Journal of Artificial Intelligence Research 59 (2017) 463-494 Submitted 01/17; published 07/17

Finding A Small Vertex Cover in Massive Sparse Graphs: Construct, Local Search, and Preprocess

Shaowei Cai SHAOWEICAI.CS@GMAIL.COM
State Key Laboratory of Computer Science,
Institute of Software, Chinese Academy of Sciences, Beijing, China

Jinkun Lin JKUNLIN@GMAIL.COM
School of Electronics Engineering and Computer Science,
Peking University, Beijing, China

Chuan Luo CHUANLUOSABER@GMAIL.COM
Institute of Computing Technology,
Chinese Academy of Sciences, Beijing, China

Abstract

The problem of finding a minimum vertex cover (MinVC) in a graph is a well known NP-hard combinatorial optimization problem of great importance in theory and practice. Due to its NP-hardness, there has been much interest in developing heuristic algorithms for finding a small vertex cover in reasonable time. Previously, heuristic algorithms for MinVC have focused on solving graphs of relatively small size, and they are not suitable for solving massive graphs as they usually have high-complexity heuristics. This paper explores techniques for solving MinVC in very large scale real-world graphs, including a construction algorithm, a local search algorithm and a preprocessing algorithm. Both the construction and search algorithms are based on low-complexity heuristics, and we combine them to develop a heuristic algorithm for MinVC called FastVC. Experimental results on a broad range of real-world massive graphs show that, our algorithms are very fast and have better performance than previous heuristic algorithms for MinVC. We also develop a preprocessing algorithm to simplify graphs for MinVC algorithms. By applying the preprocessing algorithm to local search algorithms, we obtain two efficient MinVC solvers called NuMVC2+p and FastVC2+p, which show further improvement on the massive graphs.

1. Introduction

The proliferation of massive data sets has brought a series of computational challenges, as existing algorithms usually become ineffective on massive data sets, and for most problems we need to develop new algorithms. Many data sets can be modeled as graphs, and the study of massive real-world graphs, also called complex networks, grew enormously in the last decade. In this work, we consider the Minimum Vertex Cover (MinVC) problem and propose effective techniques for addressing this problem on massive graphs.

Given an undirected graph $G = (V, E)$, a *vertex cover* is a subset $S \subseteq V$, such that each edge in G has at least one endpoint in S. Alternatively, a vertex cover is a set of vertices whose removal completely disconnects a graph. The MinVC problem requires us to find the minimum sized vertex cover in a graph. MinVC is a prominent combinatorial optimization problem with important applications, including network security, industrial machine assignment and applications in sensor networks such as monitoring link failures, facility location and data aggregation (Kavalci, Ural, & Dagdeviren, 2014). It is also closely related to the Maximum Independent Set (MaxIS) problem, which has applications in social networks, pattern recognition, molecular biology and economics (Jin & Hao, 2015).

There are important real-world tasks that call for solving MinVC on massive graphs, which mainly come from social networks and molecular biology. For example, consider the case where one has to select the minimum set of influential nodes in a social network such that some critical information is propagated to all nodes in the network in a single hop (or few hops). One solution of this problem is to determine an approximate MinVC of the network and use the nodes in MinVC for propagating information (Yadav, Sadhukhan, & Rao, 2016). In the genetic analysis of gene transcription, the concept of vertex cover is used to determine the identity, proportion and number of transcripts connected to individual phenotypes and quantitative trait loci (QTL) regulatory models (Chesler & Langston, 2005).

MinVC is a classical NP-hard problem and remains intractable even for cubic graphs and planar graphs with a maximum degree at most three (Garey & Johnson, 1979). Furthermore, it is NP-hard to approximate MinVC within any factor smaller than 1.3606 (Dinur & Safra, 2005), although one can achieve an approximation ratio of $2 - o(1)$ (Karakostas, 2005).

1.1 Previous Heuristics and Motivations

Due to its NP-hardness, research into MinVC solving has been concentrated on heuristic algorithms for finding a "good" vertex cover in reasonable time. Heuristic algorithms for NP-hard computational problems can be mainly divided into heuristic construction algorithms and local search algorithms (Hoos & Stützle, 2004).

In the context of MinVC, construction algorithms generate a vertex cover by extending a partial solution, i.e., a vertex set. A construction algorithm for MinVC starts from an empty vertex set, and then iteratively adds vertices into the set, until it becomes a vertex cover. Two typical construction algorithms for MinVC include the maximal matching based algorithm, and a greedy construction algorithm which at each iteration adds the vertex that covers the most uncovered edges. These two algorithms are so classical that they are included in a well-known textbook of combinatorial optimization (Papadimitrious & Steiglitz, 1982). However, construction algorithms alone do not provide good-quality solutions in practice, although they are of interest from a theoretical viewpoint. Due to this reason, practical work on construction algorithms for MinVC is rare. In practice, construction algorithms for MinVC are usually used to generate an initial solution for local search algorithms.

Local search is perhaps the most popular practical heuristic approach to NP-hard combinatorial optimization problems. Seen from the literature, a general scheme for local search algorithms for MinVC is as follows. It first uses a construction algorithm to obtain a vertex cover. Whenever it finds a vertex cover, it removes a vertex from the solution, and then iteratively performs small modifications to the candidate vertex set, such as removing a vertex, adding a vertex, or swapping a vertex pair, until the vertex set becomes a vertex cover. This process is repeated until a satisfactory solution is returned or a preset time limit is reached. There has been considerable interest in local search algorithms for MinVC in the last decade, e.g., (Richter, Helmert, & Gretton, 2007; Andrade, Resende, & Werneck, 2008; Pullan, 2009; Cai, Su, & Sattar, 2011; Cai, Su, Luo, & Sattar, 2013). In particular, a recent algorithm called NuMVC (Cai et al., 2013), which outperforms other heuristic algorithms on a broad range of benchmarks, makes a significant improvement in MinVC solving.

Previous local search algorithms for MinVC are mainly evaluated on randomly generated benchmarks and two benchmarks namely the DIMACS and BHOSLIB benchmark sets (Richter et al., 2007; Andrade et al., 2008; Pullan, 2009; Cai et al., 2011, 2013). The DIMACS and BHOSLIB are the two most popular benchmark sets for testing MinVC (also MaxIS and Maximum Clique) algorithms, as they are generally difficult to solve, and some DIMACS graphs arise from real-world applications. To improve the performance on these benchmarks, many sophisticated heuristics have been proposed and tested. Recent heuristics include max-gain vertex pair selection (Richter et al., 2007), edge weighting (Richter et al., 2007; Cai et al., 2011), k-improvement (also called $(k-1,k)$ swap) (Andrade et al., 2008), configuration checking (Cai et al., 2011), minimum loss removing and two-stage exchange (Cai et al., 2013). Most of the previous heuristics do not have sufficiently low complexity. Because the benchmark graphs used for testing previous algorithms are not large (usually with less than five thousand vertices), the complexity of heuristics did not show an obvious impact on the performance. However, for massive graphs where the size is much larger (e.g., with millions of vertices), the high complexity severely limits the ability of algorithms to handle these data sets.

Massive graphs call for new heuristics and algorithms. However, there is little work being done on heuristic algorithms for massive graphs. In particular, our work was the first research on local search for MinVC on massive graphs, when it was first presented in the IJCAI 2015 conference (Cai, 2015). We also study construction and preprocessing algorithms for MinVC. This work also shows that, when designing algorithms for solving problems on massive graphs, a key issue is on making a good balance between the time complexity and the effectiveness of heuristics.

1.2 Main Contributions

This paper focuses on solving massive sparse instances of the MinVC problem in practice. The main technical contributions of this paper are as follows.

1. We propose a new construction algorithm for MinVC called EdgeGreedyVC. We show theoretically that EdgeGreedyVC always returns a minimal vertex cover with a linear complexity. Also, experimental results on real-world massive graphs demonstrate that it achieves a good balance between solution quality and run time when we compare it with previous construction algorithms.

2. We propose a new local search algorithm for MinVC called FastVC. A novel technique in FastVC is a probabilistic heuristic named Best from Multiple Selections (BMS), which returns a good-quality vertex from a large set of candidate vertices with a very high probability. The BMS heuristic approximates the minimum loss removing heuristic (Cai et al., 2013) very well and lowers the complexity from $O(|V|)$ to $O(1)$. We carry out experiments to evaluate FastVC on massive real-world graphs, compared with a representative of the state of the art algorithm named NuMVC as well as its variant NuMVC$_e$ which uses the same construction heuristic as FastVC. Experimental results show that FastVC finds significantly better quality vertex covers than NuMVC and NuMVC$_e$ on most instances.

3. We improve two previous construction algorithms by adding a shrinking phase and by using an efficient data structure. Then we integrate all three construction algorithms into both

NuMVC and FastVC, leading to two improved local search algorithms for MinVC named NuMVC2 and FastVC2.

4. We develop a two-phase preprocessing algorithm to simplify graphs for MinVC algorithms. Experimental results show that the preprocessing algorithm is effective and efficient. By applying the preprocessing algorithm, we further improve NuMVC2 and FastVC2 and develop two more efficient MinVC solvers called NuMVC2+p and FastVC2+p.

This paper is an extended and improved version of a conference paper (Cai, 2015). New contributions in this paper include parts of the first and second contributions (the study and comparison of construction algorithms, the experiments with NuMVC$_e$), as well as the whole third and fourth contributions. Also, while experiments are only performed on some typical instances in the conference paper, experiments in this paper are performed on the complete set of benchmark instances.

1.3 Structure of the Paper

In the next section, we introduce some preliminary knowledge, including definitions and notation, preliminaries of local search for MinVC, as well as the benchmarks and experiment methodology in this work. In Section 3, we investigate previous construction algorithms for MinVC and propose a new construction algorithm called EdgeGreedyVC, and compare EdgeGreedyVC with previous construction algorithms. In Section 4, we describe the local search algorithm FastVC and present the key function based on the BMS heuristic, and carry out experiments to evaluate FastVC. In Section 5, we improve two previous construction algorithms and integrate all the three construction algorithms into both NuMVC and FastVC, leading to two improved local search algorithms named NuMVC2 and FastVC2. In section 6, we develop a preprocessing algorithm for MinVC and apply it to further improve NuMVC2 and FastVC2, leading to NuMVC2+p and FastVC2+p. Finally, we give some concluding remarks.

2. Preliminaries

In this section, we first introduce the basic definitions and natation that will be used in this paper, and then we give some preliminaries about local search for MinVC. Finally, we introduce the benchmarks and the experimental methodology that we use in our experiments.

2.1 Basic Definitions and Notation

An undirected graph $G = (V, E)$ consists of a vertex set V and an edge set E where each edge is a 2-element subset of V. For an edge $e = \{u, v\}$, we say that vertices u and v are the *endpoints* of edge e. For convenience of discussions on complexity, we define $n = |V|$ and $m = |E|$. Two vertices are neighbors if and only if they both belong to some edge. The neighborhood of a vertex v is denoted as $N(v) = \{u \in V | \{u, v\} \in E\}$, and the closed neighborhood as $N[v] = \{v\} \cup N(v)$. The *degree* of a vertex v is defined as $deg(v) = |N(v)|$.

For an undirected graph $G = (V, E)$, a *vertex cover* of a graph is a subset of V that contains at least one of the two endpoints of each edge. A vertex cover is minimal if taking any vertex out of it would make it not a vertex cover. An *independent set* is a subset of V where no two vertices are neighbors. A vertex set S is a vertex cover of G if and only if $V \setminus S$ is an independent set of G. We are concerned in this paper with the problem of finding a vertex cover as small as possible (MinVC). Equivalently, this problem can be viewed as seeking as large an independent set as possible, which also has important applications.

Given an undirected graph $G = (V, E)$, a *candidate solution* for MinVC is a subset of vertices $X \subset V$. An edge $e \in E$ is *covered* by a candidate solution X if at least one endpoint of e belongs to X, and otherwise we say it is *uncovered* by X. For convenience, in the rest of this paper, we use C to denote the current candidate solution. A vertex has two states: selected for covering (i.e., $v \in C$), or not selected (i.e., $v \notin C$). The age of a vertex is the number of steps since its state was last changed.

Given an undirected graph G and a candidate solution X for MinVC, for a vertex $v \in X$, the *loss* of v, denoted as *loss*(v, X), is defined as the number of covered edges that would become uncovered by removing v from X; for a vertex $v \notin X$, the *gain* of v, denoted as *gain*(v, X), is defined as the number of uncovered edges that would become covered by adding v into X. In this work, when talking about loss and gain of vertices, the candidate solution always refers to the current candidate solution C and thus it is omitted. We write *loss*(v) and *gain*(v) for *loss*(v, C) and *gain*(v, C), for the sake of convenience. Both loss and gain are *scoring properties* of vertices.

2.2 Preliminaries of Local Search for MinVC

One popular way to solve the MinVC problem is based on iteratively solving its decision version — given a positive integer number k, searching for a k-sized vertex cover. The general scheme is as follows: At the beginning, a vertex cover is constructed; whenever the algorithm finds a vertex cover of k vertices, one vertex is removed from the vertex cover[1], and the algorithm starts from the resulting vertex set to search for a vertex set of $k - 1$ vertices that covers all edges (i.e., a vertex cover of $k - 1$ vertices) by performing local search. When the algorithm terminates, it outputs the smallest vertex cover it has found.

For local search MinVC algorithms that are based on iteratively solving the decision problem, each search step consists of exchanging a pair of vertices: a vertex $u \in C$ is removed from C, and a vertex $v \notin C$ is put into C. Such a step is called an exchange step. In the literature, there are two ways to perform an exchange step. The first one is adopted by algorithms before NuMVC, which chooses a vertex pair from candidate vertex pairs, and then exchanges them and updates scoring properties accordingly. The second method, proposed in NuMVC and named two-stage exchange, works in a "separate" fashion: it first chooses a vertex $u \in C$ and removes it, and updates scoring properties accordingly, and then chooses a vertex $v \notin C$ and adds it, and updates scoring properties accordingly.

2.3 Benchmarks and Experiment Methodology

In this work, in order to study the algorithms, we carry out extensive experiments and report the results in tables. In this subsection, we introduce the benchmarks, the experiment setup and reporting methodology, so that the readers can understand the experiment parts more easily.

For our experiments, we collected all undirected simple graphs (not including DIMACS and BHOSLIB graphs) we could find from the Network Data Repository online (Rossi & Ahmed,

1. If after removing one vertex, the vertex set remains a vertex cover, then more vertices are removed until it is not a vertex cover.

2015).[2] All these graphs are generated from real-world applications. Many of these real-world graphs have millions of vertices and dozens of millions of edges, while at the same time being quite sparse. We calculate the density of each graph, i.e., $m/\binom{n}{2}$, and the averaged density of these graphs is 0.00859, while the maximum one is 0.347. We also calculate the averaged degree $2m/n$ for each graph, and the averaged value of these figures is 26.15, while the maximum one is 181.19. Some of these benchmarks have recently been used in testing algorithms for Maximum Clique and Coloring problems (Rossi & Ahmed, 2014; Rossi, Gleich, Gebremedhin, & Patwary, 2014; Wang, Cai, & Yin, 2016; Cai & Lin, 2016). There are 102 graphs in total in this suite of benchmarks. The graphs can be grouped into 11 classes, including biological networks, collaboration networks, interaction networks, infrastructure networks, Amazon recommendation networks, Tweeter networks, Facebook networks, scientific computation networks, social networks, technological networks, and web linkage networks, in the order of their appearance in the tables. There is also a group of temporal reachability networks, where the graphs are small (usually with several hundreds of vertices) and the algorithms find the same quality solution on all the graphs, and thus are not included in our experiments.

All the algorithms in our experiments, either developed in this work or not, are implemented in the C++ programming language by their authors, and have been complied by g++ (version 4.4.5) with the '-O3' option for our experiments. All experiments are carried out on a workstation under Ubuntu Linux (version 14.04), using 2 cores of an Intel i7-4800MQ 2.5 GHz CPU and 32 GByte RAM.

Most experiments in this work involve comparing the solution quality and run time of different MinVC algorithms. In particular, all local search algorithms (sometimes combined with a preprocessing algorithm) are executed 10 times with the same random seeds ({1,2,...,10}) on each instance with a time limit of 1000 seconds for each execution. For each algorithm on each instance, we report three metrics:

- The minimum size of vertex cover found by the algorithm among the 10 executions, denoted by 'min' in the tables.
- The averaged size of vertex covers found by the algorithm over the 10 executions, denoted by 'avg' in the tables. These two metrics about solution quality are presented together in one column 'min(avg)' for each algorithm.
- The averaged run time to identify the final vertex cover over the 10 executions, where the run time in an execution is the time to find the best found solution in that execution. The average run time is denoted by 'time' in the tables. In our experiments, the time is CPU time (measured in seconds), rather than wall clock time.

When comparing different algorithms, we put a higher priority on the solution quality than the run time, as in previous literatures for MinVC (Richter et al., 2007; Pullan, 2009; Cai et al., 2011, 2013) and the international algorithm competition for the NP-hard combinatorial optimization problems such as maximum satisfiability (Argelich, Li, Manyà, & Planes, 2016). In detail, the rules of algorithm comparison and the reporting method are as follows:

1. For two algorithms A and B on an instance, we say algorithm A performs better than algorithm B w.r.t. solution quality, if and only if (<u>'min' of A < 'min' of B</u> &

2. http://www.graphrepository.com/networks.php, accessed on Jan. 2015.

'avg' of A $\leq$ 'avg' of B) or ('min' of A $\leq$ 'min' of B & 'avg' of A $<$ 'avg' of B); we say algorithm A and algorithm B have the same performance w.r.t. solution quality if and only if ('min' of A $=$ 'min' of B & 'avg' of A $=$ 'avg' of B).

2. The algorithm that has the best performance w.r.t. solution quality is considered as the best algorithm for the instance. If more than one algorithms has the same solution quality which is better than other algorithms, then they are considered equally the best for the instance. The best 'min' and the best 'avg' values are indicated in **bold** face.

3. If for an instance, there exist two algorithms A and B that cannot be compared in terms of solution quality (according to principle 1), then we say there is not a clear dominant algorithm for that instance. In this case, the best 'min' and the best 'avg' values are also indicated in **bold** face, even though they are obtained by different algorithms.

4. Only when all algorithms obtain the same solution quality performance, we compare the run time of the algorithms, and the algorithm with the minimum value of average run time is the best algorithm, and its averaged time is indicated in **bold** face.

3. A New Construction Algorithm for Vertex Cover

This section investigates construction algorithms for MinVC. We first review previous construction algorithms, and then propose a new construction algorithm called EdgeGreedyVC. We compare different construction algorithms through both theoretical and experimental analysis.

3.1 Previous Construction Algorithms for Vertex Cover

Seen from the literature, there are two popular construction algorithms for vertex cover. The first one is based on finding a maximal matching, which has an approximation ratio of 2 (Papadimitrious & Steiglitz, 1982). The second one is a greedy algorithm that is employed in most practical MinVC algorithms.

3.1.1 Maximal Matching based Construction Algorithm

Given a graph $G = (V, E)$, a matching M in G is a set of pairwise non-adjacent edges, that is, no two edges share a common vertex. A maximal matching of a graph G is a matching M with the property that if any edge not in M is added to M, it is no longer a matching, that is, M is maximal if it is not a proper subset of any other matching in graph G.

A well-known construction algorithm for vertex cover is to find a maximal matching in the graph, and return the vertices in the matching as a vertex cover. Let us denote the vertex set of the found maximal matching M as $V(M)$. It is easy to prove that $V(M)$ is a vertex cover. Suppose that there is an edge e not covered by $V(M)$, then e has no common vertex with all edges in M. Thus, we can extend matching M by adding edge e, obtaining a greater sized matching $M' = M \cup \{e\}$. This contradicts the fact that M is a maximal matching.

For convenience, we denote this algorithm as *MatchVC*. For a graph $G = (V, E)$, beginning with an empty vertex set C, the MatchVC algorithm can be described as follows:

For each edge $e \in E$: if e is not covered by C, add both endpoints of e into C. Return C.

It is obvious the complexity of the MatchVC algorithm is $O(m)$. This algorithm is very fast and guarantees an approximation ratio of 2. Note that the best known approximation ratio is $2 - o(1)$

(Karakostas, 2005), which is essentially not better than 2. However, the MatchVC algorithm does not return sufficiently good solutions in practice, which will also be shown in our experiments.

3.1.2 GREEDY CONSTRUCTION ALGORITHM

Another construction algorithm for MinVC is an intuitive greedy procedure based on the $gain$ values of vertices (Papadimitrious & Steiglitz, 1982). It is the most commonly used construction algorithm for MinVC and is usually used to obtain the initial solution in local search algorithms for MinVC (Richter et al., 2007; Cai et al., 2011, 2013).

For convenience, we denote this algorithm as *GreedyVC*. For a graph $G = (V, E)$, beginning with an empty vertex set C, the GreedyVC algorithm works as follows:

Repeat the following operations until C becomes a vertex cover: select a vertex $v \notin C$ with the maximum $gain$ to add into C, breaking ties randomly. Return C.

The number of iterations of this procedure equals the size of the vertex cover C, and is denoted as ℓ. We analyze the worst case complexity for two implementations of the above algorithm as follows. A straight-forward implementation is to scan the vertex set V in each iteration in order to find the objective vertex, which has a complexity of $\Theta(n)$ for each iteration. Therefore, the complexity is $\Theta(\ell \cdot n) = O(n^2)$. A more "clever" implementation is to maintain the C set and also a set of vertices not in C, which is denoted as H. In each iteration, we scan the H set to find a vertex with the maximum $gain$. To be precise, we use C_i and H_i to denote the C set and H set at the beginning of the i^{th} iteration. We have $|C_i| = i - 1$ and $|H_i| = n - |C_i| = n - (i - 1)$. Thus, $\sum_{i=1}^{\ell} |H_i| = \sum_{i=1}^{\ell}(n - (i - 1)) = \frac{1}{2}\ell(2n + 1 - \ell)$. Since $1 \leq \ell \leq n \Rightarrow \frac{1}{2}\ell(n + 1) \leq \frac{1}{2}\ell(2n + 1 - \ell) \leq \ell n$, we have $\sum_{i=1}^{\ell} |H_i| = \Theta(\ell \cdot n)$. Therefore, the complexity of this implementation is $\sum_{i=1}^{\ell} |H_i| = \Theta(\ell \cdot n) = O(n^2)$.

We will see from the experiment results in Section 3.3 that, a quadratic complexity is too high for massive graphs and makes the algorithms inefficient so that they may fail to provide a vertex cover within a reasonable amount of time (like 1000 seconds). In Section 5, we re-implement this GreedyVC algorithm by using the heap data structure, which accelerates the procedure significantly, and we also improve GreedyVC by removing redundant vertices.

3.2 The EdgeGreedyVC Algorithm

We propose a fast vertex cover construction algorithm, which is called $EdgeGreedyVC$. The pseudo-code of $EdgeGreedyVC$ is given in Algorithm 1. The $EdgeGreedyVC$ Algorithm consists of an extending phase and a shrinking phase.

The extending phase: Starting with an empty set C, the algorithm extends C by checking and covering an edge in each iteration. If the considered edge is uncovered, the endpoint with a higher degree is added into C. If the two endpoints have the same degree, we simply choose the first one. In this way, the algorithm is deterministic, since it does not utilize random choices. Obviously, we obtain a vertex cover at the end of the extending phase.

The shrinking phase: First, we calculate the *loss* values of vertices in C; then, we scan the C set and if a vertex $v \in C$ has a *loss* value of 0, it is removed, and *loss* values of its neighbors are updated accordingly.

Theorem 1. *The $EdgeGreedyVC$ procedure returns a minimal vertex cover in $O(m)$ time, where m is the number of edges.*

Algorithm 1: EdgeGreedyVC

Input: graph $G = (V, E)$
Output: vertex cover of G
1 $C := \emptyset$;
2 **foreach** $e \in E$ **do**
3   **if** e *is uncovered* **then**
4   add the endpoint of e with higher degree into C;
5 *loss*$(v) := 0$ for each $v \in C$;
6 **foreach** $e \in E$ **do**
7   **if** *only one endpoint of* e *belongs to* C **then**
8   for the endpoint $v \in C$, *loss*(v)++;
9 **foreach** $v \in C$ **do**
10   **if** *loss*$(v) = 0$ **then**
11         $C := C \backslash \{v\}$, update *loss* of vertices in $N(v)$;
12 **return** C;

Proof: We first prove that the vertex set that the $EdgeGreedyVC$ procedure returns is a minimal vertex cover. Let us use C to denote the current vertex set during the procedure.

At the beginning of the shrinking phase, C is a vertex cover. Also, each vertex removed in the shrinking phase has a *loss* value of 0, and thus removing such vertices does not generate any uncovered edge. Hence, C is a vertex cover after the shrinking phase. Now we prove that the vertex cover C after the shrinking phase is minimal. Suppose after the shrinking phase, there exists a vertex in C whose removal keeps C a vertex cover. Without loss of generality, let this vertex be v_j, the one considered at the j^{th} iteration of the shrinking phase. From the assumption, we have $loss(v_j) = 0$ at the end of the shrinking phase. Notice that during the shrinking phase, the *loss* value of any vertex in C does not decrease[3]. Thus, the value of $loss(v_j)$ at the j^{th} iteration is at most 0, but *loss* values are non-negative, so it is 0. Therefore, v_j would have been removed at the j^{th} iteration. This completes the proof by contradiction.

In the following, we calculate the time complexity of EdgeGreedyVC. The EdgeGreedyVC procedure can be divided into three parts: the first part (lines 2-4) performs the extending phase, the second part (lines 5-8) initializes the *loss* values, while the last one (lines 9-11) removes redundant vertices. Let C^+ denotes the vertex cover obtained by the extending phase. It is clear that the complexity of the extending phase is $O(m)$. For the second part, the complexity is $O(|C^+| + m)$. Since at most one vertex is added in each iteration of the extending phase, we have $|C^+| \leq m$, and thus the complexity for the second part is $O(m)$. For the last part, the complexity depends on the total number of updating operations of *loss* values, which is calculated as $\sum_{v \in C^+} deg(v) < \sum_{v \in V} deg(v) = 2m$. Therefore, the $EdgeGreedyVC$ procedure has a complexity of $O(m)$. □

Many massive real-world graphs are sparse graphs (Barabási & Albert, 1999; Eubank, Kumar, Marathe, Srinivasan, & Wang, 2004; Lu & Chung, 2006), and heuristics with $O(m)$ complexity

3. This can be easily proved according to the definition of *loss*.

are fast on such graphs. Nevertheless, we note that there are also dense graphs from real-world applications, and our method here is particularly effective for large sparse graphs.

3.3 Comparing Construction Heuristics

In this subsection, we carry out experiments to compare the three construction algorithms, namely MatchVC, GreedyVC and EdgeGreedyVC. We adopt the implementation of GreedyVC in NuMVC (2013), and we implement MatchVC using our codes of FastVC.

Since MatchVC and EdgeGreedyVC are deterministic algorithms, they are executed once (with random seed 1) on each instance. GreedyVC uses a randomized strategy to break ties, so it is executed 10 times (with random seeds from 1 to 10) on each instance and the averaged results over the 10 runs are reported. We report the size of vertex cover and the run time for each algorithm. The best solution size for each instance is presented in **bold** face. The experiment results are presented in Tables 1 and 2. We also report the size of the graphs in these two tables. The results can be summarized in the following observations.

1. The two greedy algorithms GreedyVC and EdgeGreedyVC always find better solutions than the MatchVC algorithm. GreedyVC and EdgeGreedyVC are competitive and complementary to each other. More specifically, GreedyVC finds the best solutions among all the three algorithms for 61 instances, and EdgeGreedyVC does this for 47 instances.

2. EdgeGreedyVC and MatchVC are much faster than GreedyVC. In particular, EdgeGreedyVC and MatchVC terminate within one second for all instances, while GreedyVC requires more than 100 seconds for 23 instances.

3. Overall, EdgeGreedyVC takes a very good balance between solution quality and run time, and it is a better choice when compared to GreedyVC and MatchVC.

4. A New Local Search Algorithm for MinVC

In this section, we propose a local search algorithm for MinVC called FastVC. We utilize the EdgeGreedyVC algorithm to construct the starting vertex cover for local search. Further, we propose a probabilistic method for choosing the vertex to remove in each step, which is an important idea in FastVC. Experiments are carried out to compare FastVC with the latest state of the art local search algorithm for MinVC namely NuMVC on massive graphs.

4.1 The High Level Algorithm

We first describe the FastVC algorithm from a high level. Details of important functions in FastVC and further analysis will be presented in the next subsection.

Local search algorithms for MinVC based on iteratively solving the decision problem start with a vertex cover, which we call the *starting vertex cover*. A small starting vertex cover can save the subsequent local search from too much unnecessary search before beginning seeking a good solution. A balance must be struck between the quality of the starting vertex cover and the time consumed in constructing it. Otherwise, the resulting algorithm may be inefficient in practice. FastVC uses the EdgeGreedyVC algorithm to construct the starting vertex cover.

Table 1: Comparing three construction algorithms for MinVC.

instance	|V|	|E|	GreedyVC		EdgeGreedyVC		MatchVC	
			avg size	time	size	time	size	time
bio-celegans	453	2025	**258.3**	<0.01	259	<0.01	398	<0.01
bio-diseasome	516	1188	285.2	<0.01	**285**	<0.01	400	<0.01
bio-dmela	7393	25569	**2666.9**	0.03	2717	<0.01	4076	<0.01
bio-yeast	1458	1948	462.3	<0.01	**462**	<0.01	800	<0.01
ca-AstroPh	17903	196972	11517.8	0.6	**11511**	<0.01	15376	<0.01
ca-citeseer	227320	814134	129356	88.2	**129258**	<0.01	173614	0.01
ca-coauthors-dblp	540486	15245729	472389.2	610.01	**472259**	0.04	510992	0.02
ca-CondMat	21363	91286	12519.1	0.81	**12497**	<0.01	17528	<0.01
ca-CSphd	1882	1740	555.3	<0.01	**553**	<0.01	1044	<0.01
ca-dblp-2010	226413	716460	122194.3	84.37	**122073**	<0.01	187206	0.01
ca-dblp-2012	317080	1049866	165268	168.36	**165084**	0.01	227224	<0.01
ca-Erdos992	6100	7515	**461**	<0.01	**461**	<0.01	594	<0.01
ca-GrQc	4158	13422	2220	0.01	**2214**	<0.01	3214	<0.01
ca-HepPh	11204	117619	6574.4	0.19	**6565**	<0.01	9088	<0.01
ca-hollywood-2009	1069126	56306653	864251.2	2182.47	**864186**	0.16	1040514	0.06
ca-MathSciNet	332689	820644	140695.2	125.73	**140446**	0.01	203770	<0.01
ca-netscience	379	914	**214**	<0.01	**214**	<0.01	300	<0.01
ia-email-EU	32430	54397	**820**	0.02	827	<0.01	1410	<0.01
ia-email-univ	1133	5451	**609.8**	<0.01	615	<0.01	814	<0.01
ia-enron-large	33696	180811	12825.3	1.04	**12822**	<0.01	17504	<0.01
ia-enron-only	143	623	87.3	<0.01	**87**	<0.01	126	<0.01
ia-fb-messages	1266	6451	594.7	<0.01	**592**	<0.01	932	<0.01
ia-infect-dublin	410	2765	**297.4**	<0.01	300	<0.01	372	<0.01
ia-infect-hyper	113	2196	**93**	<0.01	**93**	<0.01	108	<0.01
ia-reality	6809	7680	**81**	<0.01	**81**	<0.01	114	<0.01
ia-wiki-Talk	92117	360767	**17411.9**	2.65	17464	<0.01	26246	<0.01
inf-power	4941	6594	2277.8	0.01	**2271**	<0.01	3736	<0.01
inf-roadNet-CA	1957027	2760388	1070458.5	7409.41	**1062463**	0.04	1660956	0.02
inf-roadNet-PA	1087562	1541514	593581.3	2291.05	**588269**	0.03	915948	0.01
inf-road-usa	23947347	28854312	n/a	n/a	**12200485**	0.7	19475384	0.23
rec-amazon	91813	125704	49245.9	15.83	**48542**	<0.01	74366	<0.01
rt-retweet	96	117	**32.5**	<0.01	33	<0.01	60	<0.01
rt-retweet-crawl	1112702	2278852	**81572.8**	115.24	82531	0.02	157810	0.01
rt-twitter-copen	761	1029	238.1	<0.01	**238**	<0.01	422	<0.01
socfb-A-anon	3097165	23667394	**376526.9**	1577.91	424586	0.29	715962	0.05
socfb-B-anon	2937612	20959854	**303568.5**	1129.9	342092	0.29	586904	0.04
socfb-Berkeley13	22900	852419	**17520.6**	1.21	17599	<0.01	21652	0.01
socfb-CMU	6621	249959	**5068**	0.08	5090	<0.01	6290	<0.01
socfb-Duke14	9885	506437	**7807.4**	0.22	7838	<0.01	9422	<0.01
socfb-Indiana	29732	1305757	**23738.9**	2.17	23788	<0.01	28508	<0.01
socfb-MIT	6402	251230	**4734.6**	0.07	4756	<0.01	6014	<0.01
socfb-OR	63392	816886	**37269.7**	6.09	37402	<0.01	48342	<0.01
socfb-Penn94	41536	1362220	**31764.1**	4.1	31851	<0.01	39246	0.01
socfb-Stanford3	11586	568309	**8634.9**	0.28	8696	<0.01	10796	<0.01
socfb-Texas84	36364	1590651	**28677.5**	3.23	28762	<0.01	34876	0.01
socfb-uci-uni	58790782	92208195	**866783**	48316	869726	1	1732226	0.43
socfb-UCLA	20453	747604	**15500.7**	0.94	15553	<0.01	19226	0.01
socfb-UConn	17206	604867	**13474.9**	0.67	13527	<0.01	16466	0.01
socfb-UCSB37	14917	482215	**11479.9**	0.48	11510	<0.01	14116	<0.01
socfb-UF	35111	1465654	27911.4	3.03	**27894**	<0.01	33570	0.01
socfb-UIllinois	30795	1264421	**24532.3**	2.31	24603	<0.01	29512	<0.01
socfb-Wisconsin87	23831	835946	**18725.9**	1.33	18782	<0.01	22750	<0.01

Table 2: Comparing three construction algorithms for MinVC (continued).

instance	|V|	|E|	GreedyVC		EdgeGreedyVC		MatchVC	
			avg size	avg time	size	time	size	time
sc-ldoor	952203	20770807	858313.1	1485.45	**857967**	0.06	893510	0.03
sc-msdoor	415863	9378650	382176.7	270.99	**382115**	0.02	398824	0.02
sc-nasasrb	54870	1311227	51714.1	5.04	**51700**	<0.01	54870	0.01
sc-pkustk11	87804	2565054	**84155.6**	11.91	84355	<0.01	87804	<0.01
sc-pkustk13	94893	3260967	**89714.2**	15.23	89868	<0.01	94534	<0.01
sc-pwtk	217891	5653221	208842.5	74.78	**208255**	0.01	217804	<0.01
sc-shipsec1	140385	1707759	**119977.5**	39.69	120339	<0.01	140354	<0.01
sc-shipsec5	179104	2200076	**149495.2**	53.09	150957	<0.01	179038	<0.01
soc-BlogCatalog	88784	2093195	**20974**	3.17	21257	<0.01	29990	0.01
soc-brightkite	56739	212945	21489.4	2.98	**21469**	<0.01	31734	<0.01
soc-buzznet	101163	2763066	**31074.4**	6.6	31795	0.01	48174	<0.01
soc-delicious	536108	1365961	**87670.5**	68.91	90812	0.01	142016	0.01
soc-digg	770799	5907132	**104624.5**	102.74	106831	0.03	133384	0.01
soc-dolphins	62	159	**34.6**	<0.01	35	<0.01	50	<0.01
soc-douban	154908	327162	8718.6	1.36	**8704**	<0.01	16156	<0.01
soc-epinions	26588	100120	9867.2	0.58	**9860**	<0.01	14758	<0.01
soc-flickr	513969	3190452	154487.2	159.06	**154471**	0.02	225688	0.01
soc-flixster	2523386	7918801	**96498.4**	216.44	96873	0.04	122862	0.01
soc-FourSquare	639014	3214986	**90688**	64.71	90719	0.01	116474	<0.01
soc-gowalla	196591	950327	**85443.7**	45.3	85538	0.01	124018	0.01
soc-karate	34	78	14.1	<0.01	**14**	<0.01	22	<0.01
soc-lastfm	1191805	4519330	**79169.9**	85.85	80205	0.03	99564	0.01
soc-livejournal	4033137	27933062	**1894963.6**	19572.94	1898700	0.31	2591926	0.08
soc-LiveMocha	104103	2193083	**44176.5**	11.61	45314	0.01	62782	<0.01
soc-orkut	2997166	106349209	**2216036**	18879	2228033	0.5	2706294	0.15
soc-pokec	1632803	22301964	**860947.8**	3717.96	902600	0.19	1203086	0.04
soc-slashdot	70068	358647	**22637.3**	3.56	22665	<0.01	34056	0.01
soc-twitter-follows	404719	713319	**2323**	0.8	2419	<0.01	4644	<0.01
soc-wiki-Vote	889	2914	414.1	<0.01	**414**	<0.01	650	<0.01
soc-youtube	495957	1936748	**148168.5**	168.35	149089	0.02	240756	<0.01
soc-youtube-snap	1134890	2987624	**279088.1**	678.8	279389	0.03	456406	0.01
tech-as-caida2007	26475	53381	**3692.7**	0.15	3704	<0.01	6648	<0.01
tech-as-skitter	1694616	11094209	**529835.2**	1883.76	553244	0.09	889676	0.03
tech-internet-as	40164	85123	**5714.9**	0.38	5766	<0.01	10872	<0.01
tech-p2p-gnutella	62561	147878	15838.6	2.01	**15759**	<0.01	28238	<0.01
tech-RL-caida	190914	607610	**75681.1**	35.72	77990	<0.01	120890	<0.01
tech-routers-rf	2113	6632	805.5	<0.01	**803**	<0.01	1376	<0.01
tech-WHOIS	7476	56943	**2297.9**	0.03	2305	<0.01	3380	<0.01
web-arabic-2005	163598	1747269	115499.4	41.66	**115256**	<0.01	138022	0.01
web-BerkStan	12305	19500	**5496.1**	0.14	5565	<0.01	8432	<0.01
web-edu	3031	6474	1591.2	<0.01	**1451**	<0.01	2818	<0.01
web-google	1299	2773	**498.8**	<0.01	499	<0.01	748	<0.01
web-indochina-2004	11358	47606	7423.8	0.17	**7367**	<0.01	9720	<0.01
web-it-2004	509338	7178413	415772.1	542.39	**415521**	0.01	447464	0.02
web-polblogs	643	2280	**246**	<0.01	**246**	<0.01	406	<0.01
web-sk-2005	121422	334419	58529.1	17.77	**58375**	<0.01	84604	<0.01
web-spam	4767	37375	**2345.2**	0.02	2352	<0.01	3566	<0.01
web-uk-2005	129632	11744049	**127774**	41.66	**127774**	0.02	128626	0.01
web-webbase-2001	16062	25593	2686.8	0.05	**2674**	<0.01	4248	<0.01
web-wikipedia2009	1864433	4507315	**660288.4**	3060.03	662296	0.07	1031900	0.02

Algorithm 2: FastVC (G, *cutoff*)

Input: graph $G = (V, E)$, the *cutoff* time
Output: vertex cover of G

$C := EdgeGreedyVC(G)$;
$gain(v) := 0$ for each vertex $v \notin C$;
while *elapsed time* $<$ *cutoff* **do**
    if C *covers all edges* **then**
        $C^* := C$;
        remove a vertex with minimum *loss* from C;
        continue;
    $u := ChooseRmVertex(C)$;
    $C := C \backslash \{u\}$, update *loss* and $gain$ values of vertices in $N[u]$;
    $e :=$ a random uncovered edge;
    $v :=$ the endpoint of e with greater $gain$, breaking ties in favor of the older one;
    $C := C \cup \{v\}$, update *loss* and $gain$ values of vertices in $N[v]$;
return C^*;

For the exchange step in local search, FastVC adopts the two-stage exchange framework, as it has lower complexity than the alternative paradigm based on vertex pair exchange. Indeed, thanks to the two-stage exchange framework, NuMVC performs several times more steps per second than other local search MinVC algorithms (Cai et al., 2013).

The FastVC algorithm is outlined in Algorithm 2, as described below. At the beginning, a vertex cover is constructed by the $EdgeGreedyVC$ function, which is taken as the initial candidate solution C for the algorithm. The *loss* values of vertices in C are calculated in the $EdgeGreedyVC$ function. For vertices outside C, their $gain$ values are set to 0, as at this point all edges are covered by C and adding any vertex into C would not increase the number of covered edges.

Now we introduce the exchange step in FastVC. At each step, the algorithm first chooses a vertex in $u \in C$ to remove, which is accomplished by the $ChooseRmVertex$ function. Then, the algorithm picks a random uncovered edge e, and chooses one of e's endpoints with the greater $gain$ and adds it into C, breaking ties in favor of the older one. Note that along with removing or adding a vertex, the *loss* and $gain$ values of the vertex and its neighbors are updated accordingly.

4.2 Best from Multiple Selections (BMS)

A critical function of FastVC is $ChooseRmVertex$, which returns a vertex from the candidate vertex set C to remove in each exchange step. We propose a fast and effective heuristic for doing this task, which strikes a good balance between the time complexity and the quality of the selected vertex (w.r.t. the *loss* value).

Local search algorithms usually need to select an element from a candidate set. Perhaps the most commonly used strategy is to choose the best element according to some criterion, which we refer to as "best-picking" heuristic. With a suitable criterion, this heuristic guides the search towards the most promising area, and is thus widely adopted in local search algorithms. Recent examples of such heuristics for MinVC include the max-gain pair selection heuristic in COVER (Richter et al.,

2007) and the minimum loss removing heuristic in NuMVC (Cai et al., 2013). More examples can be found in local search algorithms for other famous NP-hard problems, such as the Satisfiability problem (Selman, Levesque, & Mitchell, 1992; Hoos & Stützle, 2004; Li & Huang, 2005). Indeed, a lot of works on local search have been focused on the criterion for filtering the candidate set and the function for comparing elements, and once this is done, they simply pick the best one. The "best-picking" heuristic works well in most cases, but not for massive data sets where the candidate set is usually very large and finding the best element is very time-consuming.

We propose a cost-effective heuristic called Best from Multiple Selections (BMS), for picking a good element from a set. For a set S, the BMS heuristic works as follows:

Choose k elements randomly with replacement from the set S, and then return the best one (w.r.t. some comparison function f), where k is a parameter.

Algorithm 3: Best from Multiple Selection (BMS) Heuristic

Input: A set S, a parameter k, a comparison function f
/*assume f is a function such that we say an element is better than another one if it has smaller f value*/
Output: an element of S

$best$:=a random element from S;
for $iteration := 1$ **to** $k - 1$ **do**
  r :=a random element from S;
  if $f(r) < f(best)$ **then** $best := r$;
return $best$;

A more formal description of the BMS heuristic is given in Algorithm 3. Let us look at how well the BMS heuristic approximates the "best-picking" heuristic. For a real number $\rho \in (0, 1)$, the probability of the event $E = \{$the f value of the element chosen by BMS is not greater than $\rho|S|$ elements in the set $S\}$ is $Pr(E) \geq 1 - (\frac{\rho|S|-1}{|S|})^k > 1 - \rho^k$ (the "$\geq$" is because there might be the case that more than one elements in those $\rho|S|$ elements have the same f value, which is the minimum among f values of all the $\rho|S|$ elements).

For $ChooseRmVertex$, the comparison function f is simply the *loss* function on vertices, and we set $k = 50$. Then, the probability that the BMS heuristic chooses a vertex whose *loss* value is not greater than 90% vertices in C is $Pr(E) > 1 - 0.9^{50} > 0.9948$. The above calculations illustrate that the BMS heuristic returns a vertex of good quality with a very high probability.

The complexity of the BMS heuristic is $O(k) = O(1)$, since k is a constant. This is lower than $O(|C|)$ for the minimum loss heuristic used by previous local search algorithms for MinVC. Note that BMS is a generic heuristic and can be also applied to improve the time efficiency of local search algorithms for large scale instances of other problems.

4.3 Experiments on FastVC

We carry out experiments to evaluate FastVC on the real-world massive graphs, compared against the state of the art local search MinVC algorithm NuMVC. To illustrate the effectiveness of the local search procedure of FastVC, we also test a modified version of NuMVC (dubbed NuMVC$_e$) which uses the EdgeGreedyVC construction heuristic with the same implementation as FastVC.

The results show that FastVC significantly outperforms NuMVC and NuMVC$_e$ on these massive graphs.

FastVC is built on the publicly available codes of NuMVC (2013) and uses the same data structure and the same implementation for the exchange step, yet it is simpler and lighter than NuMVC. Parameter settings of FastVC: For the BMS heuristic in the $ChooseRmVertex$ function of FastVC, we set the k parameter to 50, as mentioned in the previous section. This is based on preliminary experiments testing FastVC with different k values. We test $k \in [10, 100]$ with an increment step of 10. When $k < 10$ or $k > 100$, the performance of the algorithm is obviously worse than that under $k \in [10, 100]$. We observe that when $k \in [30, 100]$, the performance is quite close, and FastVC with $k = 50$ finds better solutions than the algorithm with $k = 30, 40$; also, FastVC with $k = 50$ usually finds the same quality solutions as running the algorithm with $k > 50$, but is usually faster.

For comparisons, we use the NuMVC algorithm (Cai et al., 2013) to represent the state of the art in solving the MinVC (and also MaxIS) problem. Based on experiments on DIMACS and BHOSLIB benchmarks, NuMVC is more reliably in finding the optimal or best known solution at speeds at least several times faster than earlier algorithms for MinVC and Maximum Independent Set (Cai et al., 2013). It is acknowledged as the latest breakthrough for MinVC solving in the literature (Fang, Chu, Qiao, Feng, & Xu, 2014; Rosin, 2014; Jin & Hao, 2015). Slightly better results have been reported for two algorithms built on the top of NuMVC (Fang et al., 2014; Cai, Lin, & Su, 2015), but this does not materially change our conclusions below. The source code of NuMVC is online http://lcs.ios.ac.cn/~caisw/Code/NuMVC-Code.zip and also implemented in C++. NuMVC$_e$ is implemented in the source code of NuMVC by replacing the construction algorithm with the GreedyEdgeVC algorithm in FastVC.

The experiment comparing FastVC against NuMVC and NuMVC$_e$ is conducted according to the experiment protocol in Section 2.3, and the results are reported in Tables 3 and 4. The results demonstrate that FastVC has better performance than NuMVC and NuMVC$_e$. In detail, we have the following observations:

1. FastVC finds better vertex covers than NuMVC for 52 graphs and finds the same quality solutions for 40 graphs. FastVC finds worse solutions than NuMVC only for 10 graphs, and for 5 out of these 10 graphs, the two algorithms finds nearly the same quality solutions (with a gap of at most one vertex between the averaged sizes).

2. FastVC and NuMVC have similar performance on four classes of benchmarks, namely biological networks, interaction networks, Tweeter networks and technological networks. For other classes of benchmarks, FastVC significantly outperforms NuMVC.

3. Comparing their averaged run time, we found that FastVC is much faster than NuMVC on most of the graphs. In particular, for the 40 graphs where both algorithms find the same quality solutions, we compare the averaged time to obtain the final solution. FastVC is faster on 19 graphs, while NuMVC is faster on only 2 instances, and for the rest of the instances both algorithms have an averaged time of less than 0.01 seconds.

4. Although NuMVC$_e$ shows improvement over NuMVC, particularly on those very large instances where NuMVC fails to provide a solution, the solutions returned by NuMVC$_e$ are still worse than FastVC on most instances.

Table 3: Comparing FastVC with NuMVC and NuMVC$_e$, where NuMVC$_e$ is modified from NuMVC that uses the same construction algorithm with the same implementation as FastVC.

instance	NuMVC		NuMVC$_e$		FastVC	
	min(avg)	time	min(avg)	time	min(avg)	time
bio-celegans	249(249)	<0.01	249(249)	<0.01	249(249)	<0.01
bio-diseasome	285(285)	<0.01	285(285)	<0.01	285(285)	<0.01
bio-dmela	2630(2630)	0.9	2630(2630)	1.85	2630(2630)	**<0.01**
bio-yeast	456(456)	<0.01	456(456)	<0.01	456(456)	<0.01
ca-AstroPh	11483(11483)	10.92	11483(11483)	29.33	11483(11483)	**0.02**
ca-citeseer	129193(129195.5)	153	129193(129193.6)	59.58	129193(**129193**)	1.07
ca-coauthors-dblp	472251(472258.5)	998.45	472180(472181.5)	935.05	**472179(472179)**	14.57
ca-CondMat	12480(12480)	46.59	12480(12480)	36.97	12480(12480)	**0.02**
ca-CSphd	550(550)	<0.01	550(550)	<0.01	550(550)	<0.01
ca-dblp-2010	121971(121973.8)	143.77	121972(121972.7)	60.33	**121969(121969)**	1.47
ca-dblp-2012	164952(164956.4)	261.46	164951(164953.9)	90.54	**164949(164949)**	4.26
ca-Erdos992	461(461)	<0.01	461(461)	<0.01	461(461)	<0.01
ca-GrQc	2208(2208)	0.16	2208(2208)	<0.01	2208(2208)	<0.01
ca-HepPh	6555(6555)	18.9	6555(6555)	6.59	6555(6555)	**<0.01**
ca-hollywood-2009	n/a	n/a	864115(864115.4)	990.79	**864052(864052)**	25.59
ca-MathSciNet	139982(139989.7)	174.14	139982(139985.2)	43.44	**139951(139951)**	4.36
ca-netscience	214(214)	<0.01	214(214)	<0.01	214(214)	<0.01
ia-email-EU	820(820)	0.02	820(820)	<0.01	820(820)	<0.01
ia-email-univ	594(594)	<0.01	594(594)	<0.01	594(594)	<0.01
ia-enron-large	12781(12781)	52.47	12781(12781)	121.21	12781(12781)	**0.04**
ia-enron-only	86(86)	<0.01	86(86)	<0.01	86(86)	<0.01
ia-fb-messages	578(578)	<0.01	578(578)	<0.01	578(578)	<0.01
ia-infect-dublin	293(**293**)	<0.01	293(**293**)	<0.01	293(293.2)	480.11
ia-infect-hyper	90(90)	<0.01	90(90)	<0.01	90(90)	<0.01
ia-reality	81(81)	<0.01	81(81)	<0.01	81(81)	<0.01
ia-wiki-Talk	17288(17288.2)	510.58	17288(17288.4)	704.41	17288(**17288**)	0.08
inf-power	2203(2203)	1.05	2203(2203)	1.54	2203(2203)	**<0.01**
inf-roadNet-CA	n/a	n/a	1044047(1044195.7)	1000	**1001273(1001311.2)**	896.33
inf-roadNet-PA	n/a	n/a	560365(560410.8)	947.85	**555220(555243)**	652.14
inf-road-usa	n/a	n/a	12198972(12198983.8)	1000	**12049567(12050440.1)**	1000
rec-amazon	47655(47672.6)	953.23	47671(47681.9)	943.31	**47606(47606)**	0.89
rt-retweet	32(32)	<0.01	32(32)	<0.01	32(32)	<0.01
rt-retweet-crawl	**81043(81047.2)**	119.17	81057(81059.7)	7.637	81048(81048)	1.34
rt-twitter-copen	237(237)	<0.01	237(237)	<0.01	237(237)	<0.01
socfb-A-anon	n/a	n/a	375234(375236.2)	677.28	**375231(375232.8)**	128.55
socfb-B-anon	n/a	n/a	303049(303050)	466.86	**303048(303048.9)**	85.1
socfb-Berkeley13	17216(17218)	515.58	17215(17218.1)	517.30	**17210(17212.7)**	290.4
socfb-CMU	4986(**4986.1**)	248.73	4986(**4986.1**)	158.95	4986(4986.5)	3.91
socfb-Duke14	7683(7683.6)	212.95	7683(7683.6)	149.95	**7683(7683)**	228.3
socfb-Indiana	23320(23326.5)	556.5	23322(23327.9)	455.31	**23315(23317.1)**	517.27
socfb-MIT	4657(4657)	**8.2**	4657(4657)	10.06	4657(4657)	41.13
socfb-OR	36558(36560.6)	564.13	36557(36560.6)	678.19	**36548(36549.2)**	144.06
socfb-Penn94	31179(31183.3)	559.12	31178(31181.6)	632.33	**31162(31164.8)**	552.92
socfb-Stanford3	8518(8518)	8.77	8518(8518)	8.84	**8517(8517.9)**	101
socfb-Texas84	28174(28180.5)	639.12	28176(28184.2)	636.72	**28167(28171.1)**	495.91
socfb-uci-uni	n/a	n/a	**866768(866768)**	488.22	**866768(866768)**	37.05
socfb-UCLA	15225(15227.1)	399.21	15225(15227.4)	489.63	**15223(15224.3)**	297.92
socfb-UConn	13231(13233)	611.89	13231(13233.2)	565.24	**13230(13231.5)**	304.11
socfb-UCSB37	11262(11263)	381.45	11262(**11262.8**)	367.34	**11261**(11263)	210.62
socfb-UF	27319(27323.5)	607.36	27319(27323.5)	400.43	**27306(27309)**	459.23
socfb-UIllinois	24096(24106.4)	545.24	24100(24106.7)	534.74	**24091(24092.2)**	477.69
socfb-Wisconsin87	18388(18390.6)	733.22	18390(18392.2)	505.07	**18383(18385.1)**	295.36

5. The NuMVC2 and FastVC2 Algorithms

We observe that the construction algorithms GreedyVC and MatchVC can be improved by applying the shrinking phase of EdgeGreedyVC after construction of the vertex cover is finished. Also,

Table 4: Comparing FastVC with NuMVC and NuMVC$_e$ (continued).

instance	NuMVC		NuMVC$_e$		FastVC	
	min(avg)	time	min(avg)	time	min(avg)	time
sc-ldoor	n/a	n/a	856920(856930)	997.46	**856755(856757.4)**	218.94
sc-msdoor	381569(381574.6)	986.61	381564(381565.6)	973.41	**381558(381558.9)**	18.71
sc-nasasrb	51244(**51246.7**)	781.64	51245(51247.2)	781.34	51244(51247.3)	517.13
sc-pkustk11	83911(**83911**)	539.67	83911(**83911**)	571.60	83911(83912.5)	6.84
sc-pkustk13	89218(89222)	848.29	89219(89222)	913.69	**89217(89220.6)**	315.23
sc-pwtk	207749(207756.3)	249.46	207741(207746.1)	147.74	**207716(207719.9)**	184.89
sc-shipsec1	117477(117536.9)	998.33	117468(117523.7)	995.95	**117318(117337.5)**	722.4
sc-shipsec5	147288(147324.5)	990.71	147326(147339.9)	991.17	**147137(147173.8)**	569.7
soc-BlogCatalog	20752(**20752**)	394.43	20752(20752.2)	516.62	20752(**20752**)	0.19
soc-brightkite	21192(21193.5)	841.9	21191(21192.8)	890.157	**21190(21190)**	0.2
soc-buzznet	**30613(30613.2)**	657.14	**30613**(30613.9)	648.62	30625(30625)	15.95
soc-delicious	**85522(85585.6)**	78.09	85696(85715.2)	947.90	85686(85696.4)	2.54
soc-digg	103303(103318.7)	111.21	103371(103387)	866.31	**103244(103245.3)**	3.34
soc-dolphins	34(34)	<0.01	34(34)	<0.01	34(34)	<0.01
soc-douban	8685(8685)	1.37	8685(8685)	0.01	8685(8685)	<0.01
soc-epinions	9757(9757)	82.8	9757(9757)	142.78	9757(9757)	**0.13**
soc-flickr	153343(153352.7)	194.34	153340(153347.4)	39.028	**153272(153272)**	18.51
soc-flixster	96319(96320.7)	217.78	96321(96322.7)	446.88	**96317(96317)**	1.61
soc-FourSquare	90125(90134.1)	835.65	90127(90132)	795.37	**90108(90109.2)**	124.9
soc-gowalla	84313(84322.6)	940.31	84316(84324.5)	919.63	**84222(84222.3)**	64.58
soc-karate	14(14)	<0.01	14(14)	<0.01	14(14)	<0.01
soc-lastfm	78692(78695)	87.43	78696(78698.2)	817.99	**78688(78688)**	0.9
soc-livejournal	n/a	n/a	1888661(1888673.1)	1000	**1869046(1869051.1)**	953.11
soc-LiveMocha	43430(43432.8)	763.05	43432(43434.7)	803.47	**43427(43427)**	17.44
soc-orkut	n/a	n/a	2205976(2206173.6)	1000	**2171296(2171380.5)**	996.66
soc-pokec	n/a	n/a	847939(848147.4)	1000	**843422(843434.9)**	772.77
soc-slashdot	**22373**(22377)	774.65	22374(22376.5)	848.95	**22373(22373)**	0.2
soc-twitter-follows	2323(2323)	0.8	2323(2323)	0.09	2323(2323)	**0.01**
soc-wiki-Vote	406(406)	<0.01	406(406)	<0.01	406(406)	<0.01
soc-youtube	146456(146468.2)	213.06	146464(146475.2)	47.76	**146376(146376)**	5.67
soc-youtube-snap	277015(277025.2)	889.32	277005(277016.8)	203.18	**276945(276945)**	12.74
tech-as-caida2007	3683(3683)	2.32	3683(3683)	2.88	3683(3683)	<**0.01**
tech-as-skitter	n/a	n/a	527857(527881.2)	818.69	**527185(527195.9)**	446.49
tech-internet-as	5700(5700)	8.97	5700(5700)	33.23	5700(5700)	**0.01**
tech-p2p-gnutella	15682(15682)	24.25	15682(15682)	43.68	15682(15682)	**0.01**
tech-RL-caida	**74759(74776.5)**	979.74	75001(75015.7)	991.45	74930(74938.9)	9.62
tech-routers-rf	795(795)	0.01	795(795)	<0.01	795(795)	<0.01
tech-WHOIS	2284(2284)	0.31	2284(2284)	0.55	2284(2284)	<**0.01**
web-arabic-2005	114464(114471.8)	111.51	114454(114458.6)	64.80	**114426(114427.2)**	323.58
web-BerkStan	5384(5384)	**10**	5384(5384)	10.03	5384(5384)	25.8
web-edu	1451(1451)	0.35	1451(1451)	<**0.01**	1451(1451)	<**0.01**
web-google	498(498)	<0.01	498(498)	<0.01	498(498)	<0.01
web-indochina-2004	7300(7300)	14.54	7300(7300)	13.876	7300(7300)	**0.09**
web-it-2004	414741(414758.7)	987.89	414675(414680.8)	580.91	**414671(414676.3)**	9.92
web-polblogs	244(244)	<0.01	244(244)	<0.01	244(244)	<0.01
web-sk-2005	58199(58205.5)	304.68	58205(58209.5)	732.55	**58173(58173)**	8.67
web-spam	**2297(2297)**	0.28	**2297(2297)**	0.42	2298(2298)	<0.01
web-uk-2005	127774(127774)	41.67	127774(127774)	0.03	127774(127774)	**0.02**
web-webbase-2001	**2651(2651.9)**	48.54	2652(2652)	1.36	2652(2652)	0.01
web-wikipedia2009	n/a	n/a	649546(649571.6)	999.78	**648317(648321.8)**	692.3

the construction algorithm GreedyVC is implemented in a way with quadratic complexity in the original implementation of NuMVC. In this section, we use a heap data structure to re-implement GreedyVC in NuMVC, which leads to a significant speedup. The resulting, improved algorithms based on GreedyVC and MatchVC are dubbed GreedyVC+ and MatchVC+ respectively.

Table 5: Comparing EdgeGreedyVC, GreedyVC+ and MatchVC+, where '#win.' counts the number of winning instances and accumulated time includes the run time on all 102 instances.

	GreedyVC+	EdgeGreedyVC	MatchVC+
#win.	70	25	7
Accumulated Time	146.14	6.26	6.35

We conduct experiments to compare EdgeGreedyVC, GreedyVC+ and MatchVC+. Since EdgeGreedyVC and MatchVC+ are deterministic algorithms, they are executed once (with random seed 1) on each instance. GreedyVC+ uses a randomized strategy to break ties, so it is executed 10 times (with random seeds from 1 to 10) on each instance and the averaged results over the 10 runs are reported. Interestingly, our experiments show that these three construction MinVC algorithms have superiority on different instances (Table 5), where an algorithm wins an instance if it is the only one that gives the best solution or it has the least run time among the algorithms with the best quality solution. Therefore, a natural idea for improving local search MinVC algorithms is to integrate the three construction algorithms by using them to generate three solutions and take the best one as the initial solution for local search. We use this engineering practice to improve both NuMVC and FastVC, resulting in two local search MinVC solvers named NuMVC2 and FastVC2. They are also tested on the whole benchmark suite and the results are reported in Tables 6 and 7, which show that FastVC2 outperforms NuMVC2 on most instances. Since the initial solution is the same, this indicates the effectiveness of the local search procedure in FastVC for solving massive graphs.

Additionally, comparing the results in Tables 6 and 7 with those in Tables 3 and 4, we observe that NuMVC2 has better performance overall than NuMVC and NuMVC$_e$, and FastVC2 is better than FastVC. Note that these algorithms use the same random seeds (1,2,...,10) in our experiments. With respect to solution quality, NuMVC2 performs better than NuMVC on 31 instances while worse on 19 instances, and better than NuMVC$_e$ on 28 instances while worse on 21 instances. FastVC2 performs better than FastVC on 26 instances while worse on 17 instances. We also test different variants of NuMVC and FastVC with only one construction heuristic from EdgeGreedyVC, GreedyVC+ and MatchVC+, and the results show that NuMVC2 and FastVC2 have overall better performance w.r.t. solution quality than their variants with a single construction heuristic.

6. Improving MinVC Solving by Preprocessing Techniques

In this section, we develop a preprocessing algorithm to simplify graphs for MinVC algorithms. The preprocessing algorithm works in two phases and uses four reduction rules. We conduct experiments to study the rules and show their effectiveness. The preprocessing algorithm is used to improve NuMVC2 and FastVC2 from the previous section, resulting in two MinVC solvers named NuMVC2+p and FastVC2+p. Experiments show that the preprocessing algorithm can improve the solution quality for a considerable portion of the tested graphs.

6.1 Reduction Rules and the Preprocessing Algorithm

Our preprocessing algorithm for MinVC is based on four reduction rules. The first three rules are very simple and have been widely used (Chen, Kanj, & Jia, 2001; Abu-Khzam, Collins, Fellows,

Table 6: Experiment results of NuMVC2 and FastVC2. Both algorithms use EdgeGreedyVC, GreedyVC+ and MatchVC+ to generate three solutions and take the best one as the initial solution for local search.

instance	NuMVC2		FastVC2	
	min(avg)	time	min(avg)	time
bio-celegans	249(249)	<0.01	249(249)	<0.01
bio-diseasome	285(285)	<0.01	285(285)	<0.01
bio-dmela	**2630(2630)**	1.29	2632(2632)	0.01
bio-yeast	456(456)	<0.01	456(456)	<0.01
ca-AstroPh	11483(11483)	63.2	11483(11483)	**0.06**
ca-citeseer	129193(129193.6)	60.63	129193(**129193**)	1.36
ca-coauthors-dblp	472180(472181.8)	905.45	**472179(472179)**	16.91
ca-CondMat	12480(12480)	39.03	12480(12480)	**0.04**
ca-CSphd	550(550)	<0.01	550(550)	<0.01
ca-dblp-2010	121972(121972.7)	60.82	**121969(121969)**	1.78
ca-dblp-2012	164951(164953.9)	92.56	**164949(164949)**	4.76
ca-Erdos992	461(461)	<0.01	461(461)	<0.01
ca-GrQc	2208(2208)	<0.01	2208(2208)	<0.01
ca-HepPh	6555(6555)	6.48	6555(6555)	**0.03**
ca-hollywood-2009	864116(864117.2)	973.41	**864052(864052)**	33.17
ca-MathSciNet	139982(139985.2)	45.07	**139951(139951)**	4.54
ca-netscience	214(214)	<0.01	214(214)	<0.01
ia-email-EU	820(820)	<0.01	820(820)	<0.01
ia-email-univ	594(594)	<0.01	594(594)	<0.01
ia-enron-large	12781(12781)	74.25	12781(12781)	**0.11**
ia-enron-only	86(86)	<0.01	86(86)	<0.01
ia-fb-messages	578(578)	<0.01	578(578)	<0.01
ia-infect-dublin	293(293)	<0.01	293(293)	<0.01
ia-infect-hyper	90(90)	<0.01	90(90)	<0.01
ia-reality	81(81)	<0.01	81(81)	<0.01
ia-wiki-Talk	17288(17288.3)	523.41	17288(**17288**)	0.14
inf-power	2203(2203)	1.7	2203(2203)	<**0.01**
inf-roadNet-CA	1038328(1038553)	1000	**1001442(1001474.1)**	893.82
inf-roadNet-PA	560133(560173.4)	936.31	**555269(555324.6)**	700.32
inf-road-usa	12082301(12082356.4)	1000	**11809276(11828598)**	1000
rec-amazon	47671(47682.2)	923.45	**47606(47606)**	1.12
rt-retweet	32(32)	<0.01	32(32)	<0.01
rt-retweet-crawl	81044(81045.3)	2.85	**81040(81040)**	1.89
rt-twitter-copen	237(237)	<0.01	237(237)	<0.01
socfb-A-anon	375232(375233.4)	39.19	**375230(375230.9)**	23.7
socfb-B-anon	303048(303049.5)	26.39	303048(**303048**)	7.15
socfb-Berkeley13	17213(17216.8)	579.99	**17210(17211.2)**	345.5
socfb-CMU	**4986(4986.1)**	193.77	4987(4987)	3.49
socfb-Duke14	7683(7683.5)	115.93	7683(**7683**)	361.41
socfb-Indiana	23323(23328.8)	503.05	**23315(23317.1)**	407.46
socfb-MIT	4657(4657)	**7.97**	4657(4657)	21.47
socfb-OR	36559(36562.6)	663.33	**36548(36548.7)**	288.97
socfb-Penn94	31177(31185.1)	484.42	**31163(31165.2)**	600.93
socfb-Stanford3	8518(8518)	**10.74**	8518(8518)	25.22
socfb-Texas84	28180(28184.3)	613.51	**28165(28168.7)**	629.11
socfb-uci-uni	866766(866766)	127.19	866766(866766)	**26.07**
socfb-UCLA	15225(15227.6)	656.43	**15224(15225.3)**	355.59
socfb-UConn	13231(13233.6)	589.34	**13230(13230.9)**	352.79
socfb-UCSB37	11262(11263)	546.05	**11261(11262.5)**	269.29
socfb-UF	27322(27325.3)	544.41	**27306(27308.6)**	348.77
socfb-UIllinois	24103(24108.2)	561.88	**24093(24095.2)**	373.81
socfb-Wisconsin87	18389(18392.1)	636.37	**18383(18385)**	569.64

Table 7: Experiment results of NuMVC2 and FastVC2 (continued).

instance	NuMVC2		FastVC2	
	min(avg)	time	min(avg)	time
sc-ldoor	856936(856949)	998	**856755(856757.4)**	221.24
sc-msdoor	381565(381567.1)	958.01	**381558(381558.9)**	18.38
sc-nasasrb	51244(51246.9)	768.2	**51240(51243.2)**	475.53
sc-pkustk11	**83911(83911)**	591.98	83912(83913.2)	12.03
sc-pkustk13	89220(89222.5)	938.54	**89218(89220.3)**	392.48
sc-pwtk	207741(207746.1)	154.63	**207716(207719.9)**	208.31
sc-shipsec1	117482(117518.4)	991.04	**117281(117297.5)**	742.95
sc-shipsec5	147292(147314.2)	985.78	**147067(147077.4)**	548.72
soc-BlogCatalog	20752(20752.2)	424.94	20752(**20752**)	0.45
soc-brightkite	21191(21192.9)	859.39	**21190(21190.1)**	0.24
soc-buzznet	**30613(30613.7)**	580.13	30624(30624)	6.27
soc-delicious	85566(85573.2)	659.65	**85532(85536.1)**	1.69
soc-digg	103312(103321.7)	566.02	**103242(103242)**	2.81
soc-dolphins	34(34)	<0.01	34(34)	<0.01
soc-douban	8685(8685)	0.04	8685(8685)	0.04
soc-epinions	9757(9757)	113.99	9757(9757)	**0.17**
soc-flickr	153343(153349.3)	38.58	**153271(153271.1)**	18.33
soc-flixster	96318(96319.2)	3.14	**96317(96317)**	2.72
soc-FourSquare	90131(90136.6)	882.48	**90109(90109.7)**	78.47
soc-gowalla	84312(84321.4)	961.51	**84223(84223)**	26.96
soc-karate	14(14)	<0.01	14(14)	<0.01
soc-lastfm	78692(78695.1)	591.63	**78688(78688)**	1.14
soc-livejournal	1882773(1882841.4)	1000	**1869002(1869007.6)**	934.25
soc-LiveMocha	43430(43432.6)	786.35	**43427(43427)**	29.86
soc-orkut	2193141(2193277.4)	1000	**2171246(2171290.2)**	997.16
soc-pokec	844306(844329.2)	610.7	**843387(843390.4)**	641.83
soc-slashdot	22373(22375.3)	850.36	22373(**22373**)	0.29
soc-twitter-follows	2323(2323)	0.22	2323(2323)	**0.17**
soc-wiki-Vote	406(406)	<0.01	406(406)	<0.01
soc-youtube	146454(146461.6)	47.1	**146377(146377)**	5.27
soc-youtube-snap	277012(277017.4)	209.33	**276947(276947)**	14.49
tech-as-caida2007	3683(3683)	0.02	3683(3683)	<**0.01**
tech-as-skitter	525950(525962.1)	570.23	**525538(525542.7)**	599.46
tech-internet-as	5700(5700)	19.47	5700(5700)	**0.04**
tech-p2p-gnutella	15682(15682)	30.64	15682(15682)	**0.03**
tech-RL-caida	74763(74772.7)	936.24	**74710(74713.1)**	9.28
tech-routers-rf	795(795)	<0.01	795(795)	<0.01
tech-WHOIS	2284(2284)	0.09	2284(2284)	<**0.01**
web-arabic-2005	114454(114458.6)	65.27	**114426(114427.3)**	254.95
web-BerkStan	5384(5384)	**9.53**	5384(5384)	23.13
web-edu	1451(1451)	<0.01	1451(1451)	<0.01
web-google	498(498)	<0.01	498(498)	<0.01
web-indochina-2004	7300(7300)	14.03	7300(7300)	**0.12**
web-it-2004	414705(414720.7)	643.76	**414688(414689.9)**	315
web-polblogs	244(244)	<0.01	244(244)	<0.01
web-sk-2005	58206(58209.8)	655.21	**58173(58173)**	9.71
web-spam	2297(2297)	0.53	2297(2297)	**0.01**
web-uk-2005	127774(127774)	1.21	127774(127774)	**1.05**
web-webbase-2001	2652(2652)	1.39	2652(2652)	**0.02**
web-wikipedia2009	649261(649310.7)	999.16	**648318(648325.1)**	667.81

Langston, Suters, & Symons, 2004). The fourth rule called Dominance Rule was introduced recently (Fomin, Grandoni, & Kratsch, 2009). When using the reduction rules to simplify a graph, some vertices are fixed as covering vertices, if there must be an optimal vertex cover containing them. We use G to denote the graph we are dealing with.

- **Degree-0 Rule**: An isolated vertex u (vertex of degree 0) cannot be in a vertex cover of optimal size. Thus, the graph G can be simplified by deleting u.
- **Degree-1 Rule**: If a vertex u is a pendant vertex (vertex of degree 1), there is a vertex cover of optimal size that does not contain the pendant vertex but does contain its unique neighbor v. Thus, vertex v is fixed as a covering vertex, and G is simplified by deleting both u and v and their incident edges.
- **Degree-2 Rule**: If there is a degree-two vertex u with adjacent neighbors v and z, then there is a vertex cover of optimal size that does not contain u but does include both of these neighbors. Thus, vertex v and z are fixed as covering vertices, and G is simplified by deleting u, v, z and their incident edges.
- **Dominance Rule**: For two vertices u and v, if $N[u] \subseteq N[v]$, we say vertex v dominates vertex u. If a vertex v dominates some vertex, there always exists a vertex cover of optimal size that contains v. Therefore, vertex v is fixed as a covering vertex, and G is simplified by deleting v and its incident edges.

Note that Degree-1 and Degree-2 rules can be implied by the Dominance rule. However, these two rules can be executed much faster than the Dominance rule. Based on this observation, we develop a two-phase preprocessing algorithm for MinVC, which is called D1+D2+Dom. To implement the preprocessing algorithm, we use a queue, denoted as Q, to store the vertices to be processed by reduction rules, and a set F to store the vertices fixed as covering vertices. The preprocessing algorithm consists of two phases, and in each phase it works in an iterative way. In the first phase, it simplifies the graph only using Degree-1 and Degree-2 rules (and Degree-0 rule). When the graph cannot be simplified anymore by only using Degree-1 and Degree-2 rules, the preprocessing algorithm enters the second phase, where the Dominance rule is used to simplify the graph iteratively until no more vertex can be deleted according to the rule.

The preprocessing algorithm returns a simplified graph and a set F which includes all fixed covering vertices. Then, the simplified graph is taken as the input graph for a MinVC algorithm. Suppose the MinVC algorithm returns a vertex cover C' for the simplified graph, then $C^* = C' \cup F$ is a vertex cover for the original graph and is returned as the solution found. Additionally, if after preprocessing, a graph becomes empty (i.e., $V(G) = \emptyset$), then the set F is indeed a minimum vertex cover for the original graph.

6.2 Effectiveness of Reduction Rules

To examine the effects of the reduction rules, we implement and test three preprocessing algorithms, including D1, D1+D2, and D1+D2+Dom, which use different sets of reduction rules. As their names indicate, D1 only uses the Degree-1 rule, and D1+D2 uses both Degree-1 and Degree-2 rules, while D1+D2+Dom employs all the rules. Note that the trivial Degree-0 rule is executed in all the preprocessing algorithms.

The experiment results are listed in Tables 8 and 9. We can observe the significant effects of these reduction rules on the size of the graphs. Comparing the graph size after reduction, we can see that D1+D2+Dom produces smaller graphs than D1+D2, which in turn is better than D1. Regarding the run time, D1 and D1+D2 usually terminate within one second. For D1+D2+Dom, the run time is much longer for some instances. There are 3 instances for which D1+D2+Dom terminates in

Table 8: Effectiveness study of reduction rules.

instance	$\lvert V\rvert$	$\lvert E\rvert$	D1			D1+D2			D1+D2+Dom		
			$\lvert V'\rvert$	$\lvert E'\rvert$	time	$\lvert V'\rvert$	$\lvert E'\rvert$	time	$\lvert V'\rvert$	$\lvert E'\rvert$	time
bio-celegans	453	2025	441	1940	<0.01	32	64	<0.01	0	0	<0.01
bio-diseasome	516	1188	242	483	<0.01	96	201	<0.01	0	0	<0.01
bio-dmela	7393	25569	791	1393	<0.01	750	1295	<0.01	748	1285	<0.01
bio-yeast	1458	1948	59	79	<0.01	15	24	<0.01	8	8	<0.01
ca-AstroPh	17903	196972	14669	158845	0.01	7651	66272	0.01	0	0	0.01
ca-citeseer	227320	814134	124478	447088	0.05	57470	262641	0.07	16	16	0.08
ca-coauthors-dblp	540486	15245729	521385	14978514	0.07	495944	14555260	0.15	24	24	2.79
ca-CondMat	21363	91286	14338	57252	<0.01	5842	21081	0.01	0	0	<0.01
ca-CSphd	1882	1740	4	4	<0.01	4	4	<0.01	4	4	<0.01
ca-dblp-2010	226413	716460	109343	317705	0.06	44355	156299	0.06	8	8	0.06
ca-dblp-2012	317080	1049866	143132	377759	0.13	49573	147140	0.13	29	29	0.12
ca-Erdos992	6100	7515	0	0	<0.01	0	0	<0.01	0	0	<0.01
ca-GrQc	4158	13422	1783	6600	<0.01	574	3494	<0.01	0	0	<0.01
ca-HepPh	11204	117619	6740	67573	0.01	2589	30501	0.01	4	4	<0.01
ca-hollywood-2009	1069126	56306653	1042751	52438424	2.45	965676	40863964	9.19	13	13	35.73
ca-MathSciNet	332689	820644	55851	92336	0.12	11678	24654	0.11	59	61	0.13
ca-netscience	379	914	242	518	<0.01	86	195	<0.01	0	0	<0.01
ia-email-EU	32430	54397	3	3	0.01	0	0	<0.01	0	0	<0.01
ia-email-univ	1133	5451	620	2206	<0.01	382	1073	<0.01	178	374	<0.01
ia-enron-large	33696	180811	13214	36771	0.01	3355	7207	0.01	21	22	<0.01
ia-enron-only	143	623	119	529	<0.01	94	371	<0.01	50	112	<0.01
ia-fb-messages	1266	6451	227	391	<0.01	64	81	<0.01	61	74	<0.01
ia-infect-dublin	410	2765	374	2497	<0.01	349	2340	<0.01	186	927	<0.01
ia-infect-hyper	113	2196	111	2098	<0.01	111	2098	<0.01	108	1883	<0.01
ia-reality	6809	7680	0	0	<0.01	0	0	<0.01	0	0	<0.01
ia-wiki-Talk	92117	360767	101	106	0.03	12	12	0.03	12	12	0.03
inf-power	4941	6594	458	614	<0.01	175	214	<0.01	155	179	<0.01
inf-roadNet-CA	1957027	2760388	1062587	1571668	0.32	913469	1350136	0.34	907378	1335179	0.44
inf-roadNet-PA	1087562	1541514	561133	839083	0.17	470492	704445	0.2	467671	697305	0.26
inf-road-usa	23947347	28854312	5310709	7258703	6.13	4462295	6173990	6.72	4454350	6156614	8.02
rec-amazon	91813	125704	33657	49524	0.02	6886	11038	0.01	1650	1789	0.01
rt-retweet	96	117	0	0	<0.01	0	0	<0.01	0	0	<0.01
rt-retweet-crawl	1112702	2278852	517	1248	0.43	472	1178	0.43	415	693	0.43
rt-twitter-copen	761	1029	14	15	<0.01	0	0	<0.01	0	0	<0.01
socfb-A-anon	3097165	23667394	1032	1660	5.45	276	659	4.34	127	174	4.51
socfb-B-anon	2937612	20959854	479	733	4.28	130	204	3.94	81	91	3.77
socfb-Berkeley13	22900	852419	21474	778596	0.04	20771	735629	0.05	19561	645318	0.18
socfb-CMU	6621	249959	6182	229918	0.01	5941	217267	0.01	5386	173161	0.04
socfb-Duke14	9885	506437	9328	475087	0.02	9010	452633	0.02	8403	399227	0.1
socfb-Indiana	29732	1305757	28574	1252996	0.04	27928	1214330	0.06	26675	1096518	0.27
socfb-MIT	6402	251230	5808	225360	0.01	5513	207675	0.01	4860	155166	0.04
socfb-OR	63392	816886	42756	532334	0.05	34187	401175	0.07	25782	237632	0.16
socfb-Penn94	41536	1362220	39255	1267464	0.06	37992	1192146	0.08	35684	1042042	0.3
socfb-Stanford3	11586	568309	10421	506275	0.03	9846	473219	0.05	8995	399622	0.13
socfb-Texas84	36364	1590651	34932	1494828	0.06	34291	1449994	0.07	32661	1286733	0.33
socfb-uci-uni	58790782	92208195	147	273	18.67	118	224	18.98	67	77	19.32
socfb-UCLA	20453	747604	19062	689871	0.03	18404	656663	0.04	17251	567198	0.13
socfb-UConn	17206	604867	16444	571019	0.01	16080	551369	0.02	15270	491637	0.11
socfb-UCSB37	14917	482215	14127	452899	0.02	13748	436144	0.02	13020	389722	0.08
socfb-UF	35111	1465654	33847	1387729	0.05	33169	1341208	0.07	31385	1205668	0.32
socfb-UIllinois	30795	1264421	29530	1204844	0.04	28866	1163750	0.06	27428	1056152	0.26
socfb-Wisconsin87	23831	835946	22764	794760	0.03	22212	762643	0.04	20966	675559	0.19

more than 10 seconds, and the longest run time reaches 35 seconds. Nevertheless, this is acceptable

Table 9: Effectiveness study of reduction rules (continued).

instance	|V|	|E|	D1			D1+D2			D1+D2+Dom		
			|V'|	|E'|	time	|V'|	|E'|	time	|V'|	|E'|	time
sc-ldoor	952203	20770807	909537	20770807	0.1	909537	20770807	0.09	20301	34702	2.02
sc-msdoor	415863	9378650	404785	9378650	0.03	404785	9378650	0.03	43041	120519	0.88
sc-nasasrb	54870	1311227	54870	1311227	<0.01	54870	1311227	<0.01	15425	82814	0.12
sc-pkustk11	87804	2565054	87804	2565054	<0.01	87804	2565054	<0.01	3814	4362	0.24
sc-pkustk13	94893	3260967	94893	3260967	<0.01	94893	3260967	<0.01	14711	47933	0.36
sc-pwtk	217891	5653221	217847	5653055	<0.01	217847	5653055	<0.01	38430	164598	0.48
sc-shipsec1	140385	1707759	140385	1707759	<0.01	140376	1707739	<0.01	111692	1060673	0.14
sc-shipsec5	179104	2200076	179092	2200067	<0.01	179089	2200064	<0.01	134066	1252861	0.18
soc-BlogCatalog	88784	2093195	483	807	0.39	153	362	0.39	47	62	0.37
soc-brightkite	56739	212945	3020	6654	0.02	511	2497	0.01	120	159	0.02
soc-buzznet	101163	2763066	11375	37250	0.44	3169	8655	0.42	1881	4465	0.42
soc-delicious	536108	1365961	18690	69612	0.16	18499	69131	0.14	18367	68559	0.19
soc-digg	770799	5907132	2763	6951	1.5	848	3754	1.46	402	532	1.43
soc-dolphins	62	159	40	82	<0.01	24	41	<0.01	18	21	<0.01
soc-douban	154908	327162	17	31	0.03	17	31	0.02	16	28	0.03
soc-epinions	26588	100120	1580	3334	0.01	221	538	<0.01	60	68	<0.01
soc-flickr	513969	3190452	31972	342883	0.67	8946	54497	0.56	3921	18367	0.57
soc-flixster	2523386	7918801	137	196	1.03	23	35	0.95	8	8	1.21
soc-FourSquare	639014	3214986	1836	2644	1.97	387	680	1.42	138	149	1.34
soc-gowalla	196591	950327	31780	78356	0.11	4563	15489	0.09	1176	4887	0.11
soc-karate	34	78	9	10	<0.01	4	4	<0.01	4	4	<0.01
soc-lastfm	1191805	4519330	167	232	0.71	32	45	0.68	17	18	0.71
soc-livejournal	4033137	27933062	1220365	5202120	5.02	213667	1460786	4.87	41637	74057	5.07
soc-LiveMocha	104103	2193083	655	732	0.34	24	24	0.36	24	24	0.35
soc-orkut	2997166	106349209	2829189	95788825	3.83	2754327	91669427	5.38	2599169	82644648	34.28
soc-pokec	1632803	22301964	1090351	12491982	2.46	963857	10265224	3.23	881359	8572011	8.06
soc-slashdot	70068	358647	903	1167	0.03	112	214	0.04	26	27	0.04
soc-twitter-follows	404719	713319	0	0	0.08	0	0	0.07	0	0	0.08
soc-wiki-Vote	889	2914	121	187	<0.01	24	29	<0.01	24	29	<0.01
soc-youtube	495957	1936748	8156	10894	0.31	1464	2553	0.28	535	610	0.28
soc-youtube-snap	1134890	2987624	12035	15014	0.59	1584	2680	0.48	584	653	0.5
tech-as-caida2007	26475	53381	40	50	<0.01	15	18	<0.01	15	18	<0.01
tech-as-skitter	1694616	11094209	475124	1600224	1.56	262723	831476	1.28	228382	619077	1.36
tech-internet-as	40164	85123	102	131	<0.01	15	18	<0.01	15	18	<0.01
tech-p2p-gnutella	62561	147878	38	54	0.01	38	54	0.01	38	54	0.01
tech-RL-caida	190914	607610	70226	211590	0.06	51553	139142	0.07	47149	119062	0.1
tech-routers-rf	2113	6632	231	685	<0.01	14	22	<0.01	4	4	<0.01
tech-WHOIS	7476	56943	665	3409	<0.01	294	1611	<0.01	63	122	<0.01
web-arabic-2005	163598	1747269	102515	1560020	0.05	98504	1542669	0.05	1994	3514	0.1
web-BerkStan	12305	19500	5121	8345	<0.01	1105	1589	<0.01	771	857	<0.01
web-edu	3031	6474	149	689	<0.01	91	582	<0.01	0	0	<0.01
web-google	1299	2773	326	1140	<0.01	218	903	<0.01	0	0	<0.01
web-indochina-2004	11358	47606	6297	32421	<0.01	5355	25973	<0.01	12	12	<0.01
web-it-2004	509338	7178413	424893	6440816	0.16	423389	6428764	0.16	4995	21441	1.17
web-polblogs	643	2280	48	92	<0.01	16	23	<0.01	4	4	<0.01
web-sk-2005	121422	334419	42237	225932	0.02	39152	218233	0.02	27775	158337	0.03
web-spam	4767	37375	1971	8540	0.01	874	2397	0.01	214	312	<0.01
web-uk-2005	129632	11744049	127716	11643622	0.04	127713	11643619	0.04	0	0	2.55
web-webbase-2001	16062	25593	2085	5203	<0.01	1913	4870	<0.01	1391	2458	<0.01
web-wikipedia2009	1864433	4507315	154344	302990	0.74	37561	98257	0.68	18941	47389	0.96

for such an NP-hard problem. Additionally, 15 graphs become empty after being simplified by D1+D2+Dom, which means D1+D2+Dom alone solves these instances exactly.

We also study how often a reduction rule is executed in the D1+D2+Dom preprocessing algorithm. For each instance class, we calculate the averaged percentage of each rule over all rules,

Table 10: The percentage of executions for each reduction rule, which equals the number of executions of the rule divided by the number of executions of all rules.

instance class	D1	D2	Dom
bio	72.25%	16.40%	11.35%
ca	41.02%	18.89%	40.09%
ia	60.00%	9.31%	30.69%
inf	94.25%	5.15%	0.60%
rec	70.39%	20.31%	9.30%
rt	97.95%	2.04%	0.01%
socfb	43.16%	10.41%	46.43%
sc	<0.01%	<0.01%	>99.99%
soc	84.06%	10.68%	5.26%
tech	89.27%	7.99%	2.74%
web	51.65%	8.83%	39.52%

and the results are reported in Table 10. From these results, we have the following observations: For different instance classes (which can be regarded as graphs of different structures), the figures are considerably different. An extreme example is that the percentage of the Dominance rule is less than 1% for `inf` and `rt` instance classes, while it is almost 100% for the `sc` class. The Degree-1 rule usually occupies a large percentage over all rules, except one class. For 8 out of the total 11 classes, the Degree-1 rule's executions occupy more than a half of all executions of all rules. Overall, Degree-2 is the least often executed rule, compared to the Degree-1 rule and the Dominance rule. To some extent, we can consider it is less useful compared to the other two rules. For some instances such as the `sc` instances (one can also refer to the results on `sc` instances in Table 9), the Degree-1 and Degree-2 rules are almost useless, while the Dominance is more powerful and can simplify the graphs.

6.3 Connected Components in Simplified Graphs

In this subsection, we study the structure of the simplified graphs produced by the preprocessing algorithm. Specifically, we investigate the connected components in the simplified graph. The results are presented in Tables 11, where we report the number of connected components ('#comp.') and the size of the largest connected component ('$|V^*|$' and '$|E^*|$'). We do not report the results for the small graphs with fewer than 2000 vertices, as well as the graphs reduced to become empty by D1+D2+Dom.

Seen from the results, many graphs are decomposed into small connected components after preprocessing. For 58% of the graphs, the largest connected component has fewer than 2000 vertices, and thus can be solved by iteratively calling exact solvers on each component. Nevertheless, for the remaining 42% of the graphs, the largest connected component is still large and usually beyond the reach of exact solvers.

6.4 Comparison with State of the Art Preprocessing Algorithm

In recent years, reduction techniques for vertex cover and related problems have attracted more and more interest and have shown their power in practice (Akiba & Iwata, 2016; Lamm, Sanders, Schulz, Strash, & Werneck, 2016; Strash, 2016; Verma, Buchanan, & Butenko, 2015; Cai & Lin, 2016). Very recently, Strash developed a preprocessing algorithm named `Simple` for the maximum

Table 11: Connected components in simplified graphs by D1+D2+Dom.

instance	#comp.	largest component	
		$\|V^*\|$	$\|E^*\|$
bio-dmela	15	686	1217
ca-citeseer	4	4	4
ca-coauthors-dblp	6	4	4
ca-dblp-2010	2	4	4
ca-dblp-2012	7	5	5
ca-HepPh	1	4	4
ca-hollywood-2009	3	5	5
ca-MathSciNet	14	6	7
ia-enron-large	4	8	9
ia-wiki-Talk	3	4	4
inf-power	18	19	28
inf-roadNet-CA	4047	384123	577173
inf-roadNet-PA	2129	234989	359992
inf-road-usa	89584	58524	93033
rec-amazon	316	25	32
rt-retweet-crawl	48	62	120
socfb-A-anon	26	15	45
socfb-B-anon	18	8	13
socfb-Berkeley13	1	19561	645318
socfb-CMU	1	5386	173161
socfb-Duke14	1	8403	399227
socfb-Indiana	1	26675	1096518
socfb-MIT	1	4860	155166
socfb-OR	2	25778	237628
socfb-Penn94	1	35684	1042042
socfb-Stanford3	2	8991	399618
socfb-Texas84	1	32661	1286733
socfb-uci-uni	14	8	12
socfb-UCLA	1	17251	567198
socfb-UConn	1	15270	491637
socfb-UCSB37	1	13020	389722
socfb-UF	1	31385	1205668
socfb-UIllinois	1	27428	1056152
socfb-Wisconsin87	1	20966	675559
sc-ldoor	105	4902	7109
sc-msdoor	6	35057	100094
sc-nasasrb	8	14832	81654
sc-pkustk11	36	196	388

instance	#comp.	largest component	
		$\|V^*\|$	$\|E^*\|$
sc-pkustk13	1	14711	47933
sc-pwtk	76	36336	162504
sc-shipsec1	3	111515	1059903
sc-shipsec5	30	133624	1251679
soc-BlogCatalog	7	15	22
soc-brightkite	16	22	51
soc-buzznet	6	1849	4386
soc-delicious	754	1914	8039
soc-digg	71	29	44
soc-douban	1	16	28
soc-epinions	11	17	25
soc-flickr	147	3043	17179
soc-flixster	2	4	4
soc-FourSquare	30	8	9
soc-gowalla	79	777	4449
soc-lastfm	4	5	6
soc-livejournal	3536	16517	28617
soc-LiveMocha	6	4	4
soc-orkut	5	2599153	82644632
soc-pokec	10	881319	8571967
soc-slashdot	6	6	7
soc-youtube	123	22	82
soc-youtube-snap	131	17	50
tech-as-caida2007	2	9	12
tech-as-skitter	11158	108182	367812
tech-internet-as	3	7	10
tech-p2p-gnutella	5	12	20
tech-RL-caida	1663	34528	98587
tech-routers-rf	1	4	4
tech-WHOIS	6	26	61
web-arabic-2005	84	483	855
web-BerkStan	83	306	365
web-indochina-2004	3	4	4
web-it-2004	128	156	635
web-sk-2005	469	877	4569
web-spam	4	199	295
web-webbase-2001	5	1365	2422
web-wikipedia2009	1130	2289	5871

independent set (MaxIS) problem, which consists only of a small suite of simple reductions, which, however, are very effective (Strash, 2016). In this subsection, we compare our preprocessing algorithm D1+D2+Dom for MinVC with Strash's preprocessing algorithm dubbed Simple for MaxIS, by comparing the size of the simplified graph. Bear in mind that the simplified graphs produced by `Simple` should be solved by a MaxIS algorithm, since its reduction rules are designed for MaxIS.

The comparison results are summarized in Table 12. The instances that are reduced to empty by both algorithms are not listed in the table. Overall, the two preprocessing algorithms have similar performance and are complementary to each other on these real-world instances. Specifically, our preprocessing algorithm is more effective on Facebook networks and scientific computation networks, while the MaxIS preprocessing algorithm Simple is more effective on infrastructure

Table 12: Comparing D1+D2+Dom with the MaxIS preprocessing algorithm `Simple`.

instance	Simple			D1+D2+Dom		
	|V'|	|E'|	time	|V'|	|E'|	time
bio-dmela	**7**	**12**	0.01	748	1285	<0.01
bio-yeast	**0**	**0**	<0.01	8	8	<0.01
ca-citeseer	**5**	**8**	0.12	16	16	0.08
ca-authors-dblp	27	67	1.25	**24**	**24**	2.79
ca-CSphd	**0**	**0**	<0.01	4	4	<0.01
ca-dblp-2010	**0**	**0**	0.09	8	8	0.06
ca-dblp-2012	**13**	**24**	0.21	29	29	0.12
ca-HepPh	7	17	0.01	**4**	**4**	<0.01
ca-hollywood-09	**9**	28	15.37	13	**13**	35.73
ca-MathSciNet	**0**	**0**	0.21	59	61	0.13
ia-email-univ	**101**	**296**	<0.01	178	374	<0.01
ia-enron-large	**6**	**10**	0.02	21	22	<0.01
ia-enron-only	78	285	<0.01	**50**	**112**	<0.01
ia-fb-messages	**0**	**0**	<0.01	61	74	<0.01
ia-infect-dublin	263	1878	<0.01	**186**	**927**	<0.01
ia-infect-hyper	111	2098	<0.01	**108**	**1883**	<0.01
ia-wiki-Talk	**0**	**0**	0.06	12	12	0.03
inf-power	**0**	**0**	<0.01	155	179	<0.01
inf-roadNet-CA	**305166**	**539735**	0.58	907378	1335179	0.44
inf-roadNet-PA	**169050**	**298800**	0.32	467671	697305	0.26
inf-road-usa	**907352**	**1610582**	11.32	4454350	6156614	8.02
rec-amazon	**489**	**869**	0.03	1650	1789	0.01
rt-retweet-crawl	**60**	**123**	0.99	415	693	0.43
socfb-A-anon	**27**	**109**	7.35	127	174	4.51
socfb-B-anon	**5**	**8**	5.74	81	91	3.77
socfb-Berkeley13	19706	699207	0.07	**19561**	**645318**	0.18
socfb-CMU	5544	198448	0.01	**5386**	**173161**	0.04
socfb-Duke14	8525	429456	0.02	**8403**	**399227**	0.1
socfb-Indiana	26926	1169133	0.06	**26675**	**1096518**	0.27
socfb-MIT	5103	186191	0.01	**4860**	**155166**	0.04
socfb-OR	**24572**	305775	0.17	25782	**237632**	0.16
socfb-Penn94	36200	1136060	0.09	**35684**	**1042042**	0.3
socfb-Stanford3	9126	445324	0.04	**8995**	**399622**	0.13
socfb-Texas84	32995	1372652	0.09	**32661**	**1286733**	0.33
socfb-uci-uni	**0**	**0**	32.83	67	77	19.32
socfb-UCLA	17327	616456	0.04	**17251**	**567198**	0.13
socfb-UConn	15435	527538	0.03	**15270**	**491637**	0.11
socfb-UCSB37	13074	413812	0.02	**13020**	**389722**	0.08
socfb-UF	31872	1285610	0.09	**31385**	**1205668**	0.32
socfb-UIllinois	27786	1121816	0.05	**27428**	**1056152**	0.26
socfb-Wiscsin87	21138	720010	0.05	**20966**	**675559**	0.19
sc-ldoor	909537	20770807	0.14	**20301**	**34702**	2.02
sc-msdoor	404785	9378650	0.06	**43041**	**120519**	0.88
sc-nasasrb	54486	1305519	0.01	**15425**	**82814**	0.12
sc-pkustk11	55452	1541478	0.07	**3814**	**4362**	0.24
sc-pkustk13	69918	2228836	0.06	**14711**	**47933**	0.36
sc-pwtk	217416	5647897	0.05	**38430**	**164598**	0.48
sc-shipsec1	139660	1702022	0.01	**111692**	**1060673**	0.14
sc-shipsec5	175205	2155554	0.02	**134066**	**1252861**	0.18
soc-BlogCatalog	**6**	**12**	0.71	47	62	0.37
soc-brightkite	**24**	**66**	0.04	120	159	0.02
soc-buzznet	**237**	**1920**	5.39	1881	4465	0.42
soc-delicious	**11447**	**51092**	0.26	18367	68559	0.19
soc-digg	**163**	2272	2.68	402	**532**	1.43
soc-dolphins	**0**	**0**	<0.01	18	21	<0.01
soc-douban	**0**	**0**	0.07	16	28	0.03
soc-epinions	**41**	131	0.01	60	**68**	<0.01
soc-flickr	**1958**	**11948**	1.46	3921	18367	0.57
soc-flixster	**0**	**0**	1.57	8	8	1.21
soc-FourSquare	**29**	**81**	8.3	138	149	1.34
soc-gowalla	**765**	5556	0.62	1176	**4887**	0.11
soc-karate	**0**	**0**	<0.01	4	4	<0.01
soc-lastfm	**0**	**0**	0.82	17	18	0.71
soc-livejnal	**28694**	**471112**	11.97	41637	74057	5.07
soc-LiveMocha	**0**	**0**	3.26	24	24	0.35
soc-orkut	2623684	87854637	19.89	**2599169**	**82644648**	34.28
soc-pokec	**748755**	9207514	11.72	881359	**8572011**	8.06
soc-slashdot	**0**	**0**	0.1	26	27	0.04
soc-wiki-Vote	**0**	**0**	<0.01	24	29	<0.01
soc-youtube	**159**	**366**	0.89	535	610	0.28
soc-youtube-snap	**76**	**212**	1.68	584	653	0.5
tech-as-caida07	**0**	**0**	0.02	15	18	<0.01
tech-as-skitter	**65334**	**270110**	9.36	228382	619077	1.36
tech-internet-as	**0**	**0**	0.02	15	18	<0.01
tech-p2p-gnutel	**0**	**0**	0.03	38	54	0.01
tech-RL-caida	**9248**	**27211**	0.23	47149	119062	0.1
tech-routers-rf	**0**	**0**	<0.01	4	4	<0.01
tech-WHOIS	91	624	0.01	**63**	**122**	<0.01
web-arabic-2005	**328**	**1216**	0.14	1994	3514	0.1
web-BerkStan	**0**	**0**	<0.01	771	857	<0.01
web-indochina-04	36	171	<0.01	**12**	**12**	<0.01
web-it-2004	**3381**	**20368**	0.38	4995	21441	1.17
web-polblogs	**0**	**0**	<0.01	4	4	<0.01
web-sk-2005	**24927**	**155059**	0.04	27775	158337	0.03
web-spam	**95**	281	0.01	214	312	<0.01
web-webbase-01	**572**	**1237**	0.01	1391	2458	<0.01
web-wikipedia09	**9266**	**37579**	2.53	18941	47389	0.96

networks, technological networks, and web linkage networks. For the remaining instances, the two preprocessing algorithms have similar performance.

6.5 The NuMVC2+p and FastVC2+p Solvers

Preprocessing techniques can improve the performance of MinVC algorithms. We integrate the preprocessor D1+D2+Dom into the NuMVC2 and FastVC2 algorithms, resulting in two MinVC solvers called NuMVC2+p and FastVC2+p. These solvers call D1+D2+Dom to simplify the input

Table 13: Experiment results of NuMVC2+p and FastVC2+p.

instance	NuMVC2+p		FastVC2+p	
	min(avg)	time	min(avg)	time
bio-celegans	249(249)	<0.01	249(249)	<0.01
bio-diseasome	285(285)	<0.01	285(285)	<0.01
bio-dmela	2630(2630)	<**0.01**	2630(2630)	1.34
bio-yeast	456(456)	<0.01	456(456)	<0.01
ca-AstroPh	11483(11483)	0.01	11483(11483)	0.01
ca-citeseer	129193(129193)	0.08	129193(129193)	0.08
ca-coauthors-dblp	472179(472179)	**2.77**	472179(472179)	2.79
ca-CondMat	12480(12480)	<0.01	12480(12480)	<0.01
ca-CSphd	550(550)	<0.01	550(550)	<0.01
ca-dblp-2010	121969(121969)	0.06	121969(121969)	0.06
ca-dblp-2012	164949(164949)	0.13	164949(164949)	**0.12**
ca-Erdos992	461(461)	<0.01	461(461)	<0.01
ca-GrQc	2208(2208)	<0.01	2208(2208)	<0.01
ca-HepPh	6555(6555)	<0.01	6555(6555)	<0.01
ca-hollywood-2009	864052(864052)	**35.58**	864052(864052)	35.73
ca-MathSciNet	139951(139951)	0.13	139951(139951)	0.13
ca-netscience	214(214)	<0.01	214(214)	<0.01
ia-email-EU	820(820)	<0.01	820(820)	<0.01
ia-email-univ	594(594)	<0.01	594(594)	<0.01
ia-enron-large	12781(12781)	<0.01	12781(12781)	<0.01
ia-enron-only	86(86)	<0.01	86(86)	<0.01
ia-fb-messages	578(578)	<0.01	578(578)	<0.01
ia-infect-dublin	293(293)	<0.01	293(293)	<0.01
ia-infect-hyper	90(90)	<0.01	90(90)	<0.01
ia-reality	81(81)	<0.01	81(81)	<0.01
ia-wiki-Talk	17288(17288)	0.03	17288(17288)	0.03
inf-power	2203(2203)	<0.01	2203(2203)	<0.01
inf-roadNet-CA	1005377(1005475.4)	615.65	**1001065(1001109.7)**	514.3
inf-roadNet-PA	557312(557366.9)	158.76	**555046(555082.9)**	250.17
inf-road-usa	11654069(11654194.8)	1000	**11527630(11527731.7)**	968.01
rec-amazon	47605(47605)	0.02	47605(47605)	**0.01**
rt-retweet	32(32)	<0.01	32(32)	<0.01
rt-retweet-crawl	81040(81040)	0.43	81040(81040)	0.43
rt-twitter-copen	237(237)	<0.01	237(237)	<0.01
socfb-A-anon	375230(375230)	4.50	375230(375230)	4.51
socfb-B-anon	303048(303048)	**3.75**	303048(303048)	3.77
socfb-Berkeley13	17214(17217.3)	742.83	**17210(17211.9)**	369.86
socfb-CMU	4986(**4986**)	130.82	4986(4986.6)	104.63
socfb-Duke14	7683(7683.3)	241.79	7683(**7683**)	201.17
socfb-Indiana	23320(23327)	645.3	**23315(23317.1)**	524.84
socfb-MIT	4657(4657)	**3.25**	4657(4657)	4.53
socfb-OR	36551(36553.1)	571.88	**36548(36548.4)**	74.84
socfb-Penn94	31181(31183.7)	606.92	**31161(31164)**	414.21
socfb-Stanford3	8518(8518)	6.35	8518(8518)	**1.69**
socfb-Texas84	28177(28182.9)	570.79	**28166(28170.8)**	481.56
socfb-uci-uni	866766(866766)	**18.57**	866766(866766)	19.32
socfb-UCLA	15225(15226.4)	466.85	**15223(15224.1)**	242.24
socfb-UConn	13233(13233.7)	442.06	**13230(13231.7)**	188.83
socfb-UCSB37	11261(11262.1)	**327.93**	11261(11262.1)	395.18
socfb-UF	27320(27323.1)	544.18	**27306(27308.4)**	370.66
socfb-UIllinois	24104(24105.9)	529.50	**24092(24093.6)**	350.65
socfb-Wisconsin87	18389(18391.1)	541.04	**18383(18384.6)**	408.83

Table 14: Experiment results of NuMVC2+p and FastVC2+p (continued).

instance	NuMVC2+p		FastVC2+p	
	min(avg)	time	min(avg)	time
sc-ldoor	856754(**856754**)	81.93	856754(856754.5)	397.38
sc-msdoor	381558(**381558**)	560.29	381558(381559.2)	76.89
sc-nasasrb	51248(51249.9)	330.27	**51239(51239.2)**	275.69
sc-pkustk11	83911(83911)	**0.56**	83911(83911)	1.92
sc-pkustk13	**89221(89221.6)**	294.28	89227(89228.5)	51.87
sc-pwtk	207684(207695.9)	449.29	**207673(207681.9)**	796.31
sc-shipsec1	117282(117326.1)	997.78	**117276(117292.4)**	893.55
sc-shipsec5	147131(147152.1)	995.17	**147043(147055.2)**	545.29
soc-BlogCatalog	20752(20752)	**0.36**	20752(20752)	0.37
soc-brightkite	21190(21190)	0.02	21190(21190)	0.02
soc-buzznet	30613(30613.6)	0.46	30613(**30613**)	0.44
soc-delicious	85383(85391.6)	550.74	**85341(85342.7)**	537.59
soc-digg	103234(103234)	1.45	103234(103234)	**1.43**
soc-dolphins	34(34)	<0.01	34(34)	<0.01
soc-douban	8685(8685)	**0.03**	8685(8685)	0.03
soc-epinions	9757(9757)	<0.01	9757(9757)	<0.01
soc-flickr	153271(153271)	0.91	153271(153271)	**0.59**
soc-flixster	96317(96317)	1.25	96317(96317)	**1.21**
soc-FourSquare	90108(90108)	1.38	90108(90108)	**1.34**
soc-gowalla	84222(84222)	**0.12**	84222(84222)	1.17
soc-karate	14(14)	<0.01	14(14)	<0.01
soc-lastfm	78688(78688)	0.73	78688(78688)	**0.71**
soc-livejournal	**1868903(1868903.2)**	619.11	1868917(1868918.4)	5.25
soc-LiveMocha	43427(43427)	**0.33**	43427(43427)	0.35
soc-orkut	2190339(2190777.1)	999.8	**2171200(2171236.4)**	996.03
soc-pokec	844272(844306.2)	204.47	**843377(843380.4)**	574.95
soc-slashdot	22373(22373)	0.05	22373(22373)	**0.04**
soc-twitter-follows	2323(2323)	0.08	2323(2323)	0.08
soc-wiki-Vote	406(406)	<0.01	406(406)	<0.01
soc-youtube	146376(146376)	0.28	146376(146376)	0.28
soc-youtube-snap	276945(276945)	0.50	276945(276945)	0.51
tech-as-caida2007	3683(3683)	<0.01	3683(3683)	<0.01
tech-as-skitter	**525163(525187.4)**	963.41	525269(525277.9)	596.61
tech-internet-as	5700(5700)	<0.01	5700(5700)	<0.01
tech-p2p-gnutella	15682(15682)	<**0.01**	15682(15682)	0.01
tech-RL-caida	**74593(74597)**	832.06	74651(74654)	862.06
tech-routers-rf	795(795)	<0.01	795(795)	<0.01
tech-WHOIS	2284(2284)	<0.01	2284(2284)	<0.01
web-arabic-2005	114420(114420)	0.12	114420(114420)	**0.10**
web-BerkStan	5384(5384)	<0.01	5384(5384)	<0.01
web-edu	1451(1451)	<0.01	1451(1451)	<0.01
web-google	498(498)	<0.01	498(498)	<0.01
web-indochina-2004	7300(7300)	<0.01	7300(7300)	<0.01
web-it-2004	**414507(414510.2)**	19.71	414528(414528)	281.47
web-polblogs	244(244)	<0.01	244(244)	<0.01
web-sk-2005	58173(58173.4)	86.29	58173(**58173**)	1.41
web-spam	2297(2297)	<0.01	2297(2297)	<0.01
web-uk-2005	127774(127774)	2.55	127774(127774)	2.55
web-webbase-2001	**2651(2651)**	240.22	2652(2652)	<0.01
web-wikipedia2009	**648294(648294)**	112.3	648305(648314.9)	1.76

graph to a smaller graph, which is then solved by the local search algorithm. Finally, the vertices that have been fixed in the preprocessing procedure and the vertex cover returned by the local search algorithm for the simplified graph compose a vertex cover for the original graph.

Experiment results of FastVC2+p and NuMVC2+p are reported in Tables 13 and 14. We firstly compare FastVC2+p and NuMVC2+p, with the aim of trying to establish the latest state of the art in heuristic solvers for MinVC on massive sparse graphs. The experiments are conducted on the whole

Table 15: Comparison of NuMVC2+p against NuMVC2, and FastVC2+p against FastVC2.

	NuMVC2+p vs NuMVC2	FastVC2+p vs FastVC2
#better-solution	52	37
#equal-solution	44	60
#worse-solution	3	5

benchmark suite, and are based on the experiment protocol in Section 2.3. Note that the run time of NuMVC2+p and FastVC2+p include both the run time of D1+D2+Dom and that of the local search algorithm. The comparison results show that these two MinVC solvers are competitive with each other, and FastVC2+p has overall better performance. Specifically, with respect to solution quality, FastVC2+p performs better than NuMVC2+p on 23 instances, while NuMVC2+p is better on 10 instances. For the remaining 69 instances, the two solvers obtain the same quality solutions. As for run time, the averaged run time over all runs on all 102 instances for FastVC2+p is 126 seconds, which is faster than that for NuMVC2+p (162 seconds).

We are also interested in the improvement due to the preprocessing algorithm on NuMVC2 and FastVC2. To this end, we compare the two solvers with preprocessing against their original algorithms. That is, we compare NuMVC2+p against NuMVC2, and compare FastVC2+p against FastVC2, with focus on solution quality. The comparison results are summarized in Table 15. As shown in the table, NuMVC2+p finds better solutions than NuMVC2 on 52 instances while being worse on only 3 instances; FastVC2+p finds better solutions than FastVC2 on 37 instances, while being worse on only 5 instances. This confirms the contribution of the preprocessing algorithm.

7. Summary and Future Work

This work explored techniques for solving the MinVC problem in massive sparse graphs, including ideas in construction procedure, local search and preprocessing. We have developed a local search algorithm for MinVC called FastVC, based on two new heuristics. The first heuristic is a construction procedure with a complexity of $O(|E|)$, which strikes a good balance between solution quality and time complexity. The second one is the Best from Multiple Selections (BMS) heuristic, which approximates the minimum loss heuristic very well and lowers the complexity of each removing-vertex selection step from $O(|V|)$ to $O(1)$. Thanks to these two heuristics, the FastVC algorithm performs much better than the state of the art algorithm NuMVC on massive graphs. Experiments on massive real-world graphs show that FastVC finds smaller vertex covers than NuMVC on most graphs.

We also improved previous construction heuristics with ideas in our construction heuristic as well as efficient data structure, and integrated three construction heuristics to improve both NuMVC and FastVC. Furthermore, we have developed a two-phase preprocessing algorithm for MinVC, which is effective and fast. The preprocessing algorithm was applied to improve local search MinVC algorithms, resulting in the NuMVC2+p and FastVC2+p solvers, which further improved the performance on large graphs according to our experiments.

In the future, we would like to design more efficient heuristic algorithms for MinVC as well as other graph problems on massive graphs. We are also interested in applying the ideas in this work to solve large instances of other combinatorial problems.

Acknowledgement

This work is supported by National Natural Science Foundation of China 61502464, and 973 Program 2014CB340301. Shaowei Cai is also supported by Youth Innovation Promotion Association, Chinese Academy of Sciences. We would like to thank the anonymous referees for their helpful comments.

References

Abu-Khzam, F. N., Collins, R. L., Fellows, M. R., Langston, M. A., Suters, W. H., & Symons, C. T. (2004). Kernelization algorithms for the vertex cover problem: Theory and experiments. In *Proceedings of the Sixth Workshop on Algorithm Engineering and Experiments and the First Workshop on Analytic Algorithmics and Combinatorics*, pp. 62–69.

Akiba, T., & Iwata, Y. (2016). Branch-and-reduce exponential/FPT algorithms in practice: A case study of vertex cover. *Theor. Comput. Sci.*, *609*, 211–225.

Andrade, D. V., Resende, M. G. C., & Werneck, R. F. F. (2008). Fast local search for the maximum independent set problem. In *Workshop on Experimental Algorithms*, pp. 220–234.

Argelich, J., Li, C. M., Manyà, F., & Planes, J. (2016). The MaxSAT evaluations 2010–2016.. http://www.maxsat.udl.cat.

Barabási, A., & Albert, R. (1999). Emergence of scaling in random networks. *Science*, *286*(5439), 509.

Cai, S. (2015). Balance between complexity and quality: Local search for minimum vertex cover in massive graphs. In *Proceedings of IJCAI 2015*, pp. 747–753.

Cai, S., & Lin, J. (2016). Fast solving maximum weight clique problem in massive graphs. In *Proceedings of IJCAI 2016*, pp. 568–574.

Cai, S., Lin, J., & Su, K. (2015). Two weighting local search for minimum vertex cover. In *Proceedings of AAAI 2015*, pp. 1107–1113.

Cai, S., Su, K., Luo, C., & Sattar, A. (2013). NuMVC: An efficient local search algorithm for minimum vertex cover. *J. Artif. Intell. Res. (JAIR)*, *46*, 687–716.

Cai, S., Su, K., & Sattar, A. (2011). Local search with edge weighting and configuration checking heuristics for minimum vertex cover. *Artif. Intell.*, *175*(9-10), 1672–1696.

Chen, J., Kanj, I. A., & Jia, W. (2001). Vertex cover: Further observations and further improvements. *J. Algorithms*, *41*(2), 280–301.

Chesler, E. J., & Langston, M. A. (2005). Combinatorial genetic regulatory network analysis tools for high throughput transcriptomic data. In *Systems Biology and Regulatory Genomics, Joint Annual RECOMB 2005 Satellite Workshops on Systems Biology and on Regulatory Genomics, San Diego, CA, USA; December 2-4, 2005, Revised Selected Papers*, pp. 150–165.

Dinur, I., & Safra, S. (2005). On the hardness of approximating minimum vertex cover. *Annals of Mathematics*, *162*(2), 439–486.

Eubank, S., Kumar, V. S. A., Marathe, M. V., Srinivasan, A., & Wang, N. (2004). Structural and algorithmic aspects of massive social networks. In *Proceedings of SODA 2004*, pp. 718–727.

Fang, Z., Chu, Y., Qiao, K., Feng, X., & Xu, K. (2014). Combining edge weight and vertex weight for minimum vertex cover problem. In *Proceedings of FAW 2014*, pp. 71–81.

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

Garey, M., & Johnson, D. (1979). *Computers and Intractability: A Guide to the Theory of NP-completeness*. Freeman, San Francisco, CA, USA.

Hoos, H., & Stützle, T. (2004). *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann, San Francisco, CA, USA.

Jin, Y., & Hao, J. (2015). General swap-based multiple neighborhood tabu search for the maximum independent set problem. *Eng. Appl. of AI*, *37*, 20–33.

Karakostas, G. (2005). A better approximation ratio for the vertex cover problem. In *Proceedings of ICALP 2005*, pp. 1043–1050.

Kavalci, V., Ural, A., & Dagdeviren (2014). Distributed vertex cover algorithms for wireless sensor networks. *International Journal of Computer Networks & Communications (IJCNC)*, *6*, 95–110.

Lamm, S., Sanders, P., Schulz, C., Strash, D., & Werneck, R. F. (2016). Finding near-optimal independent sets at scale. In *Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2016, Arlington, Virginia, USA, January 10, 2016*, pp. 138–150.

Li, C. M., & Huang, W. Q. (2005). Diversification and determinism in local search for satisfiability. In *Proceedings of SAT 2005*, pp. 158–172.

Lu, L., & Chung, F. (2006). *Complex Graphs and Networks*. American Math. Society, New York, USA.

Papadimitrious, C. H., & Steiglitz, K. (1982). *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, New York, USA.

Pullan, W. (2009). Optimisation of unweighted/weighted maximum independent sets and minimum vertex covers. *Discrete Optimization*, *6*, 214–219.

Richter, S., Helmert, M., & Gretton, C. (2007). A stochastic local search approach to vertex cover. In *Proceedings of KI 2007*, pp. 412–426.

Rosin, C. D. (2014). Unweighted stochastic local search can be effective for random CSP benchmarks. *CoRR*, *abs/1411.7480*.

Rossi, R. A., & Ahmed, N. K. (2014). Coloring large complex networks. *Social Network Analysis and Mining*, *4*(1), 228.

Rossi, R. A., & Ahmed, N. K. (2015). The network data repository with interactive graph analytics and visualization. In *Proceedings of AAAI 2015*.

Rossi, R. A., Gleich, D. F., Gebremedhin, A. H., & Patwary, M. M. A. (2014). Fast maximum clique algorithms for large graphs. In *WWW (Companion Volume)*, pp. 365–366.

Selman, B., Levesque, H. J., & Mitchell, D. G. (1992). A new method for solving hard satisfiability problems. In *Proceedings of AAAI 1992*, pp. 440–446.

Strash, D. (2016). On the power of simple reductions for the maximum independent set problem. In *Proceedings of COCOON 2016*, pp. 345–356.

Verma, A., Buchanan, A., & Butenko, S. (2015). Solving the maximum clique and vertex coloring problems on very large sparse networks. *INFORMS Journal on Computing*, *27*(1), 164–177.

Wang, Y., Cai, S., & Yin, M. (2016). Two efficient local search algorithms for maximum weight clique problem. In *Proceedings of AAAI 2016*, pp. 805–811.

Yadav, T., Sadhukhan, K., & Rao, A. M. (2016). Approximation algorithm for n-distance minimal vertex cover problem. *CoRR*, *abs/1606.02889*.