 are consecutively removed (Peeling) since the degree-one reduction cannot be applied. Now, both v_1 and v_2 become degree-one vertices; we remove v_2 from the graph by the reduction rule. Similarly, v_8 and v_6 are removed by Peeling and Reducing, respectively. Thus, BDOne computes the independent set $\{v_1, v_5, v_7, v_{10}\}$ of size 4. BDTwo may often output a larger independent set than BDOne as a result of applying the reduction rules for both degree-one and degree-two vertices. For example, reconsider the graph in Figure 1. Firstly, v_9 is removed by the degree-one reduction. Secondly, v_6, v_7, v_8 are contracted into a single super-vertex by the degree-two reduction [21] since at this moment it is hard to determine which vertex should be included in a maximum independent set. Then, the super-vertex for $\{v_6, v_7, v_8\}$ becomes a degree-one vertex. Continuing BDTwo, the super-vertex is added into the solution set and v_5 is removed. Next, according to the degree-two reduction [21], $\{v_2, v_3\}$ can be removed (details can be found in Section 2.1). Thus, BDTwo obtains a maximum independent set $\{v_1, v_4, v_6, v_8, v_{10}\}$ of size 5. The time complexity of BDOne is linear to m, the number of edges in G, while BDTwo is not a linear-time algorithm and consumes more memory space (See Table 1).

Thirdly, we design new reduction rules to more efficiently process degree-two vertices than [21] does, and propose a linear-time algorithm, LinearTime, which takes similar time and space as BDOne but computes independent sets of similar sizes as BDTwo. Regarding the graph in Figure 1, after v_{10} is added into the solution set and v_9 is removed from the original graph by the degree-one reduction,

v_5 in the remaining graph is removed by our new reduction rule, and finally v_7 and $\{v_2, v_3\}$ are removed by the degree-one reduction and our new reduction rule, respectively. Thus, LinearTime also obtains $\{v_1, v_4, v_6, v_8, v_{10}\}$ but runs in linear time.

Fourthly, we aim to remove more vertices so that the algorithm may generate a more accurate solution. We propose a near-linear time algorithm, NearLinear, to further improve the solution quality by incorporating the dominance reduction [21]. We prove that the dominance reduction captures many other reduction rules, and develop efficient triangle maintenance-based techniques to incrementally and iteratively apply the dominance reduction. Consider a modified version of the graph in Figure 1 by removing v_{10} and connecting v_9 to v_1, v_5, v_6, v_7, v_8. The minimum degree of the graph is 3, and the reduction rules for handling degree-one and degree-two vertices cannot be applied. Nevertheless, we show that v_9 is dominated by v_5 and thus v_9 can be removed from the graph; subsequently, the remaining graph can be solved by LinearTime. A summary of our four approaches is given in Table 1.

Finally, we extend our techniques to improve the accuracy of the iterated local search technique ARW and also accelerate ARW and its variants. As a byproduct, we are also able to deliver a tighter upper bound than the existing ones to guide an exact computation.

Contributions. Our main contributions are summarized as follows.

- We develop a Reducing-Peeling framework for efficiently and effectively computing an approximate maximum independent set of a graph (Section 3.1).
- We design two baseline algorithms, BDOne (Section 3.2) and BDTwo (Section 3.3), by applying the existing reduction rules for handling degree-one and degree-two vertices.
- We propose a linear-time algorithm, LinearTime, based on our newly designed reduction rules for efficiently handling degree-two vertices (Section 4).
- We devise a near-linear time algorithm, NearLinear, to further improve the solution quality by proposing new efficient techniques such that we can iteratively and incrementally apply the dominance reduction in near-linear time (Section 5).
- We extend our techniques to accelerate the iterated local search algorithm ARW (Section 6).

We conduct extensive empirical studies on large real and synthetic graphs in Section 7. The results show that (i) even the baseline algorithm BDOne computes a much larger independent set than Greedy and DU. (ii) LinearTime has a similar running time as BDOne but improves the independent set size. (iii) NearLinear further improves the solution quality with a slightly increased running time and can find near-maximum or even maximum independent sets. (iv) By combining NearLinear with ARW, we also significantly speed up ARW. Duo to space limits, proofs of lemmas and theorems are presented in Section A.1 in Appendix.

Related Works. We categorize the related works as computing independent sets/vertex covers, and computing cliques.

Computing Independent Sets/Vertex Covers. As discussed above, exact exponential-time algorithms for computing a maximum independent set are studied in [1, 21, 38], approximation techniques with accuracy guarantees are investigated in [7, 20, 25], and heuristic algorithms without accuracy guarantees are developed in [2, 11, 19, 27, 28]. The existing techniques either are only able to output independent sets of limited quality or take an excessive long time to find a high-quality independent set. In this paper, we develop a new Reducing-Peeling paradigm and propose efficient techniques such that we can compute a much higher quality independent set, in linear or near-linear time, than the existing techniques.

The problems of enumerating maximal independent sets and computing independent sets in a streaming environment are also studied in [10] and [17], respectively. These techniques cannot be applied since they only focus on theoretical complexities. Recently, computing large independent sets in an I/O efficient manner is also investigated in [30], which is included in our experiments.

Computing Cliques. The maximum clique problem, which is to find a clique (*i.e.*, every two vertices in it are connected by an edge) of the maximum cardinality in a graph, is also an NP-hard problem. Exponential-time exact methods are designed in [31, 33, 35] based on the branch-and-bound framework. Although the exact algorithms can find the maximum clique, they take excessive long running time; as a result, they can only process small graphs. On the other hand, various approximation and heuristic algorithms [6, 9] have also been devised aiming to compute a good solution within an acceptable running time. Recently, distributed maximum clique computation and efficient maximal clique enumeration are studied in [37] and [12, 15], respectively. All these techniques cannot be applied to our problem, even though computing independent sets and computing cliques are theoretically equivalent; that is, $I \subseteq V$ is an independent set of $G = (V, E)$ if and only if it is a clique of $\overline{G} = (V, \overline{E})$, where $(u, v) \in \overline{E}$ if and only if $(u, v) \notin E$. This is because the complement of a sparse graph is an extremely dense graph of size $O(n^2)$, and in practice we are facing sparse graphs.

2. PRELIMINARY

In this paper, we focus on *an unweighted undirected graph* $G = (V, E)$ [24], where V is the set of vertices and E is the set of edges. We denote the number of vertices, $|V|$, and the number of edges, $|E|$, in G by n and m, respectively. Let $(u, v) \in E$ denote an edge between u and v; u (resp. v) is said to be connected to and a neighbor of v (resp. u). The *neighborhood* (*i.e.*, the set of neighbors) of u is $N(u) = \{v \in V \mid (u, v) \in E\}$, and the *degree* of u is $d(u) = |N(u)|$. For ease of presentation, we simply refer to an unweighted undirected graph as a graph.

Given a vertex subset $S \subseteq V$ of a graph $G = (V, E)$, we may modify G based on two operations on S. (i) *(Deletion)* We use $G \backslash S$ to denote the graph obtained from G by removing all vertices of S and their associated edges. (ii) *(Contraction)* We use G/S to denote the graph obtained from G by contracting all vertices of S into a single supervertex. Specifically, we remove all vertices of S from the graph and add a new vertex u' into the graph, and there is an edge between u' and any remaining vertex u if and only if u is previously connected to a vertex of S. We also use $G \cup \{(u, v)\}$ to denote the graph obtained from G by adding edge (u, v).

Definition 2.1: A vertex subset $I \subseteq V$ of a graph $G = (V, E)$ is an **independent set** if for any two vertices u and v in I, there is no edge between u and v in G; that is, $(u, v) \notin E, \forall u, v \in I$.

The *size of an independent set* is the number of vertices in it. An independent set I is a *maximal independent set* if there does not exist a superset I' of I (*i.e.*, $I' \supset I$) such that I' is also an independent set. An independent set I of G is a *maximum independent set* if its size is the largest among all independent sets of G, and this size is called the *independence number* of G, denoted $\alpha(G)$. Note that, the maximum independent set of G is not unique; that is, there may exist several independent sets of G with size $\alpha(G)$.

For example, consider the graph in Figure 2. $\{v_2, v_6\}$ is a maximal independent set, $\{v_1, v_3, v_4\}$ is a maximum independent set, and the independence number of the graph is 3.

Problem Statement. Given a graph $G = (V, E)$, we study the problem of efficiently computing a high-quality independent set of G.

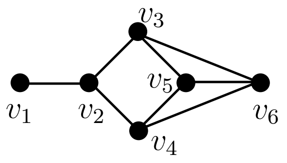


Figure 2: Example graph

Vertex Cover. A subset $C \subset V$ of a graph $G = (V, E)$ is a *vertex cover* if every edge of E has at least one end-point in C. It is well-known [34] that $C \subset V$ is a (minimum) vertex cover of G if and only if $V \backslash C$ is a (maximum) independent set of G. Thus, computing a maximum independent set is equivalent to computing a minimum vertex cover, and *the techniques proposed in this paper can be directly applied to compute a high-quality vertex cover.*

Graph Representation. In this paper, we use adjacency arrays to represent a graph. That is, we store the neighborhood of a vertex consecutively in an array. Moreover, we store the adjacency arrays of all vertices in a single large array, where a pointer is maintained for each vertex to indicate the start position of its adjacency array. Thus, a graph is represented by $2m + n$ integers in main memory.

2.1 Reduction Rules

The problem of computing a maximum independent set of a graph is NP-hard [23]. Nevertheless, in view of the importance of this problem, many techniques have been devised for accelerating the computation of a maximum independent set [1] or for heuristically computing a large independent set by iteratively improving the solution quality [19, 28]. Among them, applying reduction rules have recently been found to be very promising. The main reason is that applying reduction rules can (significantly) reduce the graph instance while preserving the maximum independent set.

In this paper, we also make use of reduction rules. However, applying the full set of reduction rules in [1] incurs significant processing time [19]. Thus, we mainly focus on the simple reduction rules that are designed for handling vertices of degree ≤ 2 and can be applied efficiently. Obviously, for a degree-zero vertex u, any maximum independent set must include u. Thus, *to include a vertex u into an independent set, we can simply remove all its neighbors (*i.e., *$N(u)$) from the graph to make it a degree-zero vertex.* The reduction rules for handling degree-one and degree-two vertices are illustrated by the following two lemmas, respectively.

Lemma 2.1: ***(Degree-one (Vertex) Reduction [21])*** *For a degree-one vertex u in G with neighbor v, there exists a maximum independent set of G that includes u; thus, we can remove v from G as shown in Figure 3(a), and $\alpha(G) = \alpha(G\backslash\{v\})$.*

Lemma 2.2: ***(Degree-two (Vertex) Reductions)*** *For a degree-two vertex u in $G = (V, E)$ with neighbors v and w:*
(1) (Isolation [19]) *$(v, w) \in E$. There exists a maximum independent set of G that includes u; thus, we can remove v and w from G as shown in Figure 3(b), and $\alpha(G) = \alpha(G\backslash\{v, w\})$.*
(2) (Folding [21]) *$(v, w) \notin E$. We have $\alpha(G) = \alpha(G/\{u, v, w\}) + 1$ and we contract $\{u, v, w\}$ into a supervertex uvw as shown in Figure 3(c). Let I be a maximum independent set of $G/\{u, v, w\}$. If $uvw \in I$ then $(I\backslash uvw) \cup \{v, w\}$ is a maximum independent set of G, otherwise $I \cup \{u\}$ is a maximum independent set of G.*

For simplicity, we refer to the degree-one vertex reduction and degree-two vertex reductions as degree-one reduction and degree-two reductions, respectively. We also refer to the two sub-rules in Lemma 2.2 as *degree-two isolation* and *degree-two folding*. We say a vertex satisfies a reduction rule if the reduction rule can be applied on this vertex; for example, a vertex satisfies the degree-one reduction if it is of degree one.

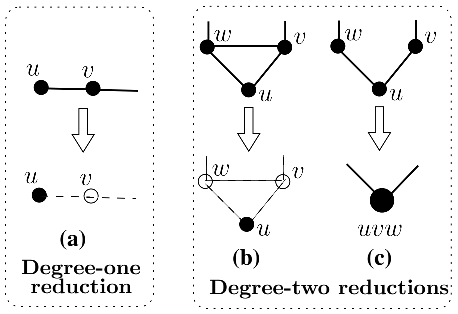


Figure 3: Degree-one and degree-two vertex reductions

3. A NEW FRAMEWORK AND TWO BASELINE SOLUTIONS

In this section, we develop a Reducing-Peeling framework for computing a high-quality independent set of a graph in Section 3.1, based on which we design two baseline algorithms in Sections 3.2 and 3.3, respectively.

3.1 The Reducing-Peeling Framework

Reduction rules were originally proposed in [21, 38] for reducing the time complexity of exact exponential-time algorithms in theory. Recently, they are also found to be successful in accelerating exact exponential-time algorithms in practice [1] and in accelerating heuristic local search and evolutionary algorithms [19, 28]. We explain the reasons as follows. (1) Real networks are usually power-law graphs such that the degree distribution follows a power-law distribution [4]; that is, there are many low-degree vertices and a few high-degree vertices. (2) Reduction rules can effectively handle low-degree vertices [19] (*e.g.*, reduction rules in Lemmas 2.1 and 2.2) to reduce the graph size while preserving the maximality of independent sets. Therefore, after iteratively applying reduction rules until no rule can be applied, we usually will get a small resulting graph, which is called the *kernel graph*; this process is referred to as *kernelization*. That is, the effective size of the graph that needs to be processed subsequently is significantly reduced.

However, there are two major issues that prevent the algorithms in [1, 28] from running on large graphs. *(i) Iteratively and repeatedly trying all the reduction rules on every vertex is time-consuming*. Note that, once the graph changes (*i.e.*, one reduction rule is applied), the techniques in [1, 28] need to try all the reduction rules on every vertex again (*i.e.*, repeatedly). *(ii) The kernel graph may still be too large for a large input graph*; thus, algorithms with high time complexities still take an excessive long time to run on the resulting kernel graph.

The Reducing-Peeling Framework. In this paper, we develop a new framework to remedy the above issues. Firstly, *we propose efficient techniques to incrementally apply reduction rules*, which are the main focuses of the following sections. Secondly, *we propose an effective strategy to destroy the equilibrium of the current kernel graph such that the reduction rules can be applied again.* To do so, we define the following inexact reduction rule.

Definition 3.1: **(Inexact Reduction)** Given a graph G, we remove/peel the vertex with the highest degree from G.

The inexact reduction rule is designed based on the intuition that *high-degree vertices are less likely to be in a maximum independent set* [19]; for example, if a high-degree vertex is added into the independent set, then all its neighbors, which are of a large quantity, are ruled out from the independent set. Furthermore, removing high-degree vertices from a graph sparsifies the graph [29]; that is, given a kernel graph on which no reduction rule can be applied, it is usually the case that we can apply the reduction rules again after removing one or a few high-degree vertices.

We refer to the reduction rules in [1] as *exact reduction rules* to distinguish them from the above inexact reduction rule. The exact reduction rules preserve the optimal solution while the inexact reduction rule may not; that is, it is possible that $\alpha(G) \neq \alpha(G\backslash\{u\})$ where u is the vertex with the highest degree in G. Consequently, we apply the inexact reduction rule as few as possible.

Algorithm 1: Reducing-Peeling Framework

```
Input: A graph $G = (V, E)$, and a set of exact reduction rules $\mathcal{R}$
Output: A maximal independent set $I$ in $G$

1 while $G$ contains edges do
2     if a reduction rule in $\mathcal{R}$ can be applied on a vertex $u$ then
3         Apply the exact reduction rule on $u$;     /* Reducing */;
4     else Apply the inexact reduction rule;       /* Peeling */;
5 $I \leftarrow$ the set of degree-zero vertices in $G$;
6 Extend $I$ to be a maximal independent set;
7 return $I$;
```

We propose the Reducing-Peeling framework, shown in Algorithm 1. We iteratively apply exact reduction rules (*i.e.*, Reducing) or the inexact reduction rule (*i.e.*, Peeling) until the graph G contains no edges (Lines 1–4), where the inexact reduction rule is applied only when the exact reduction rules can no longer be applied on the current graph. The set I of degree-zero vertices is an independent set (Line 5). Note that, I may not be a maximal independent set, since it is possible that for a vertex that has been removed by the inexact reduction rule, all its neighbors are not in I. Thus, we extend I to be a maximal independent set (Line 6). That is, the inexact reduction rule only temporarily removes the highest-degree vertex, and the vertex may be added back to the independent set later if none of its neighbors is in the independent set. As each reduction rule will reduce the graph size by at least one vertex, Algorithm 1 terminates after at most n iterations. Moreover, we can report I to be a maximum independent set if the inexact reduction rule is not applied during the algorithm execution.

Discussions. We discuss the novelty of our framework, an alternative inexact reduction rule, and applying the reduction rules in [1].

Novelty of Our Framework. Despite that the idea of removing high-degree vertices is also considered in [19], our framework is novel in the following aspects. Firstly, we apply the inexact reduction rule (*i.e.*, remove the highest-degree vertex) only when the exact reduction rules can no longer be applied (*i.e.*, Reducing has a higher priority than Peeling), while the techniques in [19] heuristically remove a fixed portion (*e.g.*, 1%) of the high-degree vertices. Secondly, after removing high-degree vertices, we apply the exact reduction rules again on the resulting graph, while the techniques in [19] run local search on the resulting graph. Moreover, we design new reduction rules and propose efficient techniques for incrementally applying reduction rules in this paper.

Alternative Inexact Reduction. An alternative definition of the inexact reduction rule is adding the vertex with the minimum degree into the independent set and removing all its neighbors. Actually, DU is such an algorithm with this alternative definition, where $\mathcal{R}$ is the degree-one reduction. However, this strategy does not work well since a low-degree vertex has an equal probability to be in a maximum independent set or not in. Our experiments in Section 7 show that the baseline algorithm within our framework by letting $\mathcal{R}$ be the degree-one reduction finds significantly larger independent sets than DU.

Applying the Reduction Rules and the Techniques in [1]. Within our framework, an algorithm can be designed by letting $\mathcal{R}$ be the full set of reduction rules in [1] and using the existing techniques in [1] to apply the reduction rules. However, this is not a good idea. Firstly, some of the reduction rules take a long time to apply (*e.g.*, the unconfined reduction rule), and some of the reduction rules may dynamically enlarge the neighborhood of a vertex; thus, larger running time and memory space are consumed for applying all the reduction rules (see Section 7). Secondly, efficient techniques for incrementally applying reduction rules are not studied. Note that, the graph is dynamically changing after applying each reduction rule. In [1], even if a reduction rule cannot be applied on the current graph, it still needs to try the reduction rule on all the vertices to confirm that it cannot be applied. As a result, the techniques in [1] take an excessive long time to compute the kernel graph as verified by our experiments in Section 7.

3.2 An Efficient Baseline Algorithm

Following the Reducing-Peeling framework, we propose an efficient baseline algorithm for computing a high-quality independent set in linear time by letting $\mathcal{R}$ be the degree-one reduction. The pseudocode is shown in Algorithm 2, denoted BDOne. We first let $d(v)$ be the degree of v (Line 1), and let I, $V_{=1}$, and $V_{\geq 2}$ be the sets of vertices in G of degree zero, one, and at least two (Line 2), respectively. Then, we go to iterations until both $V_{=1}$ and $V_{\geq 2}$ are empty (Lines 3–7); that is, the graph contains no edges. If the degree-one reduction can be applied (*i.e.*, $V_{=1} \neq \emptyset$) on a vertex u, then we remove the unique neighbor v of u from G by DeleteVertex (Line 5). Otherwise, we apply the inexact reduction rule on the highest-degree vertex by removing it from G (Line 7).

Algorithm 2: BDOne

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

1 for each $v \in V$ do $d(v) \leftarrow$ the degree of $v$ in $G$;
2 Let $I$, $V_{=1}$, and $V_{\geq 2}$ be the sets of vertices in $G$ of degree zero, one, and
  at least two, respectively;
3 while $V_{=1} \neq \emptyset$ or $V_{\geq 2} \neq \emptyset$ do
4     if $V_{=1} \neq \emptyset$ then
5         Delete from $G$ the neighbor $v$ of a vertex $u \in V_{=1}$ by invoking
          DeleteVertex($v$);     /* Degree-one reduction */;
6     else
7         Delete from $G$ the vertex $u$ with the highest degree by
          invoking DeleteVertex($u$);  /* Inexact reduction */;
8 Extend $I$ to be a maximal independent set;
9 return $I$;

  Procedure DeleteVertex($v$)
10 for each neighbor $w$ of $v$ in $G$ do
11     $d(w) \leftarrow d(w) - 1$;          /* Remove edge $(v, w)$ */;
12     if $d(w) = 1$ then Remove $w$ from $V_{\geq 2}$ and add $w$ into $V_{=1}$;
13     else if $d(w) = 0$ then Remove $w$ from $V_{=1}$ and add $w$ into $I$;
14 Remove $v$ from $G$, $V_{=1}$ and $V_{\geq 2}$;
```

The procedure DeleteVertex removes a vertex v from the graph G and updates the three sets, I, $V_{=1}$, $V_{\geq 2}$, accordingly. When removing a vertex from G, we also remove all its adjacent edges from G. Thus, for each neighbor w of v, the degree $d(w)$ of w reduces by 1 (Line 11). Moreover, if $d(w)$ becomes 1, then we add w into $V_{=1}$

and remove w from $V_{\geq 2}$ (Line 12); this is because $d(w)$ must be 2 before the update. If $d(w)$ becomes 0, then we add w into I and remove w from $V_{=1}$ (Line 13). Lastly, we also remove v from G, $V_{=1}$, and $V_{\geq 2}$ if it is in the corresponding set (Line 14).

Running Example. Consider the graph in Figure 2. After initialization, $V_{=1} = \{v_1\}$ and $V_{\geq 2} = \{v_2, v_3, v_4, v_5, v_6\}$. In the first iteration of the while loop (Lines 3–7 of Algorithm 2), we apply the degree-one reduction on vertex v_1 by removing its neighbor v_2 from G; thus, we have $I = \{v_1\}$, $V_{=1} = \emptyset$ and $V_{\geq 2} = \{v_3, v_4, v_5, v_6\}$ with $d(v_3) = d(v_4) = 2$ and $d(v_5) = d(v_6) = 3$. In the second iteration, as the degree-one reduction cannot be applied, we apply the inexact reduction by removing v_6 from G; now, $I = \{v_1\}$, $V_{=1} = \{v_3, v_4\}$, and $V_{\geq 2} = \{v_5\}$. In the third iteration, we apply the degree-one reduction on vertex v_3 by removing its neighbor v_5 from G; we obtain $I = \{v_1, v_3, v_4\}$ and the algorithm terminates since no edge exists.

Analysis and Implementation Details. *Algorithm 2 runs in linear time (*i.e., *$O(m)$) as follows.* Firstly, Lines 1–2 take $O(m)$ time. Secondly, all the degree-one reductions run in $O(m)$ total time since DeleteVertex for a vertex v takes $d(v)$ time. For time-efficiency consideration, we mark a vertex as deleted rather than actually deleting it from G. Thus, the representation of the input graph remains unchanged during the execution of the algorithm, and *the space complexity of Algorithm 2 is* $2m + O(n)$ by the graph representation in Section 2.

Next, we show that retrieving the vertex with the highest degree at Line 7 can be implemented in $O(m)$ total time for all such operations, where vertex degree dynamically changes (*e.g.*, at Line 11). The bin-sort [18] like data structure in [13] can be used for this purpose based on the fact that the degrees are integers in the range of 1 to n. The data structure has n bins, one for each distinct degree value; bin i consists of a *doubly-linked list* linking together all vertices of degree i. Updating a vertex's degree in the data structure can be accomplished in $O(1)$ time by removing it from one doubly-linked list and adding it to another. The total n operations of retrieving the vertex with the highest degree can be achieved in $O(n)$ time by maintaining the highest degree of the remaining vertices, which is non-increasing. Furthermore, to improve the practical efficiency, we propose to update a vertex's degree in the data structure lazily (*i.e.*, only when it is going to be chosen as the vertex with the highest degree at Line 7), since the degree of a vertex can only decrease. As a result, *singly-linked lists*, which consumes $2n$ space, rather than doubly-linked lists are sufficient in the data structure; moreover, the number of update operations of the data structure is significantly reduced in practice.

3.3 An Effective Baseline Algorithm

We also propose an effective baseline algorithm by letting $\mathcal{R}$ be the degree-one reduction and the degree-two reductions; intuitively it can find a larger independent set than BDOne as a result of applying more exact reduction rules. The pseudocode is shown in Algorithm 3, denoted BDTwo. We first invoke Initialization (Lines 9–10) to initialize $I, V_{=1}, V_{=2}$ and $V_{\geq 3}$, similar to the initialization in Algorithm 2. Then, we go to iterations (Lines 3–5) until the graph contains no edges. In an iteration, if $V_{=1}$ is not empty, then we apply the degree-one reduction (Line 11); if $V_{=1}$ is empty but $V_{=2}$ is not empty, then we apply the degree-two reduction (Lines 12–15); otherwise, we apply the inexact reduction (Line 16).

The procedure DegreeTwo-Reduction follows Lemma 2.2. If there is an edge between the two neighbors v and w of u (*i.e.*, degree-two isolation), then we remove v and w from the graph (Lines 13–14). Otherwise, we apply degree-two folding by contracting $\{u, v, w\}$ which is equivalent to first removing u and then

Algorithm 3: BDTwo

```
Input: A graph $G = (V, E)$
Output: A maximal independent set $I$ in $G$
1  Initialization();
2  while $V_{=1} \neq \emptyset$ or $V_{=2} \neq \emptyset$ or $V_{\geq 3} \neq \emptyset$ do
3      if $V_{=1} \neq \emptyset$ then DegreeOne-Reduction();
4      else if $V_{=2} \neq \emptyset$ then DegreeTwo-Reduction();
5      else Inexact-Reduction();
6  Backtrack the contraction operations to get the correct $I$;
7  Extend $I$ to be a maximal independent set;
8  return $I$;

   Procedure Initialization()
9  for each $v \in V$ do $d(v) \leftarrow$ the degree of $v$ in $G$;
10 Let $I, V_{=1}, V_{=2}$, and $V_{\geq 3}$ be the sets of vertices in $G$ of degree zero,
   one, two, and at least three, respectively;

   Procedure DegreeOne-Reduction()
11 Delete from $G$ the neighbor $v$ of a vertex $u \in V_{=1}$ by invoking
   DeleteVertex($v$);

   Procedure DegreeTwo-Reduction()
12 Let $v, w$ be the two neighbors of a vertex $u \in V_{=2}$;
13 if there is an edge between $v$ and $w$ in $G$ then
14     Delete vertices $v$ and $w$ by invoking DeleteVertex($v$) and
       DeleteVertex($w$);      /* Degree-two isolation */;
15 else DeleteVertex($u$); Contract($v, w$); /* Degree-two folding */;

   Procedure Inexact-Reduction()
16 Delete from $G$ the vertex $u$ with the highest degree by invoking
   DeleteVertex($u$);

   Procedure DeleteVertex($v$)
17 for each neighbor $w$ of $v$ in $G$ do
18     $d(w) \leftarrow d(w) - 1$;          /* Remove edge $(v, w)$ */;
19     if $d(w) = 2$ then Remove $w$ from $V_{\geq 3}$ and add $w$ into $V_{=2}$;
20     else if $d(w) = 1$ then Remove $w$ from $V_{=2}$ and add $w$ into $V_{=1}$;
21     else if $d(w) = 0$ then Remove $w$ from $V_{=1}$ and add $w$ into $I$;
22 Remove $v$ from $G$, $V_{=1}$, $V_{=2}$, and $V_{\geq 3}$;

   Procedure Contract($v, w$)
23 Add edges, if not previously exist, between $w$ and neighbors of $v$;
24 Update $w$ to be in the correct set, $V_{=1}$, $V_{=2}$, or $V_{\geq 3}$;
25 Remove $v$ from $G$, $V_{=1}$, $V_{=2}$, and $V_{\geq 3}$;
```

contracting $\{v, w\}$ (Line 15). Note that, to obtain the actual independent set, we also need to backtrack the contraction operations by following Lemma 2.2 (Line 6).

Running Example. Reconsider the graph in Figure 2. After initialization, $I = \emptyset$, $V_{=1} = \{v_1\}$, $V_{=2} = \emptyset$, and $V_{\geq 3} = \{v_2, v_3, v_4, v_5, v_6\}$. In the first iteration of the while loop (*i.e.*, Lines 3–5 of Algorithm 3), we apply the degree-one reduction on v_1, and we obtain $I = \{v_1\}$, $V_{=1} = \emptyset$, $V_{=2} = \{v_3, v_4\}$, and $V_{\geq 3} = \{v_5, v_6\}$. In the second iteration, we apply the degree-two reduction on vertex v_3; as there is an edge between the two neighbors v_5 and v_6 of v_3, we remove v_5 and v_6 from G and obtain $I = \{v_1, v_3, v_4\}$, $V_{=1} = V_{=2} = V_{\geq 3} = \emptyset$. The algorithm terminates. Here, we can report $\{v_1, v_3, v_4\}$ as a maximum independent set since the inexact reduction rule is not applied.

Analysis and Implementation Details. While it is easy to obtain an upper bound of the time complexity of BDTwo as $O(n \times m)$, it is hard to give a tighter analysis of the time complexity of BDTwo. We prove in the following theorem that it is bounded below by $\Omega(m + n \log n)$. Thus, BDTwo is not a linear-time algorithm.

Theorem 3.1: *The time complexity of Algorithm 3 is* $\Omega(m+n \log n)$.

Due to the contraction operation, the neighborhood size (*i.e.*, degree) of a vertex may increase in BDTwo (*e.g.*, Line 23). Thus, we need the bin-sort like data structure discussed in Section 3.2 to be implemented as doubly-linked lists. Moreover, we need a graph representation that supports dynamically enlarging the neighborhood of a vertex. The graph representation discussed in Section 2 does not support this operation. As a result, we represent the graph in BDTwo as adjacency lists, where the two directions of each edge have mutual references, and this representation consumes $6m + n$ space. Therefore, the space complexity of BDTwo is $6m + O(n)$, which is also larger than that of BDOne.

4. AN EFFECTIVE LINEAR-TIME ALGORITHM

BDTwo, as a result of additionally applying the degree-two reductions, is more effective (*i.e.*, can find a larger independent set) but has higher time and space complexities than BDOne. In this section, we propose degree-two path reductions to efficiently handle degree-two vertices and to remedy the deficiencies of BDTwo.

Degree-two Path Reductions. We propose new reduction rules for degree-two vertices in terms of degree-two paths.

Definition 4.1: A path P of G is a **degree-two path** if every vertex of P has degree two in G. A cycle C of G is a **degree-two cycle** if every vertex of C has degree two in G.

A degree-two path P is *maximal* if it is not a subpath of any other degree-two paths. Given a graph with minimum degree two, a degree-two path P is maximal if and only if the two vertices that are adjacent to the two end-points of P but are not in P have degree at least three. It is easy to see that every degree-two vertex is either in a degree-two cycle or in a maximal degree-two path, and moreover, the degree-two cycles and maximal degree-two paths form a partition of the set of all degree-two vertices. To efficiently handle degree-two vertices, we have the following lemma.

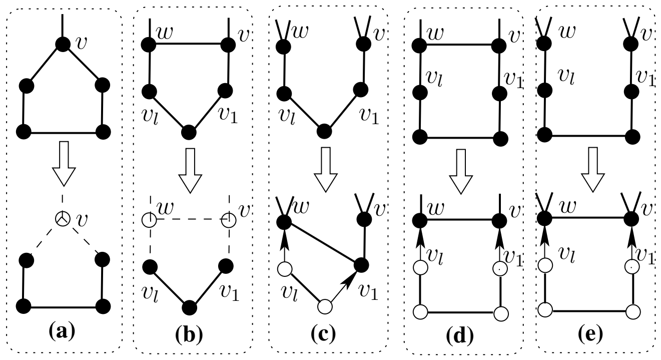


Figure 4: Degree-two path reductions

Lemma 4.1: **(Degree-two Path Reductions)** *Consider a graph $G = (V, E)$ with minimum degree two. For a degree-two cycle C in G and an arbitrary vertex $v \in C$, there exists a maximum independent set of G that excludes v. Thus, we can remove v from G as shown in Figure 4(a) (here, we ignore the edge connecting v to the other part of G) and $\alpha(G) = \alpha(G\backslash\{v\})$.*

For a maximal degree-two path $P = (v_1, v_2, \ldots, v_l)$, let $v \notin P$ and $w \notin P$ be the unique vertices connected to v_1 and v_l, respectively. Denote the number of vertices in P by $l = |P|$. There are five cases.

(1) $v = w$**.** *There exists a maximum independent set of G that excludes v; thus, we can remove v from G as shown in Figure 4(a) and $\alpha(G) = \alpha(G\backslash\{v\})$. In the following cases, we assume $v \neq w$.*

(2) $|P|$ **is odd and** $(v, w) \in E$**.** *There exists a maximum independent set of G that excludes v and w; thus, we can remove v and w from G as shown in Figure 4(b) and $\alpha(G) = \alpha(G\backslash\{v, w\})$.*

Algorithm 4: LinearTime

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

Initialization();
Initialize a stack $S$ to be empty;
while $V_{=1} \neq \emptyset$ or $V_{=2} \neq \emptyset$ or $V_{\geq 3} \neq \emptyset$ do
    if $V_{=1} \neq \emptyset$ then DegreeOne-Reduction();
    else if $V_{=2} \neq \emptyset$ then DegreeTwoPath-Reduction();
    else Inexact-Reduction();
Iteratively pop a vertex $u$ from the stack $S$, and add $u$ to $I$ if none of its
    neighbors is in $I$;
Extend $I$ to be a maximal independent set;
return $I$;

Procedure DegreeTwoPath-Reduction()
$u \leftarrow$ a vertex in $V_{=2}$;
Find the maximal degree-two path/cycle $P = (v_1, \ldots, v_l)$ containing $u$;
if $P$ is a cycle then DeleteVertex($u$);
else
    Let $v, w \notin P$ be the two vertices connected to $v_1, v_l$, respectively;
    if $v = w$ then DeleteVertex($v$);        /* Fig. 4(a) */;
    else if the number of vertices in $P$ is odd then
        if there is an edge between $v$ and $w$ in $G$ then
            DeleteVertex($v$); DeleteVertex($w$); /* Fig. 4(b) */;
        else
            Remove all vertices except $v_1$ of $P$ from $G$, remove all
            vertices of $P$ from $V_{=2}$, and add an edge between $v_1$ and
            $w$;                /* Fig. 4(c) */;
            Push vertices $v_l, \ldots, v_2$, obeying the order, into $S$;
    else
        Remove all vertices of $P$ from $G$ and $V_{=2}$, and add an edge, if
        not exist, between $v$ and $w$; /* Figs. 4(d) and 4(e) */;
        Push vertices $v_l, \ldots, v_1$, obeying the order, into $S$;
```

(3) $|P|$ **is odd and** $(v, w) \notin E$**.** *There exists a maximum independent set of G that excludes either v_1 or w and includes half of the vertices $\{v_2, \cdots, v_l\}$; thus, we can remove $\{v_2, \ldots, v_l\}$ from G and add edge (v_1, w) as shown in Figure 4(c), and $\alpha(G) = \alpha(G\backslash\{v_2, \ldots, v_l\} \cup \{(v_1, w)\}) + \frac{|P|-1}{2}$.*

(4) $|P|$ **is even and** $(v, w) \in E$**.** *There exists a maximum independent set of G that excludes either v or w and includes half of the vertices $\{v_1, \cdots, v_l\}$; thus, we can remove $\{v_1, \ldots, v_l\}$ from G as shown in Figure 4(d), and $\alpha(G) = \alpha(G\backslash\{v_1, \ldots, v_l\}) + \frac{|P|}{2}$.*

(5) $|P|$ **is even and** $(v, w) \notin E$**.** *There exists a maximum independent set of G that excludes either v or w and includes half of the vertices $\{v_1, \cdots, v_l\}$; thus, we can remove $\{v_1, \ldots, v_l\}$ from G and add edge (v, w) as shown in Figure 4(e), and $\alpha(G) = \alpha(G\backslash\{v_1, \ldots, v_l\} \cup \{(v, w)\}) + \frac{|P|}{2}$.*

Note that, in case-3, case-4, and case-5 of Lemma 4.1, we first compute the maximum independent set I' of $G\backslash\{v_2, \cdots, v_l\} \cup \{(v_1, w)\}$, $G\backslash\{v_1, \cdots, v_l\}$, and $G\backslash\{v_1, \cdots, v_l\} \cup \{(v, w)\}$, respectively, and then extend I' by including a proper subset of vertices from $\{v_1, \cdots, v_l\}$, to get the maximum independent set of G. The subset of vertices in $\{v_1, \cdots, v_l\}$ that will be included to extend I' is determined by the vertices already in I'. A comparison between our degree-two path reductions and the existing degree-two vertex reductions is discussed in Section A.2 in Appendix.

The Algorithm LinearTime**.** Following the framework in Algorithm 1 and by letting $\mathcal{R}$ be the degree-one reduction and degree-two path reductions, we propose an effective linear-time algorithm for computing a large independent set of G. The pseudocode is shown in Algorithm 4, denoted LinearTime. The algorithm is similar to Algorithm 3, but we use the degree-two path reductions rather than the degree-two vertex reductions.

The degree-two path reductions are conducted at Lines 10–24. We first get an arbitrary degree-two vertex u from $V_{=2}$ (Line 10), and find the maximal degree-two path/cycle $P = (v_1, \ldots, v_l)$ containing u (Line 11). Then, we process P by following Lemma 4.1. If P is a cycle, then we remove u from G (Line 12); note that, the remaining vertices of P will all be iteratively processed by the degree-one reduction before processing the remaining part of G. Otherwise, P is a path, we process vertices of P in five cases. Let $v \notin P$ and $w \notin P$ be the two vertices connected to v_1 and v_l, respectively (Line 14). (1) If $v = w$, we remove v from G (Line 15) and then other vertices of P will be iteratively processed by the degree-one reduction. (2) If $|P|$ is odd and $(v, w) \in E$, we remove v and w from G (Line 17–18) and then other vertices of P will be iteratively processed by the degree-one reduction. (3) If $|P|$ is odd and $(v, w) \notin E$, we construct the graph G' by removing all vertices except v_1 of P from G and adding edge (v_1, w) to G (Line 20). We will first compute an independent set I' of G', and then add alternating vertices of $P \backslash \{v_1\}$ to I' to obtain an independent set of G; the decision of adding which vertices from $P \backslash v_1$ to I' is postponed to the end of the algorithm (Line 7) by pushing all vertices, $v_l, \ldots, v_2$, into a stack S (Line 21). Note that, in order to correctly put half of the vertices $v_l, \ldots, v_2$ into the independent set I at Line 7, we need to push these vertices into the stack S by obeying this order. Similarly, we process the two cases for P being even at Lines 23–24.

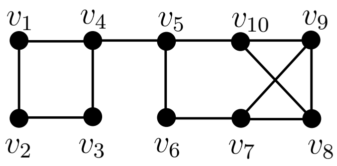


Figure 5: Example graph for LinearTime

Running Example. Consider the graph in Figure 5. After initialization, we have $I = \emptyset, V_{=1} = \emptyset, V_{=2} = \{v_1, v_2, v_3, v_6\}$, and $V_{\geq 3} = \{v_4, v_5, v_7, v_8, v_9, v_{10}\}$. In the first iteration of the while loop (*i.e.*, Lines 4–6 of Algorithm 4), we apply the degree-two path reduction on v_1. The maximal degree-two path is $P = (v_1, v_2, v_3)$, and the neighbors of v_1 and v_3 that are not in P are $v = w = v_4$. Thus, we remove v_4 from G (Line 15), and obtain $I = \emptyset, V_{=1} = \{v_1, v_3\}, V_{=2} = \{v_2, v_5, v_6\}$ and $V_{\geq 3} = \{v_7, v_8, v_9, v_{10}\}$. In the second iteration, we apply the degree-one reduction on v_1. We remove the unique neighbor v_2 of v_1 from G, and obtain $I = \{v_1, v_3\}, V_{=1} = \emptyset, V_{=2} = \{v_5, v_6\}$ and $V_{\geq 3} = \{v_7, v_8, v_9, v_{10}\}$. In the third iteration of the while loop, we apply the degree-two path reduction on v_5; $P = (v_5, v_6)$, $v = v_{10}$, $w = v_7$. As $(v, w) \notin E$ and $|P| = 2$, we push v_6, v_5 into the stack S, remove v_5, v_6 from G and add edge (v_{10}, v_7) into G. We obtain $I = \{v_1, v_3\}, V_{=1} = \emptyset, V_{=2} = \emptyset, V_{\geq 3} = \{v_7, v_8, v_9, v_{10}\}$ and $S = \{v_5, v_6\}$, and G is a clique of $\{v_7, v_8, v_9, v_{10}\}$. Now, assume we add v_{10} of the four-vertex clique to I, G contains no edges. We process the vertices in the stack S. Firstly, as the neighbor v_{10} of v_5 is in I, v_5 cannot be added to I; secondly, as none of v_6's neighbors is in I, we add v_6 to I. Finally, we obtain the maximum independent set $I = \{v_1, v_3, v_{10}, v_6\}$ of the graph in Figure 5. A detailed illustration of the above process is given in Figure 12 in the Appendix.

Analysis and Implementation Details. We show that LinearTime runs in linear time (*i.e.*, $O(m)$) as follows. Firstly, similar to BDOne in Section 3.2, we use the lazy update strategy and use singly-linked lists in the bin-sort like data structure. Secondly, instead of inserting a new edge (v, w) into G at Lines 20,23 (corresponding to Figure 4(c)(e)), we modify the adjacent edge (w, v_l) of w to be (w, v) and modify the adjacent edge (v, v_l) of v to be (v, w); in this way, the neighborhood size of a vertex will not increase during the algorithm execution, and thus we can use the graph representation in Section 2. Thirdly, each time of applying the degree-two path reduction, we reduce the graph size by removing $\geq 2(|P|-1)$ directed edges in $2|P|$ time with $|P| \geq 2$; the maximal degree-two path P containing a vertex u can be found by conducting a depth-first search from u by visiting only vertices of degree two, which runs in linear time to the number of vertices in P. Thus, the total running time of applying degree-two path reductions is bounded by $2m$. Note that, if a vertex forms a maximal degree-two path itself (*i.e.*, $|P| = 1$), we only check this once. Fourthly, in Algorithm 4, we also need to check whether there is an edge between v and w, which can be achieved in constant time by constructing a hash structure on all the edges. In practice, we iterate through the neighborhood of v, rather than building a hash structure, for checking whether w is a neighbor of v. Thus, the space complexity of LinearTime is $2m + O(n)$.

5. A NEAR-LINEAR-TIME ALGORITHM

In this section, we propose efficient techniques to incrementally apply more reduction rules such that we can find near-maximum independent sets in near-linear time.

Incrementally Applying The Dominance Reduction. Besides the reduction rules designed specifically for handling degree-one and degree-two vertices, there are also other reduction rules studied in [1, 21, 38] for handling vertices of arbitrary degree that satisfy some properties, *e.g.*, the dominance reduction.

Lemma 5.1: ***(Dominance Reduction)*** *[21] Vertex v* **dominates** *vertex u if $(v, u) \in E$ and all neighbors of v other than u are also connected to u* (i.e., $N(v) \backslash \{u\} \subseteq N(u)$). *If v dominates u, then there exists a maximum independent set of G that excludes u; thus, we can remove u from G as shown in Figure 6, and $\alpha(G) = \alpha(G \backslash \{u\})$.*

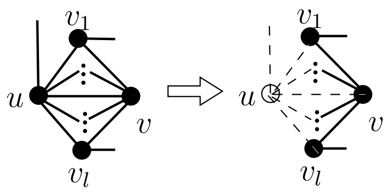


Figure 6: Dominance reduction

We prove some important properties of the dominance reduction in Section A.3 in Appendix. For example, the dominance reduction captures the *isolated vertex reduction* [19] which includes the degree-one reduction and the degree-two isolation as special cases, and *it can also handle two of the four cases for a degree-three vertex*. In view of the importance of the dominance reduction, we propose efficient techniques to incrementally apply the dominance reduction, based on triangle counts as shown in the lemma below.

Lemma 5.2: *Given a graph G, a vertex u dominates its neighbor v if and only if $\delta(u, v) = d(u) - 1$, where $\delta(u, v)$ denotes the number of triangles in G containing u and v and is called the triangle count of edge (u, v).*

Following Lemma 5.2, checking whether u dominates its neighbor v can be efficiently conducted by comparing $\delta(u, v)$ with $d(u) - 1$. Therefore, we propose to maintain a triangle count $\delta(u, v)$ for each edge (u, v) in G, and thus the dominance relationship between two adjacent vertices u and v can be checked in constant time after each updating of either $\delta(u, v)$ or $d(u)$ or $d(v)$. Recall that, the graph G changes each time after applying a reduction rule.

Algorithm 5: NearLinear

```
Input: A graph $G = (V, E)$
Output: A maximal independent set $I$ in $G$
1  Initialization();
2  Initialize a stack $S$ to be empty;
3  Compute a triangle count $\delta(u, v)$ for every edge $(u, v)$ in $G$;
4  $D \leftarrow \{u \in V \mid \exists(v, u) \in E$ s.t. $\delta(v, u) = d(v) - 1\}$;   /* $D$ is the dynamically maintained set of dominated vertices */;
5  while $V_{=2} \neq \emptyset$ or $D \neq \emptyset$ or $V_{\geq 3} \neq \emptyset$ do
6      if $V_{=2} \neq \emptyset$ then DegreeTwoPath-Reduction();
7      else if $D \neq \emptyset$ then
8          Pop a vertex $u$ from $D$ and remove $u$ from $G$ if $u$ is dominated by one of its neighbors;   /* Dominance reduction */;
9      else Inexact-Reduction();
10 Iteratively pop a vertex $u$ from the stack $S$, and add $u$ to $I$ if none of its neighbors is in $I$;
11 Extend $I$ to be a maximal independent set;
12 return $I$;
```

The Algorithm NearLinear**.** Following the framework in Algorithm 1 and by letting $\mathcal{R}$ be the degree-two path reductions and the dominance reduction, we propose a near-linear time algorithm for computing a near-maximum independent set. Recall that, the degree-one reduction is captured by the dominance reduction; thus, we do not include the degree-one reduction into $\mathcal{R}$. The pseudocode is shown in Algorithm 5, denoted NearLinear. Firstly, we do the initialization (Lines 1–2) same as that in Algorithm 4, compute a triangle count $\delta(u, v)$ for every edge (u, v) in G (Line 3), and maintain in D the set of dominated vertices (Line 4); a vertex is put into D if it is dominated by one of its neighbors. Then, we go to iterations (Lines 6–9) to reduce the graph until it contains no edges. In an iteration, we first try the degree-two path reductions (Line 6), then try the dominance reduction (Line 7–8), and lastly apply the inexact reduction (Line 9) if neither of the two exact reductions can be applied. One thing to notice is that before removing a vertex $u \in D$ from G, we need to check whether it is still dominated by one of its neighbors in the current graph (Line 8). This is because, two vertices u and v can mutually dominate each other as discussed in Section A.3 in Appendix, and it is possible that, after removing u that dominates v, v is no longer dominated by any vertex in the resulting graph. As the order of removing dominated vertices in D from the graph does not matter as discussed in Section A.3 in Appendix, all dominated vertices will be removed from the graph by the algorithm.

When removing a vertex from G, we also need to maintain the triangle counts for edges and maintain the set D of dominated vertices. It is easy to see that, when deleting u, only the edges that form triangles with u will have their triangle counts changed (*i.e.*, decrease by 1), and moreover, only the two-hop neighbors (*i.e.*, neighbors of neighbors) of u may change from not dominated to dominated vertices. For example, consider the neighbor w of $v \in N(u)$, if $w \in N(u)$, then $d(v)$, $d(w)$ and $\delta(v, w)$ all decrease by the same amount 1; otherwise, w is a two-hop neighbor of u and w may become dominated by v since $d(v)$ decreases by 1 while $\delta(v, w)$ remains the same. Note that, in DegreeTwoPath-Reduction, we also need to maintain the triangle counts and the set D of dominated vertices, as follows. For the case corresponding to Figure 4(c), we only need to set $\delta(w, v_1)$ to be 0. For the case corresponding to Figure 4(d), since the degrees of w and v decrease by 1 while the triangle counts of all edges remain the same, w and v may dominate their neighbors after the update. For the case corresponding to Figure 4(e), for each common neighbor u of v and w, we increase $\delta(u, v)$ and $\delta(u, w)$ by 1 while the degrees of all vertices remain the same; thus, u may dominate v or w, and v or w may dominate w, after the update. We omit the details and the pseudocode.

Analysis and Implementation Details. The time complexity of Algorithm 5 is $O(m \times \Delta)$, where Δ is the maximum vertex degree in G. The most time-consuming parts of Algorithm 5 are related to the computation and maintenance of triangle counts, and the maintenance of the set D of dominated vertices; note that, the remaining part is similar to Algorithm 4 and has a time complexity of $O(m)$. Firstly, Line 3 takes time $O\left(\sum_{(u,v)\in E}(d(u) + d(v))\right) = O\left(\sum_{(u,v)\in E} \Delta\right) = O(m \times \Delta)$, where the triangle count of edge (u, v) is computed by intersecting $N(u)$ and $N(v)$. Secondly, updating the traingle counts and the set D of dominated vertices for all vertex removals also takes $O(m \times \Delta)$ time in total. Thirdly, rechecking the dominance at Line 8 takes $O\left(\sum_{u\in V} d(u)^2\right) = O(m \times \Delta)$ total time, since each vertex u is checked at most $d(u)$ times each of which takes $O(d(u))$ time. Fourthly, for case-4 and case-5 in DegreeTwoPath-Reduction (corresponding to Figure 4(d) and 4(e)), each updating of $\delta(u, v)$ and D takes $O(d(w) + d(v))$ time. Note that, this happens at most $n/2$ times since the number of vertices reduces by at least two each time; thus, the total time is $O(n \times \Delta)$.

As the time complexity of Algorithm 5 depends on Δ, in our implementation of NearLinear, we first run an efficient one-pass dominance reduction to reduce Δ. The intuition is that a high-degree vertex is most likely to be dominated by some low-degree vertices; thus, these dominated vertices can be removed from the graph, which reduces Δ without affecting the optimal solution size. The one-pass dominance reduction works as follows. We first sort the vertices in degree-decreasing order in linear time by count sort [18], and then apply the dominance reduction for vertices according to this order. When checking whether u is dominated by one of its neighbors, (1) we only need to consider a neighbor v if $d(v) \leq d(u)$, and (2) we can conclude that u is not dominated by v once we encounter a neighbor w of v that is not connected to u. In this way, the time complexity of the one-pass dominance reduction is $O\left(\sum_{(u,v)\in E} \min\{d(u), d(v)\}\right) = O(m \times a(G))$ [16], where $a(G)$ is the arboricity of G—the minimum number of forests needed to cover all edges of G—with $a(G) \leq \sqrt{m}$ and $a(G) \leq \Delta$. Moreover, to further reduce the graph size, we also run once the linear-programming based reduction studied in [1], after running the one-pass dominance reduction; this has a time complexity of $O(m \times \sqrt{n})$. In practice, both the one-pass dominance reduction and the linear-programming based reduction run in linear time for real graphs.

The worst-case space complexity of NearLinear is $4m + O(n)$, since we also need to store a triangle count for each edge. However, in practice it can be achieved in $2m + O(n)$ space, the same as that of BDOne. This is because the number of undirected edges, after running the one-pass dominance reduction and the linear-programming based reduction, is usually smaller than $\frac{m}{2}$; thus, all remaining edges and their triangle counts can be stored in $2m$ space.

6. EXTENSIONS

In this section, we extend our techniques in two ways: accelerating ARW, and computing an upper bound of $\alpha(G)$.

Accelerating the Iterated Local Search Algorithm ARW**.** Our techniques can be extended to accelerate the iterated local search algorithm ARW, where details of ARW are given in Section A.5 in Appendix. Let $\mathcal{K}$ be the kernel graph obtained immediately before applying the inexact reduction rule for the first time, and let $I(\mathcal{K})$ be the independent set of I induced on $\mathcal{K}$. Then, we run ARW on $\mathcal{K}$ based on the initial independent set $I(\mathcal{K})$ to iteratively improve the size of the independent set, and finally extend the best independent set obtained for $\mathcal{K}$ to an independent set of the original input graph. We denote the extensions based on LinearTime and NearLinear as ARW-LT and ARW-NL, respectively.

ARW-LT and ARW-NL differ from OnlineMIS [19] in the following aspects. (1) The kernel graph of ARW-NL is smaller than that of OnlineMIS. OnlineMIS applies only the degree-one reduction and degree-two isolation in view of the inefficiency of applying other reduction rules [19], while ARW-NL applies more reduction rules based on our newly designed reduction rules and our techniques for efficiently and incrementally applying the reduction rules. (2) The initial independent set of ARW-NL is much larger than that of OnlineMIS (see Section 7). OnlineMIS computes the initial solution by first performing a quick *single pass* of the degree-one reduction and degree-two isolation, and then invoking DU on the remaining graph, while ARW-NL computes the initial solution by our new algorithm NearLinear.

Computing Upper Bound. Each of our algorithms following the Reducing-Peeling framework directly computes a nontrivial upper bound of the independence number of a graph (*i.e.*, upper bound of $\alpha(G)$). Let I be the computed independent set, F be the set of vertices that are temporarily removed by the inexact reduction rule, and R be the subset of vertices in F that are not in I (*i.e.*, $R = F \backslash I$). We prove in Theorem 6.1 that, $|I| + |R|$ is an upper bound of $\alpha(G)$.

Theorem 6.1: $\alpha(G) \leq |I| + |R|$.

From Theorem 6.1, it is obvious that if $R = \emptyset$, then I is a maximum independent set of G, since $\alpha(G) \geq |I|$. As $R \subseteq F$, this is stronger than the statement claimed in Section 3.1, which says that if the inexact reduction rule is not applied (*i.e.*, $F = \emptyset$), then I is a maximum independent set of G.

7. EXPERIMENTS

We conduct extensive empirical studies to evaluate the efficiency and effectiveness of our new framework and algorithms for computing a high-quality independent set of a graph. Firstly, we evaluate the following algorithms that have polynomial time complexities.

- Greedy: the existing greedy algorithm (see Section 1).
- DU: the existing dynamic updating algorithm (see Section 1).
- SemiE: the semi-external algorithm in [30] with two-k swap; we store the entire graph in main memory to avoid I/Os.
- BDOne: our efficient baseline algorithm (see Section 3.2).
- BDTwo: our effective baseline algorithm (see Section 3.3).
- LinearTime: our effective linear-time algorithm (see Section 4).
- NearLinear: our near-linear-time algorithm (see Section 5).

Secondly, we evaluate the following local search and evolutionary algorithms.

- ARW: the local search algorithm in [2], initialized by DU.
- OnlineMIS: the improved local search algorithm in [19].
- ReduMIS: the evolutionary algorithm in [28].
- ARW-LT: the ARW algorithm boosted by LinearTime (see Section 6).
- ARW-NL: the ARW algorithm boosted by NearLinear (see Section 6).

All algorithms are implemented in C++ and compiled by GNU GCC 4.8.2 with the -O3 optimization; the source code of ReduMIS is obtained from the authors of [28] while all other algorithms are implemented by us. All experiments are conducted on a machine with an Intel(R) Xeon(R) 3.4GHz CPU and 16GB main memory running Linux (64bit Debian). We evaluate the performance of the algorithms on real graphs as follows.

Graph	#Vertices	#Edges	$\overline{d}$
GrQc	5,242	14,484	5.53
CondMat	23,133	93,439	8.08
AstroPh	18,772	198,050	21.10
Email	265,214	364,481	2.75
Epinions	75,879	405,740	10.69
cnr-2000	325,557	2,738,969	16.83
dblp	933,258	3,353,618	7.19
wiki-Talk	2,394,385	4,659,565	3.89
BerkStan	685,230	6,649,470	19.41
as-Skitter	1,696,415	11,095,398	13.08
in-2004	1,382,870	13,591,473	19.66
eu-2005	862,664	16,138,468	37.42
soc-pokec	1,632,803	22,301,964	27.32
LiveJ	4,847,571	42,851,237	17.68
hollywood	1,985,306	114,492,816	115.34
indochina	7,414,768	150,984,819	40.73
uk-2002	18,484,117	261,787,258	28.33
uk-2005	39,454,746	783,027,125	39.70
webbase	115,657,290	854,809,761	14.78
it-2004	41,290,682	1,027,474,947	49.77

Table 2: Statistics of real graphs ($\overline{d}$: average degree)

Real Graphs. We evaluate the algorithms on twenty real graphs from different domains. The graphs are downloaded from the Stanford Network Analysis Platform[1] and the Laboratory of Web Algorithmics[2]; descriptions of these graphs can also be found there. Statistics of these graphs are shown in Table 2, where the last column gives the average degree $\overline{d}$. The graphs in Table 2 are sorted in increasing order regarding the number of edges.

Evaluation Metrics. We evaluate the different algorithms from three aspects: independent set size, processing time, and memory usage. Firstly, the larger the independent set outputted by an algorithm, the better the algorithm. Secondly, for the processing time, the smaller the better; we run an algorithm three times and report the average CPU time. Thirdly, the smaller memory consumed by an algorithm the better; we measure the heap memory usage by the Linux command memusage[3].

7.1 Experimental Results

In order to see how far is the computed independent set to a maximum independent set for an input graph, we also obtain the source code of VCSolver from the authors of [1] which computes a maximum independent set but with exponential worst-case time complexity. We categorize the twenty graphs in Table 2 into easy instances and hard instances, where the easy instances are the graphs that VCSolver can finish in five hours and are shown in Table 3. Note that, additional experimental results, *e.g.*, on synthetic graphs, are presented in Section A.4 in Appendix.

Eval-I: Evaluating Our Baseline Algorithm BDOne Against the Existing Polynomial-Time Algorithms Greedy, DU and SemiE. We first evaluate the effectiveness of our Reducing-Peeling framework against the existing techniques; thus, we consider our baseline algorithm BDOne. The gaps/errors of the reported independent set sizes by the four algorithms to the independence number computed by VCSolver [1] for the twelve easy real graphs are shown in the third-to-sixth columns of Table 3. DU has a smaller gap than Greedy thanks to the adaptive selection of the minimum-degree vertex in DU, and SemiE improves upon Greedy as a result of one-k swaps and two-k swaps in SemiE. Note that, SemiE first computes an initial independent set by Greedy, and then itera-

[1]http://snap.stanford.edu/

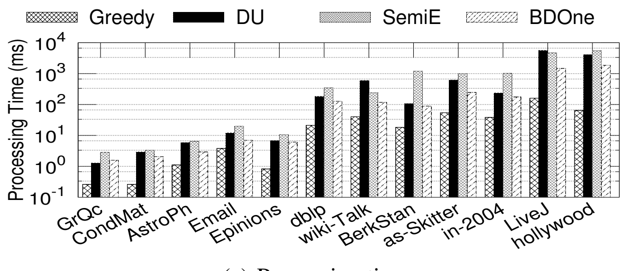
[2]http://law.di.unimi.it/datasets.php

[3]http://man7.org/linux/man-pages/man1/memusage.1.html

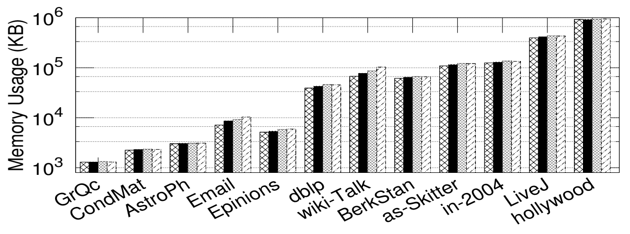
Graphs	Independence Number	Gap to the Independence Number							Accuracy of NearLinear	Kernel Graph Size by NearLinear
		Greedy	DU	SemiE	BDOne	BDTwo	LinearTime	NearLinear		
GrQc	2,459	5	1	1	0	0	0	**0***	100%	0
CondMat	9,612	17	5	1	4	2	1	**0***	100%	0
AstroPh	6,760	24	10	1	2	0	1	**0***	100%	0
Email	246,898	76	0	1	0	**0***	0	**0***	100%	0
Epinions	53,599	170	3	14	0	0	0	0	100%	6
dblp	434,289	484	63	53	45	5	4	**0***	100%	0
wiki-Talk	2,338,222	536	0	14	0	0	0	**0***	100%	0
BerkStan	408,482	11,092	3,000	4,458	1,088	385	766	428	99.895%	55,990
as-Skitter	1,170,580	34,591	2,336	5,886	319	55	170	39	99.997%	9,733
in-2004	896,724	14,832	3,553	5,918	656	351	412	57	99.993%	19,575
LiveJ	2,631,903	32,997	6,138	7,364	1,494	343	378	33	99.998%	10,173
hollywood	327,949	98	45	8	16	4	4	**0***	100%	0

Table 3: The gap of the reported independent set size to the independence number computed by VCSolver [1] (* denotes that the independent set is reported as a maximum independent set by our algorithms)

tively improves the independent set size by one-*k* swaps and two-*k* swaps [30]. BDOne consistently has a smaller gap than Greedy and DU across all the twelve graphs, and has a smaller gap than SemiE on most of the graphs. This demonstrates the superiority of our Reducing-Peeling framework to the existing paradigms.



(a) Processing time

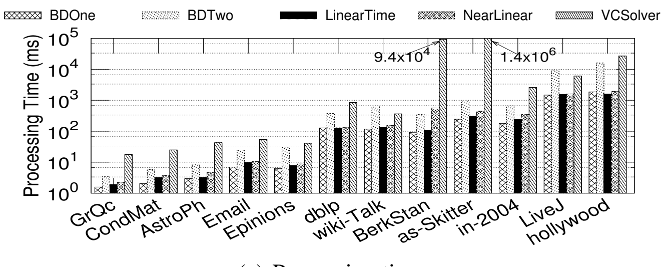


(b) Memory usage

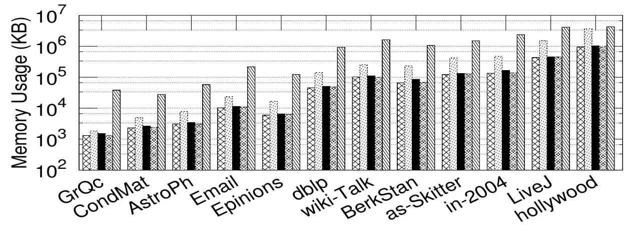
Figure 7: Time and space of Greedy, DU, SemiE and BDOne

The processing time and memory usage of the algorithms on the twelve easy real graphs are shown in Figure 7. Generally, the processing time and memory usage of all four algorithms increase along with the increasing of the graph size. Regarding the processing time in Figure 7(a), Greedy runs the fastest due to its simplicity, BDOne runs faster than DU as a result of our lazy updating strategy of the data structure (see Section 3.2), and SemiE runs the slowest due to the time-consuming two-*k* swaps. Regarding the memory usage in Figure 7(b), all the four algorithms consume similar memory spaces. Thus, BDOne computes much larger independent sets without sacrificing processing time or memory usage.

Eval-II: Evaluating Our Algorithms, BDOne, BDTwo, LinearTime, and NearLinear. The gaps of the reported independent set sizes by our algorithms to the independence numbers on the twelve real graphs are shown in Table 3 (from sixth to ninth columns). We can see that BDOne has the largest gap since it only applies the degree-one reduction. Both BDTwo and LinearTime have smaller gaps than BDOne because they additionally apply reduction rules for handling degree-two vertices. BDTwo has a relatively smaller gap than LinearTime due to that BDTwo can handle all degree-two vertices while LinearTime can handle all but one cases (see Section A.2 in Appendix); nevertheless, the difference is not significant. Generally, NearLinear has the smallest gap due to the additional reduction rules applied, and has an accuracy of ≥ 99.895% for all the twelve graphs as shown in the tenth column. Moreover, among the twelve graphs, NearLinear reports the maximum independent set for seven graphs (as indicated by $*$) since the kernel graph is empty as shown in the last column of Table 3. Note that, although the gap of the independent set reported by other algorithms can also be 0, they cannot report it as maximum independent set since they cannot ensure it to be a maximum independent set without knowing the independence number of the graph; for example, DU on *Email*, BDOne on *GrQc*, and LinearTime on *Epinions*.



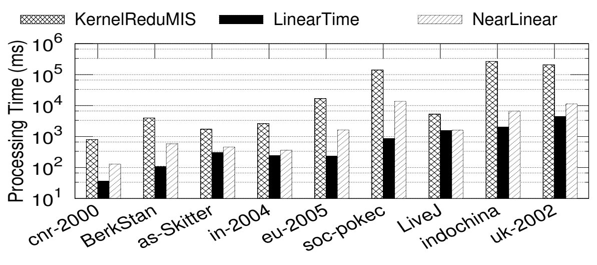
(a) Processing time



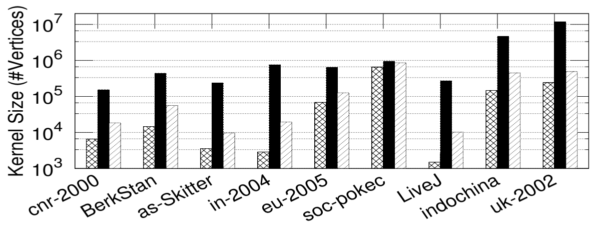
(b) Memory usage

Figure 8: Time and space of our algorithms

The processing time and memory consumption of these algorithms are shown in Figure 8. Generally, BDOne, LinearTime and NearLinear have similar running time, where both BDOne and LinearTime have time complexity $O(m)$. Although NearLinear has time complexity $O(m \times \Delta)$, it takes linear time for most of the graphs (except *BerkStan*), due to the small Δ after our optimizations in Section 5. Moreover, the memory consumptions of these three algorithms are also similar as shown in Figure 8(b). On the other hand, BDTwo runs slower and also consumes more memory space, due to its higher complexities of time and space. Specifically, BDTwo consumes almost 3 times more memory space than BDOne, LinearTime, and NearLinear. As a reference, the exact exponential-time algorithm VCSolver takes significantly more time and space than other algorithms, as shown in Figure 8. Note that, the time complexity of VCSolver highly depends on the kernel graph size. VCSolver obtains an empty kernel graph for each of the twelve easy graphs, except *Berkstan* and *as-Skitter*, and is thus only slower than our algorithms by one order of magnitude on these graphs. In summary, LinearTime is the best algorithm if linear time complexity is required, and NearLinear is a better choice if higher solution quality is preferred but still with near-linear time.



(a) Processing time



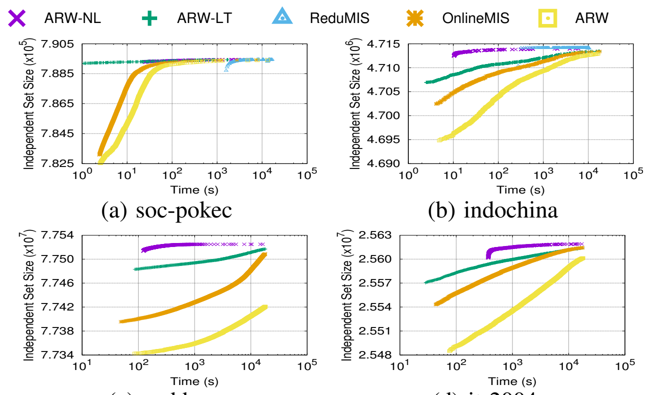
(b) Kernel size

Figure 9: Processing time and kernel size

Eval-III: Evaluating Kernelization Techniques. We also evaluate the kernelization techniques in ReduMIS, LinearTime, and NearLinear; the results are shown in Figure 9. Here, KernelReduMIS is modified from ReduMIS by terminating the algorithm immediately after obtaining the kernel graph; note that, the set of reduction rules used in KernelReduMIS is exactly the same as in VCSolver [1]. In Figure 9, we only show the results for the graphs that KernelReduMIS obtains a non-empty kernel graph; note that, NearLinear also obtains empty kernel graphs for the other graphs not in Figure 9, except *Epinions*, as illustrated in Table 3. The three largest graphs, *uk-2005*, *webbase*, and *it-2004*, are also omitted in Figure 9 since KernelReduMIS runs out of memory.

From Figure 9, we can see that KernelReduMIS runs much slower than LinearTime and NearLinear; note that, KernelReduMIS only obtains the kernel graph while LinearTime and NearLinear further compute independent sets. Thus, applying the full set of reduction rules in [1] takes an excessive long time such that it is not suitable for processing large graphs (see our discussions in Section 3.1), which motivates us to design new reduction rules and develop efficient techniques to incrementally apply reduction rules. Regarding the kernel graph size, KernelReduMIS computes the smallest kernel graph as a result of applying the full set of reduction rules in [1], while LinearTime outputs the largest kernel graph with the fastest processing time. To strike a balance between the processing time and kernel graph size, NearLinear computes a smaller kernel graph than LinearTime while still guaranteeing a near-linear time complexity which is required for efficiently processing large graphs. NearLinear is able to output maximum or near-maximum independent sets for most of the graphs as shown in Table 3.

Eval-IV: Boosting ARW by NearLinear. We also evaluate the speedup of the local search algorithm ARW boosted by our algorithms LinearTime and NearLinear. We run the algorithms on the eight hard real graphs that VCSolver cannot finish in five hours, and



(a) soc-pokec

(b) indochina

(c) webbase

(d) it-2004

Figure 10: Convergence plots for local search algorithms

set the time limit for each algorithm to be five hours (*i.e.*, 1.8×10^4 seconds). An algorithm reports a tuple $(t, |I|)$ if it finds a new larger independent set I at time t; the first independent sets reported by ARW, ARW-LT, and ARW-NL are the ones obtained after running one iteration of the local search on the initial independent sets computed by DU, LinearTime, and NearLinear, respectively.

The results on graphs *soc-pokec*, *indochina*, *webbase* and *it-2004*, are shown in Figure 10 while the results on the other four graphs are shown in Figure 15 in Appendix; here, x-axis shows the time t and y-axis shows the independent set size $|I|$. ReduMIS cannot run on *uk-2005*, *webbase*, and *it-2004* due to running out of memory. The accuracy of the first independent set outputted by ARW-NL compared with the largest one among all independent sets outputted by these algorithms during a five-hour run on these four graphs are 99.979%, 99.963%, 99.985%, and 99.931%, respectively. Across all graphs, ARW-LT and ARW-NL significantly speed up ARW; the initial independent set outputted by ARW-NL could also be larger than the largest independent set outputted by ARW for a five hours' running (*e.g.*, on *webbase*). ARW-LT and ARW-NL also take an early lead over ReduMIS that applies the full set of reduction rules in [1]; that is, ARW-NL takes much shorter time than ReduMIS to output independent sets of the same sizes. Moreover, both ARW-LT and ARW-NL outperform OnlineMIS, the most recent work aiming to speedup ARW. ARW-LT and ARW-NL also scale well to large graphs, while ReduMIS fails on the three largest graphs due to running out of memory.

8. CONCLUSION

In this paper, we developed a new Reducing-Peeling framework for efficiently computing a near-maximum independent set of a large graph. Following the framework, we designed two baseline algorithms based on the existing reduction rules for handling degree-one and degree-two vertices. To further improve the efficiency and the solution quality, we proposed a linear-time algorithm and a near-linear time algorithm by designing new reduction rules and developing techniques for efficiently and incrementally applying reduction rules. Moreover, we also extended our techniques to accelerate the local search algorithm ARW. Extensive empirical studies demonstrate that all our algorithms output much larger independent sets than the existing linear-time algorithms while having a similar running time, and by combining our techniques with ARW, we achieve significant speedup against ARW. As a possible future direction, developing new exact reduction rules and new techniques for efficiently applying these reduction rules might be an interesting issue to be investigated. Extending our techniques to compute independent sets I/O efficiently might also be an interesting issue to be investigated.

Acknowledgements

Lijun Chang is supported by ARC DE150100563 and DP160101513. Wenjie Zhang is supported by ARC DP150103071 and DP150102728.

9. REFERENCES

[1] T. Akiba and Y. Iwata. Branch-and-reduce exponential/fpt algorithms in practice: A case study of vertex cover. *Theor. Comput. Sci.*, 609, 2016.

[2] D. V. Andrade, M. G. Resende, and R. F. Werneck. Fast local search for the maximum independent set problem. *J. Heuristics*, 18(4), 2012.

[3] 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 and Distributed Computing*, 71(10), 2011.

[4] A.-L. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286, 1999.

[5] S. Basagni. Finding a maximal weighted independent set in wireless networks. *Telecommunication Systems*, 18(1-3), 2001.

[6] U. Benlic and J.-K. Hao. Breakout local search for maximum clique problems. *Computers & Operations Research*, 40(1), 2013.

[7] P. Berman and T. Fujito. On approximation properties of the independent set problem for low degree graphs. *Theor. Comput. Sys.*, 32(2), 1999.

[8] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient subgraph matching by postponing cartesian products. In *Proc. of SIGMOD'16*, 2016.

[9] S. Butenko. *Maximum independent set and related problems with applications*. PhD thesis, University of Florida, 2003.

[10] J. M. Byskov. Enumerating maximal independent sets with applications to graph colouring. *Oper. Res. Lett.*, 32(6), 2004.

[11] S. Cai. Balance between complexity and quality: local search for minimum vertex cover in massive graphs. In *Proc. of IJCAI'15*, 2015.

[12] L. Chang, J. X. Yu, and L. Qin. Fast maximal cliques enumeration in sparse graphs. *Algorithmica*, 2012.

[13] L. Chang, J. X. Yu, L. Qin, X. Lin, C. Liu, and W. Liang. Efficiently computing k-edge connected components via graph decomposition. In *Proc. of SIGMOD'13*, 2013.

[14] J. Cheng, Y. Ke, S. Chu, and C. Cheng. Efficient processing of distance queries in large graphs: a vertex cover approach. In *Proc. of SIGMOD'12*, 2012.

[15] J. Cheng, Y. Ke, A. W.-C. Fu, J. X. Yu, and L. Zhu. Finding maximal cliques in massive networks by h*-graph. In *Proc. of SIGMOD'10*, 2010.

[16] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14(1), 1985.

[17] R. H. Chitnis, G. Cormode, M. T. Hajiaghayi, and M. Monemizadeh. Parameterized streaming: Maximal matching and vertex cover. In *Proc. of SODA'15*, 2015.

[18] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[19] J. Dahlum, S. Lamm, P. Sanders, C. Schulz, D. Strash, and R. F. Werneck. Accelerating local search for the maximum independent set problem. In *Proc. of SEA'16*, 2016.

[20] U. Feige. Approximating maximum clique by removing subgraphs. *SIAM J. Discrete Math.*, 18(2), 2004.

[21] F. V. Fomin, F. Grandoni, and D. Kratsch. A measure & conquer approach for the analysis of exact algorithms. *J. ACM*, 56(5), 2009.

[22] A. W.-C. Fu, H. Wu, J. Cheng, and R. C.-W. Wong. Is-label: an independent-set based labeling scheme for point-to-point distance querying. *PVLDB*, 6(6), 2013.

[23] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[24] A. Gibbons. *Algorithmic Graph Theory*. Cambridge University Press, 1985.

[25] M. M. Halldórsson and J. Radhakrishnan. Greed is good: Approximating independent sets in sparse and bounded-degree graphs. *Algorithmica*, 18(1), 1997.

[26] J. Håstad. Clique is hard to approximate within n$^{1\text{-epsilon}}$. In *Proc. of FOCS'96*, 1996.

[27] S. Lamm, P. Sanders, and C. Schulz. Graph partitioning for independent sets. In *Proc. of SEA'15*, 2015.

[28] S. Lamm, P. Sanders, C. Schulz, D. Strash, and R. F. Werneck. Finding near-optimal independent sets at scale. In *Proc. of ALENEX'16*, 2016.

[29] Y. Lim, U. Kang, and C. Faloutsos. Slashburn: Graph compression and mining beyond caveman communities. *IEEE Trans. Knowl. Data Eng.*, 26(12), 2014.

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

[31] P. M. Pardalos and J. Xue. The maximum clique problem. *J. global Optimization*, 4(3), 1994.

[32] D. Puthal, S. Nepal, C. Paris, R. Ranjan, and J. Chen. Efficient algorithms for social network coverage and reach. In *2015 IEEE International Congress on Big Data*, 2015.

[33] P. S. Segundo, A. Lopez, and P. M. Pardalos. A new exact maximum clique algorithm for large and massive sparse graphs. *Computers & Operations Research*, 66, 2016.

[34] S. S. Skiena. *The Algorithm Design Manual*. Springer Publishing Company, Incorporated, 2nd edition, 2008.

[35] E. Tomita, Y. Sutani, T. Higashi, S. Takahashi, and M. Wakatsuki. A simple and faster branch-and-bound algorithm for finding a maximum clique. In *Proc. of WALCOM'10*, 2010.

[36] P.-J. Wan, B. Xu, L. Wang, S. Ji, and O. Frieder. A new paradigm for multiflow in wireless networks: Theory and applications. In *Proc. of INFOCOM'15*, 2015.

[37] J. Xiang, C. Guo, and A. Aboulnaga. Scalable maximum clique computation using mapreduce. In *Proc. of ICDE'13*, 2013.

[38] M. Xiao and H. Nagamochi. Confining sets and avoiding bottleneck cases: A simple maximum independent set algorithm in degree-3 graphs. *Theor. Comput. Sci.*, 469, 2013.

[39] Z. Zhang, L. Qin, and J. X. Yu. Contract & expand: I/O efficient sccs computing. In *Proc. of ICDE'14*, 2014.

[40] Y. Zhu, L. Qin, J. X. Yu, Y. Ke, and X. Lin. High efficiency and quality: large graphs matching. *VLDB J.*, 22(3), 2013.

A. APPENDIX

A.1 Proofs of Lemmas and Theorems

Proof of Theorem 3.1. We prove the theorem by constructing a graph instance such that Algorithm 3 runs in $O(n \log n)$ time and there are only $O(n)$ edges. Consider the graph in Figure 11, which consists of vertices in four layers. The first layer is $\{v_{3n+1}, v_{3n+2}\}$, the second layer is $\{v_{n+1}, \cdots, v_{3n}\}$, the third layer is $\{v_1, \cdots, v_n\}$, and the fourth layer consists of other vertices. Vertices in the first layer and the second layer form a complete bipartite graph. Each vertex v_i in the third layer is connected to two vertices $v_{n+2*i-1}$ and v_{n+2*i} in the second layer, and vertices in the fourth layer are connected to vertices in the third layer to trigger the degree-two folding.

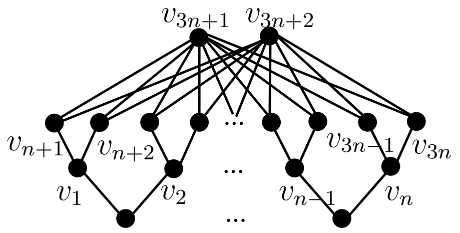


Figure 11: Graph in proving the time complexity of Algorithm 3

We consider the degree-two folding (*i.e.*, contraction) in iterations. When contracting two vertices, we assume the resulting vertex is the vertex with a larger id, and the cost is the smaller degree of these two vertices; for example, for contracting v_1 and v_2, the resulting vertex is v_2, and the cost is 2. The first iteration of the degree-two folding contracts $\{v_1, v_2\}, \cdots, \{v_{n-1}, v_n\}$ by the vertices at the fourth layer as shown in Figure 11, with cost $\frac{n}{2} \times 2 = n$. To trigger the degree-two folding for the second iteration, we assume there are other vertices at the fourth layer, each of degree three; for example, the additional vertex connected to v_1, v_2, v_4 triggers the contraction of v_2 and v_4 in the second iteration. In the second iteration, we have $n/4$ contraction operations each of cost 4; thus, the total cost is $\frac{n}{4} \times 4 = n$. So on so forth, to contract all vertices at the third layer into a single vertex, the degree-two folding occurs for $\log n$ iterations, with total cost $n \log n$.

Now, let's analyze the number of vertices and edges of the graph in Figure 11. The first layer, second layer, and third layer contains 2, $2n$, and n vertices respectively. For the fourth layer, to trigger the contraction operations on all vertices at the third layer at iteration i, we need $\frac{n}{2^i}$ additional vertices; for example, the first iteration needs $\frac{n}{2}$ vertices as shown at the fourth layer in Figure 11. Moreover, each of such vertices except those shown in Figure 11 has degree 3. Thus, the number of vertices at the fourth layer is $\frac{n}{2} + \frac{n}{2^2} + \cdots + \frac{n}{2^{\log n}} = n - 1$; here, we assume n is a power of 2. Therefore, the total number of vertices is $4n + 1$, and the total number of edges is $2n \times 2 + n \times 2 + \frac{n}{2} \times 2 + (\frac{n}{2} - 1) \times 3 = \frac{17}{2}n - 3$. That is, there exists a graph with $4n + 1$ vertices and $\frac{17}{2}n - 3$ edges such that Algorithm 3 runs in $O(n \log n)$ time. Thus, the theorem holds. □

Proof of Lemma 4.1. Firstly, if G contains a degree-two cycle C, it is easy to see that vertices in C are disconnected from the remaining part of G and $\alpha(G) = \alpha(G\backslash C) + \alpha(C)$. Moreover, $\alpha(C) = \lfloor \frac{|C|}{2} \rfloor$, and the maximum independent set of C can be computed by removing an arbitrary vertex $v \in C$ from C and then iteratively applying the degree-one reduction.

Secondly, if G contains a maximal degree-two path P with $v = w$ (*i.e.*, case-1), then there must exist a maximum independent set of G that excludes v. Consider a maximum independent set I that includes v, it must satisfy $|I \cap P| = \lceil \frac{|P|-2}{2} \rceil$ since $v \in I$. If we exclude v from I, we can enlarge $I\backslash(\{v\} \cup P)$ by adding $\lceil \frac{|P|}{2} \rceil$ vertices of P to obtain an independent set of the same size as I, which is a maximum independent set and does not contain v.

Thirdly, if G contains a degree-two path P such that $|P|$ is odd and $(v, w) \in E$ (*i.e.*, case-2), then there must exist a maximum independent set of G that excludes v and w. Consider a maximum independent set I of G that includes either v or w but not both (without loss of generality, assume $v \in I$), it must satisfy $|I \cap P| = \frac{|P|-1}{2}$ since v_1 cannot be in I. Therefore, $(I\backslash(\{v\} \cup P)) \cup \{v_1, v_3, \dots, v_{l-2}, v_l\}$ is an independent set of the same size as I, which is a maximum independent set and contains neither v nor w.

Fourthly, if G contains a degree-two path P such that $|P|$ is odd and $(v, w) \notin E$ (*i.e.*, case-3), then there must exist a maximum independent set of G that excludes either v_1 or w. Here, we assume $|P| \geq 3$, since it is trivial for $|P| = 1$. Consider a maximum independent set I of G that includes both v_1 and w, it must satisfy $|I \cap \{v_2, \dots, v_l\}| = \frac{|P|-3}{2}$ since $v_1 \in I$ and $w \in I$. Therefore, both $(I\backslash(\{v_1\} \cup P)) \cup \{v_2, v_4, \dots, v_{l-3}, v_{l-1}\}$ and $(I\backslash(\{w\} \cup P)) \cup \{v_3, v_5, \dots, v_{l-2}, v_l\}$ are independent sets of the same size as I, which are maximum independent sets and excludes v_1 or w. Moreover, both $I\backslash(\{v_1\} \cup P)$ and $I\backslash(\{w\} \cup P)$ are maximum independent sets of the graph $G\backslash\{v_2, \dots, v_l\} \cup \{(v_1, w)\}$. Thus, we can remove $\{v_2, \dots, v_l\}$ from the graph and add edge (v_1, w) to the graph.

Similarly, we can prove case-4 and case-5 for $|P|$ being even. □

Proof of Lemma 5.2. According to the definition of dominance, u dominates its neighbor v if and only if every neighbor (except v) of u is also connected to v. Equivalently, every vertex in $N(u)\backslash\{v\}$ forms a triangle with u and v (*i.e.*, $\delta(u, v) \geq |N(u)\backslash\{v\}| = d(u) - 1$). Moreover, we have $\delta(u, v) \leq d(u) - 1$. Thus, $\delta(u, v) = d(u) - 1$. □

Proof of Theorem 6.1. In order to prove Theorem 6.1, we first prove that $\alpha(G\backslash S) \leq \alpha(G)$ for any subset S of vertices of V; that is, removing any vertex from a graph G will not result in a larger maximum independent set. This claim can be proved by contradiction. Assume there is a maximum independent set I' of $G\backslash S$ that is larger than a maximum independent set I of G (*i.e.*, $|I'| > |I|$). By following the facts that $I' \subset V$ and for any two vertices u and v in I', (u, v) is in G if and only if it is in $G\backslash S$, it is easy to see that I' is also an independent set of G and is larger than I. This contradicts that I is a maximum independent set of G. Thus, the claim of $\alpha(G\backslash S) \leq \alpha(G)$ holds.

Now, we prove the theorem by induction. Firstly, let's consider the case $R = \emptyset$. We prove that I is a maximum independent set of G by contradiction. Assume I' with $|I'| > |I|$ is a maximum independent set of G. Then, $I'\backslash F$ must be an independent set of $G\backslash F$. As $R = \emptyset$, we have $F \subseteq I$, and thus $|I'\backslash F| > |I\backslash F|$. However, from our Reducing-Peeling framework in Algorithm 1 we know that $I\backslash F$ is a maximum independent set of $G\backslash F$, since $I\backslash F$ is obtained by solely applying the exact reduction rules on $G\backslash F$. Contradiction. Thus, the theorem holds for $R = \emptyset$.

Secondly, assume the theorem holds for $|R| = k$, we prove that the theorem also holds for $|R| = k+1$. Consider the first vertex v in R that is removed from the graph. (1) If there is a maximum independent set of G not including v, then $\alpha(G) = \alpha(G\backslash\{v\})$. By replacing the input graph G with $G\backslash\{v\}$, we still will get the independent set I, and let the subset of vertices that are removed by the inexact reduction rule and are not in I be R'. Then $R' = R\backslash\{v\}$, and $\alpha(G\backslash\{v\}) \leq |I| + |R'|$, which implies that $\alpha(G) \leq |I| + |R|$. (2) If every maximum independent set of G includes v, then $\alpha(G) = \alpha(G\backslash N[v]) + 1$, where $N[v] = N(v) \cup \{v\}$ is the closed neighborhood of v. Moreover, we have $\alpha(G\backslash\{v\}) \leq |I| + |R'|$ by the assumption, where $R' = R\backslash\{v\}$ is the subset of vertices that are removed by the inexact reduction rule and are not in I. Thus, $\alpha(G) = \alpha(G\backslash N[v]) + 1 \leq \alpha(G\backslash\{v\}) + 1 \leq |I| + |R'| + 1 = |I| + |R|$, and the theorem holds. □

A.2 Degree-two Path Reduction vs. Degree-two Vertex Reduction

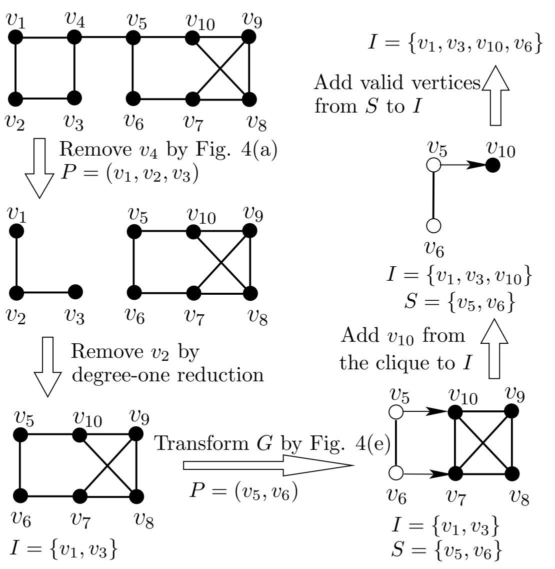
Our newly proposed degree-two path reductions can handle all degree-two vertices except the case that both neighbors of a degree-two vertex have degrees at least three. That is, if the two neighbors v and w of a degree-two vertex u have $d(v) \geq 3$ and $d(w) \geq 3$, then the degree-two path reduction cannot be applied on u. But, we still remove u from $V_{=2}$ in Algorithm 4 (Line 20); if u later participates in a degree-two path P with $|P| \geq 2$, P still can be found since at least one vertex of P is in $V_{=2}$. On the other hand, the degree-two vertex reductions can handle all degree-two vertices; for example, for the above case, the degree-two folding will contract $\{u, v, w\}$ into a single vertex. Nevertheless, we do not apply the degree-two folding since it does not guarantee worst-case linear time complexity and moreover it consumes more memory space (see Section 3.3).

A.3 Properties of The Dominance Reduction

Capturing Other Reduction Rules. The dominance reduction captures some other reduction rules as follows. Firstly, the *isolated vertex reduction* [19] is applied to a vertex u whose neighbors are connected to each other and form a clique with u as shown in Figure 13(a); if a vertex u satisfies the isolated vertex reduction, then we can remove all its neighbors from the graph while preserving the maximality of independent sets (*i.e.*, $\alpha(G) = \alpha(G\backslash N(u))$) [19].

Graphs	Best Result Size	Gap to the Best Result Size						
		Greedy	DU	SemiE	BDOne	BDTwo	LinearTime	NearLinear
cnr-2000	230,036	4,007	1,432	1,652	407	153	293	228
eu-2005	452,352	11,399	5,043	3,640	1,456	1,022	1,106	294
soc-pokec	789,447	63,389	13,494	21,781	836	221	444	286
indochina	4,714,147	109,577	48,676	37,083	11,920	8,156	8,986	1,871
uk-2002	11,915,614	294,714	138,035	98,468	20,239	12,036	15,418	3,650
uk-2005	23,697,232	550,726	255,947	167,238	38,027	-	28,906	10,455
webbase	77,524,841	1,292,632	435,646	444,512	81,758	-	53,079	12,808
it-2004	25,619,067	690,823	337,833	249,501	80,479	-	64,668	21,300

Table 4: The gap of the reported independent set size to the best one obtained by local search algorithms for the eight hard graphs (BDTwo cannot run on *uk-2005, webbase, it-2004* due to running out-of-memory)



v_1 v_4 v_5 v_{10} v_9
v_2 v_3 v_6 v_7 v_8
Remove v_4 by Fig. 4(a)
$P = (v_1, v_2, v_3)$
Remove v_2 by degree-one reduction
Transform G by Fig. 4(e)
$P = (v_5, v_6)$
$I = \{v_1, v_3\}$
$I = \{v_1, v_3\}$
$S = \{v_5, v_6\}$
Add v_{10} from the clique to I
$I = \{v_1, v_3, v_{10}\}$
$S = \{v_5, v_6\}$
Add valid vertices from S to I
$I = \{v_1, v_3, v_{10}, v_6\}$

Figure 12: Running example of LinearTime

It is easy to verify that, if u satisfies the isolated vertex reduction, then u dominates every vertex $v \in N(u)$; thus, $N(u)$ can also be removed by the dominance reduction. Note that, the degree-one reduction and the degree-two isolation discussed in Section 2.1 are special cases of the isolated vertex reduction; thus, the dominance reduction also captures these two reduction rules.

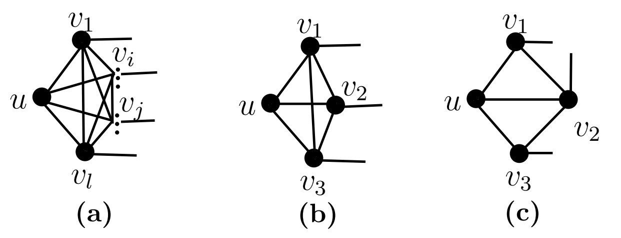


Figure 13: Isolated vertex reduction and degree-three reductions

Secondly, let u be a vertex of degree three with neighbors v_1, v_2 and v_3, the dominance reduction captures two of the four configurations of edges between $\{v_1, v_2, v_3\}$. (1) If there are three edges between $\{v_1, v_2, v_3\}$ as shown in Figure 13(b), then all v_1, v_2, v_3 are dominated by u. (2) If there are two edges between $\{v_1, v_2, v_3\}$ as shown in Figure 13(c), then v_2 is dominated by u. The other two configures are that there are one edge or no edge between these vertices.

Mutual Dominance. It is possible that there are two vertices u and v such that u dominates v and v also dominates u, as shown in Figure 14. Moreover, as shown in Figure 14, if u is removed due to being dominated by v, then v is no longer dominated by any other vertex in the resulting graph; this also holds if we remove v.

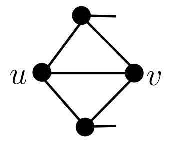


Figure 14: Mutual dominance

Order Oblivious. Another important property of the dominance reduction is that, the order of removing dominated vertices from the graph does not matter, as shown by the lemma below.

Lemma A.1: *Given vertices v, u, w such that v dominates u and u dominates w, then v must dominate w. Moreover, v still dominates w after removing u from the graph.*

Proof. Firstly, as v dominates u, we have $(v, u) \in E$; thus, $(v, w) \in E$ since u dominates w. Secondly, we prove that $N(v)\backslash\{w\} \subseteq N(w)$. As v dominates u, we have $N(v)\backslash\{u\} \subseteq N(u)$. Moreover, we have $N(u)\backslash\{w\} \subseteq N(w)$ since u dominates w. Thus, we have $N(v)\backslash\{w\} = (N(v)\backslash\{u\})\backslash\{w\} \cup \{u\} \subseteq (N(u)\backslash\{w\}) \cup \{u\} \subseteq N(w) \cup \{u\} = N(w)$, where the last equality follows since $(u, w) \in E$.

It is easy to see that, $(v, w) \in E$ and $N(v)\backslash\{w\} \subseteq N(w)$ still holds after removing u from the graph. □

Following Lemma A.1, given a vertex u that dominates other vertices and is itself dominated by v, if we remove u from the graph, then all the vertices that are dominated by u before the removal of u will be still dominated by v in the resulting graph. Thus, exactly the same number of vertices will be removed from the graph by the dominance reduction, regardless of the order of applying dominance reduction on vertices.

A.4 Additional Experimental Results

In this subsection, we present additional experimental results.

Experimental Results of Independent Set Sizes on Hard Graphs. The gaps of the independent sets reported by different algorithms to the best result size, which is obtained by the local search algorithms (*e.g.*, see Figure 10), on the eight hard graphs are shown in Table 4. The trend is similar to that in Table 3. Again, our baseline algorithm BDOne has a much smaller gap than all existing heuristic algorithms. Regarding our algorithms, NearLinear generally has the smallest gap, except on *cnr-2000* and *soc-pokec* on which BDTwo is better. Note that, the dominance reduction used in NearLinear is orthogonal to the degree-two folding used in BDTwo. Thus, BDTwo may perform better if dominance reduction does not have much effect on a graph while degree-two folding can reduce the graph significantly. Nevertheless, BDTwo has the drawback of higher time and space complexities. We will consider integrating both dominance reduction and degree-two folding into our framework in our future work.

Experimental Results on Synthetic Graphs. We also evaluate the algorithms on synthetic graphs.

Graphs	β	Independence Number	Gap to the Independence Number			
			Greedy	DU	SemiE	BDOne
PLR1	1.9	9,094,639	10,701	0	576	**0***
PLR2	2.0	8,252,480	22,101	0	1,528	**0***
PLR3	2.1	7,739,174	25,025	0	1,915	**0***
PLR4	2.2	7,457,251	24,007	0	1,877	**0***
PLR5	2.3	7,147,471	22,310	0	1,732	**0***
PLR6	2.4	6,936,962	19,991	0	1,438	**0***
PLR7	2.5	6,753,897	16,560	0	1,141	**0***
PLR8	2.6	6,608,463	13,870	0	858	**0***
PLR9	2.7	6,489,153	11,671	0	723	**0***

Table 5: The gaps to the independence number on power-law graphs (all our algorithms report maximum independent sets)

Power-Law Graphs. We generate nine Power-Law Random (PLR) graphs with 10^7 vertices by varying the growth exponent β from 1.9 to 2.7, by NetworkX[4]; this kind of graph is also tested in [30]. The results on PLR graphs are shown in Table 5, which have a similar trend as Table 3. One thing to notice is that, the power-law random graphs are actually very easy to process, which is also the main motivation of our Reducing-Peeling framework; for example, even our baseline algorithm BDOne reports maximum independent sets for all these synthetic graphs. Note that, although DU also has a zero gap, there is no mechanism for DU to certain that a reported independent set is maximum. This confirms the advantage of our Reducing-Peeling framework.

G	Best Size	Gap to the Best Result Size				
		DU	SemiE	BDOne	BDTwo	NearLinear
R1	472,545	0	4,072	**0***	**0***	**0***
R2	480,982	0	5,246	**0***	**0***	**0***
R3	484,889	0	6,579	**0***	**0***	**0***
R4	485,487	29	8,004	6	**0***	2
R5	*483,510*	1,441	9,218	286	4	254

Table 6: The gap of the reported independent set size to the best one for random graphs

Random Graphs. We also generate random graphs by the graph generator GTGraph[5], where each edge is generated by randomly choosing a pair of vertices. Specifically, we have 5 random graphs $R1$, $R2$, $R3$, $R4$, $R5$ with average degrees 2, 2.25, 2.5, 2.75, 3, respectively; each graph has 10^6 vertices. The results are shown in Table 6, which has a similar trend as Table 4 and Table 5. Note that, our algorithms report the optimal solution for $R1$, $R2$, $R3$ and $R4$, while none of the algorithms can find an optimal solution for $R5$; VCSolver runs stack overflow on $R5$.

Graphs	Existing	Our	Graphs	Existing	Our
GrQc	2,462	2,459	wiki-Talk	2,338,225	2,338,222
CondMat	9,645	9,612	BerkStan	414,440	415,032
AstroPh	6,790	6,760	as-Skitter	1,173,313	1,172,422
Email	246,898	246,898	in-2004	906,763	899,143
Epinions	53,670	53,600	LiveJ	2,674,121	2,633,626
dblp	434,870	434,289	hollywood	328,074	327,949

Table 7: Upper bounds of the independence number computed by [1] and our NearLinear

Experimental Results on our Obtained Upper Bounds. The exact algorithm for computing a maximum independent set in [1] needs an upper bound of the independence number of a graph for pruning unpromising branches. Thus, the best existing upper bound in [1], is computed as the minimum of the clique cover-based upper bound, linear programming-based upper bound, and cycle cover-based upper bound. The results for the upper bounds computed by the existing technique in [1] and our NearLinear are shown in Table 7, where the upper bound reported by [1] is computed on the input graph (*i.e.*, without applying reduction rules). From Table 7, we can see that NearLinear reports a slightly tighter upper bound; moreover, our upper bound is obtained as a by-product without any extra cost.

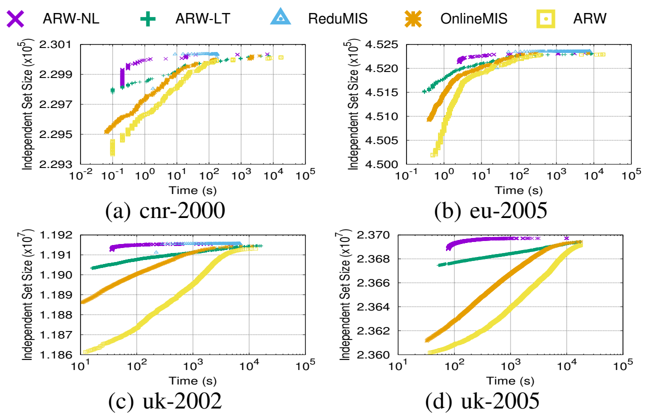


Figure 15: Convergence plots for local search algorithms

Experimental Results of Boosting ARW by NearLinear on the other Four Graphs. The results of evaluating, ARW-NL, ARW-LT, ReduMIS, OnlineMIS, and ARW on graphs *cnr-2000*, *eu-2005*, *uk-2002* and *uk-2005*, are shown in Figure 15. The accuracy of the first independent set outputted by ARW-NL compared with the largest one among all independent sets outputted by these algorithms during a five-hour run on these four graphs are 99.908%, 99.949%, 99.973%, and 99.962%, respectively. The general trend is the same as in Figure 10.

A.5 The Local Search Algorithm ARW

The local search algorithm ARW was proposed in [2] for iteratively improving the sizes of independent sets, based on the *Iterated Local Search* (ILS) metaheuristic. Given an initial independent set, the ARW algorithm runs in iterations with each iteration consisting of a *perturbation* and a *local search* step. The perturbation step is used for diversification, which forces vertices into the solution and at the same time removes their neighbors from the solution. In most cases, a single vertex is forced into the solution, and with a small probability, $f > 1$ vertices are forced into the solution; that is, f is set to $i + 1$ with probability $1/2^i$. The vertices to be forced into a solution are randomly picked from a set of candidates, with priority given to those that are outside the solution for the longest time.

The second step is the local search step, which gradually improves the current solution by using $(1, 2)$-swaps; a $(1, 2)$-swap removes 1 vertex from the current solution and inserts 2 vertices into the solution. By using a data structure that facilitates the insertion and deletion operations on a vertex to be implemented in linear time to its degree, efficient techniques are developed in [2] for finding a valid $(1, 2)$-swap in $O(m)$ time if it exists. In our implementation of ARW, the initial independent set is computed by DU and then improved by running one iteration of the local search.

[4]http://networkx.github.io/
[5]http://www.cse.psu.edu/~madduri/software/GTgraph/