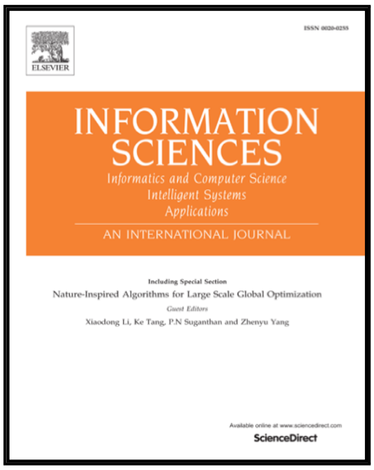
Accepted Manuscript



Towards Faster Local Search for Minimum Weight Vertex Cover on Massive Graphs

Shaowei Cai, Yuanjie Li, Wenying Hou, Haoran Wang

PII: S0020-0255(18)30670-4
DOI: https://doi.org/10.1016/j.ins.2018.08.052
Reference: INS 13896

To appear in: *Information Sciences*

Received date: 7 February 2018
Revised date: 16 August 2018
Accepted date: 24 August 2018

Please cite this article as: Shaowei Cai, Yuanjie Li, Wenying Hou, Haoran Wang, Towards Faster Local Search for Minimum Weight Vertex Cover on Massive Graphs, *Information Sciences* (2018), doi: https://doi.org/10.1016/j.ins.2018.08.052

Towards Faster Local Search for Minimum Weight Vertex Cover on Massive Graphs☆

Shaowei Cai[a,b], Yuanjie Li[a,b], Wenying Hou[a], Haoran Wang[a,b]

[a] *State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China*
[b] *School of Computer and Control Engineering, University of Chinese Academy of Sciences, Beijing, China*

Abstract

The minimum weight vertex cover (MWVC) problem is a well known NP-hard problem with various real-world applications. In this paper, we design an efficient algorithm named FastWVC to solve MWVC problem in massive graphs. To this end, we propose a construction procedure, which aims to generate a quality initial vertex cover in short time. We also propose a new exchange step for reconstructing a vertex cover. Additionally, a cost-effective strategy is used for choosing adding vertices , which can accelerate the algorithm. Experiments on 102 instances were conducted to confirm the effectiveness of our algorithm. The results show that the FastWVC algorithm outperforms other algorithms in terms of both solution quality and computational time in most of the instances. We also carry out experiments that analyze the effectiveness of the underlying ideas.

Keywords: Minimum weighted vertex cover, Local search, Massive graph

1. Introduction

The minimum vertex cover (MVC) problem is to find a minimum sized vertex cover in a graph, where a vertex cover is a subset of vertices that contains at least one endpoint of each edge. The minimum weight vertex cover (MWVC) problem is a generalization of MVC. In a vertex weighted graph, each vertex has a positive weight and the purpose of the MWVC problem is to find a vertex cover with the minimum weight. The MWVC problem has applications in various fields such as network flow, circuit design, transportation and telecommunication.

The MVC problem is NP hard. Moreover, it is NP-hard to approximate MVC within any factors smaller than 1.3606 [6]. Due to the computational intractability of the MVC problem, various heuristic algorithms have been proposed to find approximate solutions within reasonable time. Among them, the most successful ones share

☆This is an improved and extended version of a conference paper [11].

Email addresses: caisw@ios.ac.cn (Shaowei Cai), liyj@ios.ac.cn (Yuanjie Li), houwyvenny@126.com (Wenying Hou), wanghr@ios.ac.cn (Haoran Wang)

the same method called local search, which moves from solution to solution based on the iterative use of certain criterion until a deemed optimal solution is found or a time bound is elapsed. Typical local search algorithms for MVC include COVER [15], EWCC [4] and NuMVC [3]. Particularly, NuMVC made a significant improvement on solving the popular DIMACS and BHOSLIB benchmarks. Its main ideas include two-stage exchange step and the forgetting mechanism for edge weighting, as well as the configuration checking strategy [4] . However, these algorithms do not perform well on massive graphs. Some researchers have been working on algorithms that are tailored to massive graphs. The most representative one is FastVC algorithm [2]. It uses a fast heuristic for constructing a vertex cover and a probabilistic sampling heuristic for choosing the vertex. Since the introduction of FastVC, several algorithms have been proposed for solving MVC on large graphs [12, 9, 19], most of which are improved from FastVC.

Research on MWVC problem is relatively less, which may be due to the fact that MWVC is more complicated than MVC. Most works on MWVC are also focused on heuristic algorithms. In [5], a greedy algorithm was used to find a feasible solution. A population-based iterated greedy algorithm [1] refines a population of solutions at each iteration. Ant colony optimization was also used to get a quality approximate solution [17, 8]. In [18], the authors proposed an algorithm which combined genetic algorithms and greedy heuristic. A hybrid intelligent algorithm that integrates stochastic simulation with genetic algorithm was designed to solve the MWVC problem under stochastic environments [13]. Recently, a tabu search algorithm named Multi-Start Iterated Tabu Search algorithm (MS-ITS) [20] achieved state-of-the-art performance on a broad range of benchmarks. As for solving MWVC on massive graphs, a recent algorithm named Diversion Local Search based on Weighted Configuration Checking (DLSWCC) [10] has made a significant improvement. DLSWCC combines the Weighted Configuration Checking (WCC) strategy with a dynamic scoring function .

The direction of solving MWVC on massive graphs calls for more efficient algorithms. There are several shortcomings in existing algorithms. Firstly, they use simple strategies to obtain one initial vertex cover [20, 10]. This initial vertex cover may be far from the optimal solution, taking the search process much time to move closer to the final solution. Secondly, in each exchange step of local search, most of them remove just one vertex from the candidate solution and then reconstruct the candidate solution to a vertex cover. However, the number of uncovered edges resulted by removing just one vertex from the candidate solution may be too small, making the search range in the search phase narrow. Thirdly , in the vertex cover reconstruction phase, some vertices need to be added into the candidate solution. Previous algorithms mainly achieve this by searching all vertices that are not in the candidate solution, and then choosing vertices based on some rules. This strategy is time-consuming especially when the size of the instance is very large.

We try to address the shortcomings mentioned above. Firstly, we propose an efficient construction procedure called *ConstructWVC*, which is an improved and iterative version of *ConstructVC* in FastVC [2]. This construction procedure is used to produce a high quality initial solution. It consists of two phases. Different vertex covers are constructed according to an edge scanning heuristic with different orders, and the best one is selected, which is then shrunk by removing redundant vertices. The intention of this strategy is to take the advantage of the low complexity of the construction process to get a competitive initial candidate solution.

Secondly, we propose a new exchange step for reconstructing a vertex cover. As with previous local search algorithms for MWVC, our algorithm moves from a vertex cover to another vertex cover in each step. In the beginning of each step, the current candidate solution C is a vertex cover, and the algorithm moves to a new vertex cover by removing two vertices from C and adding vertices to it until C becomes a vertex cover again. The first removed vertex is selected according to a greedy strategy, while the second is chosen according to a balanced strategy.

Thirdly, we use a cost-effective strategy for choosing adding vertices. When selecting a vertex to add into the candidate solution, we simply scan the closed neighborhood of the removed vertices in this step. Each time a vertex with the largest *gain* is picked to add into C, until C becomes a vertex cover. This strategy is equal to the time-consuming strategy that scans all vertices not in C, but is much more efficient. Here *gain* represents the contribution of adding the vertex to the candidate solution, and will be introduced in Section 2.

Based on these techniques, we develop a local search algorithm named FastWVC for MWVC. We conducted an extensive experiment study on a large range of benchmarks, covering a variety of fields such as biological networks, collaboration networks, social networks. We compare FastWVC algorithm with state-of-the-art algorithms including MS-ITS, DLSWCC and ILS-VND [14] . The results show that FastWVC algorithm outperforms them on most benchmark graphs.

The reminder of this paper is organized as follows: Section 2 presents some preliminary knowledge. Section 3 presents the framework of the FastWVC algorithm. Section 4 explains the three main ideas. Detailed description of our FastWVC algorithm is showed in Section 5. Section 6 presents the empirical results. Finally, Section 7 gives our conclusions and future works.

2. Preliminaries

In this section, we introduce some basic definitions and background knowledge. We also introduce the popular techniques that will be used in our algorithm.

2.1. Basic Definitions

Given an undirected weighted graph $G = (V, E, w)$, V is the vertex set , E is the edge set of G, and each vertex v has a weight $w(v) > 0$. Each edge $e = (u, v)$ consists of two vertices u and v, and we say u and v are the endpoints of edge e. Two vertices are neighbors if they belong to a same edge , and we use $N(v) = \{u \in V | (u, v) \in E\}$ to denote all neighbors of v, and $N[v] = N(v) \cup \{v\}$ is the closed neighborhood of vertex v. A *candidate solution*, denoted as C , is a subset of V, and we say it covers an edge e if it contains at least one endpoint of e. We use $w(C)$ to denote the total weight of vertices in C.

A vertex cover is a special *candidate solution* which covers all $e \in E$. An *independent set* is a subset of V where no two vertices are neighbors. A vertex set C is a vertex cover of G if and only if $V \setminus C$ is an independent set of G. The MWVC problem is to find a vertex cover with the minimum weight, which is equivalent to the Maximum Weight Independent Set (MWIS) problem, i.e., seeking for an independent set with the largest weight.

A vertex cover is minimal if removing any vertex out of it would make it no longer a vertex cover. For convenience, we use C^* to denote the best found solution during the search process. Further, we use $s_v = \{1,0\}$ to denote the state of a vertex. If $v \in C$, then $s_v = 1$, otherwise, $s_v = 0$. The *age* of a vertex is defined as the number of steps happened since v last changed its state.

For local search algorithms adopting edge weighting mechanism, we use a weighting function $edge\_w$ and each edge $e \in E$ is associated with a positive number $edge\_w(e)$ as its weight. The cost of C denoted by $cost(C)$ means the total weight of edges uncovered by C, which can be formally defined as

$$cost(C) = \sum_{e \in E \text{ and } e \text{ is uncovered by } C} edge\_w(e) \,. \tag{1}$$

We denote the change on *cost* caused by changing the state of a vertex v as $dscore(v)$. Formally, we define *dscore* as

$$dscore = cost(C) - cost(C'), \tag{2}$$

where C' is the candidate solution after changing the state of v, that is, if $v \in C$, $C' = C \backslash \{v\}$, otherwise $C' = C \cup \{v\}$. Note that $dscore(v) \geqslant 0$ if $v \notin C$ and $dscore(v) \leqslant 0$ when $v \in C$.

In local search algorithms for MWVC, usually a scoring function that considers both vertex weights and the cost is employed. For example, in DLSWCC [10], such a function is defined as follow

$$score(v) = \frac{dscore(v)}{w(v)} \,. \tag{3}$$

In this paper, we use two conceptions, namely *gain* and *loss*, which are easier to understand and more clear when explaining the algorithm. They are actually modified versions of *score*, and are used as criteria to choose a vertex to add into or remove from C respectively. Here are the formal definitions :

$$gain(v) = \frac{dscore(v)}{w(v)} \,. \tag{4}$$

$$loss(v) = \frac{|dscore(v)|}{w(v)} \,. \tag{5}$$

Note that only vertices in $V \setminus C$ have *gain* values and only vertices in C have *loss* values.

2.2. Configuration Checking

The Configuration Checking (*CC*) strategy, first proposed in [4], is a strategy aiming to solve the cycling problem. The cycling problem refers to revisiting the same vertices that have been visited recently. It is a main issue for local search algorithms. The *CC* strategy has been successfully applied to local search algorithms for MVC [4, 3] and MWVC [10], greatly improving the performance of these algorithms. The idea of configuration checking is to forbid adding any vertex whose circumstance information has not been changed since its last removal. For each vertex, the circumstance information is formally defined as the concept of configuration. Typically, the configuration of a vertex v refers to a vector consisting of the states (i.e. being selected or not) of vertices in $N(v)$.

The CC strategy is usually implemented with a boolean array $confChange$ for vertices, where $confChange[v] = 0$ implies the configuration of v has not changed since v's last removing from C, and $confChange[v] = 1$ on the contrary. When adding vertex into C, only vertices with $confChange$ values of 1 are eligible to be added. The $confChange$ array is maintained by the following rules:

Rule 1: For each $v \in V$, $confChange[v]$ is initialized as 1.
Rule 2: When removing v from C, $confChange[v]$ is set to 0; for each $u \in N(v)$, $confChange[u]$ is set to 1.
Rule 3: When adding v into C, for each $u \in N(v)$, $confChange[u]$ is set to 1.

A variant of *CC* strategy is called Weighted Configuration Checking (*WCC*). We adopt the *WCC* strategy in our algorithm. It considers the circumstance of a vertex as two parts, one of which considers the states of its neighbors, and the other considers the states of its incident edges. Specifically, apart from the three rules above, it adds a new rule:

Rule 4: When updating $edge\_w[e]$, $confChange[u]$ and $confChange[v]$ are set to 1, where u and v are the endpoints of edge e.

2.3. The Tabu Mechanism

Tabu search [7] is a meta-heuristic search method that uses memory structures to deal with the cycling problem of local search. To prevent the local search to immediately return to a previously visited candidate solution, the tabu mechanism forbids reversing the recent changes. Our algorithm uses tabu technique in vertex selection phase by maintaining an array called *tabu list*. If a vertex has just been added into C in the most recent step, it is added into the *tabu list*, and vertices in *tabu list* are forbidden to be removed from C. For a vertex, we use $tabu[v] = 1$ to mean that v is in the tabu list, and $tabu[v] = 0$ otherwise.

3. Algorithmic Framework

This section describes the general framework of the FastWVC algorithm. Detailed description and analysis will be presented in Section 5.

At the beginning, an initial vertex cover C is constructed by function *ConstructWVC*. Then the algorithm repeats the main loop until reaching time limit. In each step, the

Algorithm 1: FastWVC Framework

```
Input: An undirected graph G = (V, E), the cutoff time
Output: A minimum weighted vertex cover of G
begin
    C ← ConstructWVC();
    C* ← C;
    while elapsed_time < cutoff do
        remove two vertices from C;
        while there exist edge uncovered by C do
            add a vertex into C;
        remove redundant vertices from C;
        if w(C) < w(C*) then
            C* ← C;
    return C*;
```

algorithm firstly removes two vertices, according to two different strategies. After that, the algorithm reconstructs a vertex cover by adding vertices into C until all edges are covered . During the search, the values of $confChange$, *tabu* and edge weights are updated accordingly. At the end of each local search step, redundant vertices (i.e., with a $loss$ value of 0) are removed from C. If the obtained solution C is better than C^*, C^* is updated to C.

4. Main Ideas

This section presents the main ideas used in FastWVC, including a construction procedure, a new exchange step, and a cost-effective strategy for choosing the adding vertices.

4.1. ConstructWVC

In this subsection, we introduce the $ConstructWVC$ procedure, which is utilized by the FastWVC algorithm to generate an initial vertex cover.

The $ConstructWVC$ procedure is inspired by the $ConstructVC$ procedure [2] for MVC, which is a greedy algorithm designed to find a vertex cover quickly for massive graphs. The $ConstructVC$ procedure has an extending phase and a shrinking phase. In the extending phase, it traverses all edges, if the edge being checked is uncovered, the endpoint with a higher degree is added into C. In the shrinking phase, redundant vertices are removed to obtain a minimal vertex cover. The time complexity of $ConstructVC$ is $O(m)$, where m is the number of edges.

Our $ConstructWVC$ procedure can be viewed as an iterative version of $ConstructVC$, and it is outlined below. The procedure consists of two phases: a repeated extending phase and a shrinking phase. In the repeated extending phase, a vertex cover is first generated utilizing the same method in the extending phase of $ConstructVC$ (line 3-5). Then, the algorithm utilizes a random strategy to generate vertex covers

Algorithm 2: ConstructWVC

```
   Input: An undirected graph G = (V,E), max_tries
   Output: A weighted vertex cover of G
1  begin
2      C ← ∅;
3      foreach e ∈ E do
4          if both endpoints of e are not in C then
5              put the endpoint with larger degree(v)/w(v) into C;
6      for tries ← 0; tries < constuct_tries; tries ← tries + 1 do
7          C' ← ∅;
8          foreach uncovered e ∈ E (in a random order) do
9              if both endpoints of e are not in C then
10                 put the endpoint with larger degree(v)/w(v) into C';
11         if C' is better than C then
12             C ← C';
13     harm_value(v) ← 0, for each v ∈ C;
14     foreach e ∈ E do
15         if only one endpoint v of e belongs to C then
16             harm_value(v) ← harm_value(v) + 1;
17     foreach v ∈ C do
18         if harm_value(v) = 0 then
19             C ← C \ {v};
20             harm_value(u) ← harm_value(u) + 1 for each u ∈ N(v);
21     return C;
```

for *constuct_tries* times, and finally the best vertex cover (with minimum weight) is handed to the shrinking phase. More specifically, the construction of each vertex cover works as follow: starting with an empty set C' (line 7), while there are uncovered edges, the algorithm randomly chooses an uncovered edge e and puts the endpoint of e which has a higher $degree(v)/w(v)$ into C' (line 8-10).

The best vertex cover built in the repeated extending phase is handed to the shrinking phase. In the shrinking phase, the algorithm removes redundant vertices from C, which is accomplished with the help of an array denoted as *harm_value*, where $harm\_value(v)$ measures the number of edges that would change from covered to uncovered if v were removed from C . For each $v \in C$, the algorithm removes a vertex v if its *harm_value* is 0 and then updates $harm\_value(u)$ for each $u \in N(v)$ accordingly (line 17-20).

The difference between *ConstructVC* and *ConstructWVC* is in the extending phase, where the former constructs one vertex cover as the initial solution but the latter constructs multiple vertex covers and chooses the best one. There are two reasons we make this modification. Firstly, it takes little time to construct more vertex covers. Secondly, the initial solution constructed only once cannot guarantee a good quality. It may be

ACCEPTED MANUSCRIPT

far away from the final solution, taking the search process more time to move closer to the optimal solution. Comparatively, the initial solution constructed by *ConstructWVC* can guarantee a competitive one.

4.2. A New Exchange Step

We propose a new exchange step for our algorithm. In each step, the algorithm tries to seek for another feasible solution by removing two vertices from C and reconstructing C. We choose the first removing vertex greedily and choose the second removing vertex based on a balanced strategy. More specifically, a vertex with the minimum *loss* is firstly removed, making C an infeasible solution. Then one more vertex is selected to remove by the *BMS* (Best from Multiple Selections) heuristic [2] and the *tabu* mechanism. In detail, the algorithm chooses t (a parameter) vertices from C, and then selects the one not belonging to *tabu list* and with the minimum *loss*. After removing the two vertices, the algorithm adds vertices into C until it covers all edges.

Instead of removing one vertex, we suggest to remove two vertices in each step. The strategy of removing one vertex from C in each step results in a limited number of uncovered edges, especially for sparse graphs such as most massive graphs in real world, making the search range narrower. The purpose we suggest to remove two vertices at each step is to produce more uncovered edges in the removing phase so that more vertices will be considered in the adding phase, making the search range wider and increasing the possibility of finding a better alternative solution. Also, while the first removing vertex is chosen according to a greedy strategy, the second one is selected by a balanced strategy, striking a good balance between intensification and diversification.

4.3. A Cost-effective Strategy for Choosing Adding Vertices

To reconstruct a valid solution, some vertices should be added into C after two vertices are removed. Most algorithms scan all vertices outside C to select adding vertices. When the size of $V \setminus C$ is large, it is rather time-consuming to scan all these vertices. Observing that all uncovered edges are incident to at least one removed vertex in the removing phase, we only need to scan those vertices in $N[u] \cup N[w]$, where u and w are the last two vertices that have just been removed.

Hence, our algorithm chooses the adding vertitces as follows: when selecting a vertex to add into C, scan vertices in $N[u] \cup N[w] \setminus C$. Then, choose a vertex with the largest *gain*, if there is more than one such vertex, break ties in favor of the oldest one. This strategy can accelarate the adding phase of the local search procedure. Experimental results in Section 6 will confirm its effectiveness.

5. The FastWVC Algorithm

In this section, we present the FastWVC algorithm and give a detailed description. Our algorithm is outlined in Algorithm 3, as described below.

The initial solution C is constructed by *ConstructWVC*. The best found solution C^* is initialized as C. The weight of each edge is set to 1 , $confChange(v)$ is set to

Algorithm 3: FastWVC

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

begin
- $C \leftarrow ConstructWVC()$;
- $C^* \leftarrow C$;
- for each $e \in E$, $edge\_w(e) \leftarrow 1$;
- for each $v \in V$, $confChange(v) \leftarrow 1$;
- calculate *gain* and *loss* of vertices;
- $tabulist \leftarrow \varnothing$;
- **while** *elapsed_time* $<$ *cutoff* **do**
    - choose a vertex w with minimum *loss* from C, breaking ties in favor of the oldest one;
    - $C \leftarrow C \setminus \{w\}$;
    - $confChange(w) \leftarrow 0$, $confChange(z) \leftarrow 1$ for each vertex $z \in N(w)$;
    - choose a vertex u with $tabu[u] = 0$ from C according to *BMS* strategy, breaking ties in favor of the oldest one;
    - $C \leftarrow C \setminus \{u\}$;
    - $confChange(u) \leftarrow 0$, $confChange(z) \leftarrow 1$ for each vertex $z \in N(u)$;
    - $tabulist \leftarrow \varnothing$;
    - **while** *some edge is uncovered by C* **do**
        - choose a vertex v, whose $confChange(v) = 1$, with maximum *gain* from $V \setminus C$, breaking ties in favor of the oldest one;
        - $C \leftarrow C \cup \{v\}$;
        - $tabulist \leftarrow tabulist \cup \{v\}$;
        - $confChange(z) \leftarrow 1$ for each vertex $z \in N(v)$;
        - $w(e) \leftarrow w(e) + 1$ for each uncovered edge e, and for its endpoints (x, y), $confChange(x) \leftarrow 1$ and $confChange(y) \leftarrow 1$;
    - remove redundant vertices from C;
    - **if** $w(C) < w(C^*)$ **then**
        - $C^* \leftarrow C$;
- **return** C^*;

1 for each $v \in V$ and then *gain* of vertices in $V \setminus C$ and and *loss* of vertices in C are calculated respectively. Finally, the *tabu list* is set empty.

After the initialization, the algorithm iteratively updats C and C^* until reaching the time limit. At the end of each step, if a better solution is found, the best found solution C^* is updated accordingly (lines 23-24).

Each step of the process consists of two phases, the removing phase and the adding phase. In the removing phase, the algorithm first removes a vertex from C with the minimum *loss*, note that after this removing, *loss* of vertices are updated accordingly. Next, it removes one more vertex u according to the *BMS* strategy [2], which samples t (=50 in this work) vertices from C, and chooses the one not belonging to *tabu list* and with the minimum *loss*. The $confChange$ values are updated according to the *CC* Rules introduced in Section 2.

In the adding phase, a loop is executed until C becomes a vertex cover again. In each iteration, the algorithm chooses a vertex $v \notin C$ such that $confChange$ is 1 and the *gain* is the largest , breaking ties by preferring the oldest one. Then v is added into C and also *Tabu list*. After that, weights of all uncovered edges are increased by 1. The $confChange$ values are updated according to the *CC* Rules.

Finally, redundant vertices (i.e., those vertices for which *loss* is 0) are removed from C, in a similar way as in $ConstructWVC$, to make C become a minimal vertex cover. If the obtained solution C is better than C^*, update C^* to C.

6. Empirical Results

In this section, we present the experimental results. We compare FastWVC with three state-of-the-art algorithms, and show that our algorithm has better performance.

6.1. Benchmark Instances

All the graphs used in our experiments are obtained from Network Data Repository [16], including different types of real-life graphs, which can be categorized into biological networks, collaboration networks, interaction networks, infrastructure networks, massive network data, facebook networks, technological networks, web graphs, scientific networks, retweet networks and recommendation networks. Within all the instances, the number of vertices varies from about 30 to 60 million and the number of edges takes value from about 80 to 100 million. The weight of each vertex is assigned to a value from [20,100] uniformly at random, as with the generation method adopted in testing DLSWCC [10].

6.2. Experimental Preliminaries

Implementation: FastWVC is implemented in C++. FastWVC has two parameters: the *constuct_tries* parameter for $ConstructWVC$ and the t parameter for the BMS heuristic (generating several solutions and choose the best one) when choosing the second removing vertex. Indeed, the $ConstructWVC$ procedure also uses a BMS heuristic, and *constuct_tries* parameter is for the BMS heuristic. These two parameters are set to 50, as suggested in the work where the BMS heuristic is proposed [2], with both theoretical and experimental analysis.

Table 1: Experiment results on massive graphs

instance	$\|V\|$	$\|E\|$	MS-ITS			DLSWCC			ILS-VND			FASTWVC		
			w(min)	w(avg)	avg(t)	w(min)	w(avg)	avg(t)	w(min)	w(avg)	avg(t)	w(min)	w(avg)	avg(t)
bio-celegans	453	2025	**13767**	**13767.0**	0.54	**13767**	**13767.0**	0.09	**13767**	**13767.0**	<0.01	**13767**	**13767.0**	0.04
bio-diseasome	516	1188	**15096**	**15096.0**	0.05	**15096**	**15096.0**	0.07	**15096**	**15096.0**	<0.01	**15096**	**15096.0**	0.01
bio-dmela	7393	25569	147876	147949.0	2636.45	147210	147258.8	91.60	**147207**	**147208.2**	414.21	147241	147265.6	396.14
bio-yeast	1458	1948	24517	24530.5	16.50	**24495**	24498.9	1.80	**24495**	**24495.0**	0.03	**24495**	**24495.0**	0.08
ca-AstroPh	17903	196972	654523	654825.5	7587.39	644786	645016.3	297.96	643008	643028.7	230.40	**643007**	**643011.2**	418.77
ca-citeseer	227320	814134	N/A	N/A	N/A	7040526	7040764.7	481.55	**7014232**	**7016067.2**	993.52	7016968	7018567.5	993.79
ca-coauthors-dblp	540486	15245729	N/A	N/A	N/A	N/A	N/A	N/A	**27108326**	**27110233.8**	994.10	27111167	27112456.6	589.15
ca-CondMat	21363	91286	699396	699684.5	15395.41	681640	682003.8	369.07	**679072**	**679072.0**	145.11	679102	679108.3	393.15
ca-CSphd	1882	1740	29562	29574.0	8.26	**29497**	**29497.0**	4.84	**29497**	**29497.0**	0.04	**29497**	**29497.0**	42.90
ca-dblp-2010	226413	716460	N/A	N/A	N/A	6632625	6632925.4	351.23	**6602963**	**6604224.2**	995.63	6605746	6606471.6	996.66
ca-dblp-2012	317080	1049866	N/A	N/A	N/A	8979954	8980245.9	819.82	8957399	8958166.8	997.90	**8953523**	**8956561.0**	997.49
ca-Erdos992	6100	7515	**26945**	**26945.0**	4.34	**26945**	**26945.0**	0.08	**26945**	**26945.0**	0.09	**26945**	**26945.0**	<0.01
ca-GrQc	4158	13422	121676	121718.0	350.75	121567	121613.1	62.36	**121560**	**121560.0**	2.07	**121560**	121560.2	6.36
ca-HepPh	11204	117619	369191	369319.0	4569.85	366632	366828.7	198.17	**365469**	**365471.2**	149.62	365470	365475.2	297.90
ca-hollywood-2009	1069126	56306653	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	**48930484**	**48934214.2**	1000.00
ca-MathSciNet	332689	820644	N/A	N/A	N/A	7660999	7661455.3	569.05	7635837	7641013.7	998.43	**7614812**	**7619427.4**	997.14
ca-netscience	379	914	11570	11570.0	0.63	**11551**	**11551.0**	0.02	**11551**	**11551.0**	<0.01	**11551**	**11551.0**	0.02
ia-email-EU	32430	54397	N/A	N/A	N/A	**48447**	**48447.0**	5.09	**48447**	**48447.0**	0.34	**48447**	**48447.0**	0.15
ia-email-univ	1133	5451	32670	32670.0	35.95	**32666**	**32666.0**	0.77	**32666**	**32666.0**	0.94	**32666**	**32666.0**	0.69
ia-enron-large	33696	180811	N/A	N/A	N/A	695527	695879.5	523.12	**692139**	**692154.9**	214.21	692150	692161.3	613.55
ia-enron-only	143	623	**4827**	**4827.0**	<0.01	**4827**	**4827.0**	<0.01	**4827**	**4827.0**	<0.01	**4827**	**4827.0**	<0.01
ia-fb-messages	1266	6451	32476	32479.5	22.96	**32446**	**32446.0**	0.96	**32446**	**32446.0**	0.40	**32446**	**32446.0**	0.23
ia-infect-dublin	410	2765	16291	16291.0	0.22	**16289**	**16289.0**	0.27	**16289**	**16289.0**	0.04	**16289**	**16289.0**	0.21
ia-infect-hyper	113	2196	**5362**	**5362.0**	<0.01	**5362**	**5362.0**	<0.01	**5362**	**5362.0**	<0.01	**5362**	**5362.0**	<0.01
ia-reality	6809	7680	**4439**	**4439.0**	5.33	**4439**	**4439.0**	0.02	**4439**	**4439.0**	<0.01	**4439**	**4439.0**	<0.01
ia-wiki-Talk	92117	360767	N/A	N/A	N/A	955466	955588.5	949.84	**946074**	946130.7	836.05	946099	**946105.4**	694.19
inf-power	4941	6594	121258	121305.0	443.45	120286	120309.0	75.15	**120267**	**120267.0**	25.61	120284	120285.9	200.54
inf-roadNet-CA	1957027	2760388	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	**57012891**	**57042381.2**	1000.00
inf-roadNet-PA	1087562	1541514	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	**31240366**	**31257976.7**	999.86
inf-road-usa	23947347	28854312	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	**668606851**	**668629969.7**	1000.00
rec-amazon	91813	125704	N/A	N/A	N/A	2623301	2624500.8	968.27	**2572949**	2573546.5	996.53	2572976	**2573190.1**	971.14
rt-retweet	96	117	**1958**	**1958.0**	<0.01	**1958**	**1958.0**	<0.01	**1958**	**1958.0**	<0.01	**1958**	**1958.0**	99.57
rt-retweet-crawl	1112702	2278852	N/A	N/A	N/A	N/A	N/A	N/A	4740416	4741717.4	984.59	**4729503**	**4729765.8**	893.50
rt-twitter-copen	761	1029	**12266**	**12266.0**	1.92	**12266**	**12266.0**	0.08	**12266**	**12266.0**	<0.01	**12266**	**12266.0**	0.03
sc-ldoor	952203	20770807	N/A	N/A	N/A	N/A	N/A	N/A	49533919	49540268.9	999.76	**49478677**	**49481350.5**	999.77
sc-msdoor	415863	9378650	N/A	N/A	N/A	N/A	N/A	N/A	22034469	22036997.4	993.61	**22026919**	**22027981.7**	283.93
sc-nasasrb	54870	1311227	N/A	N/A	N/A	2998286	2998914.7	753.58	2981070	2981734.2	857.41	**2979810**	**2980134.8**	891.93
sc-pkustk11	87804	2565054	N/A	N/A	N/A	N/A	N/A	N/A	4871723	4872176.0	974.55	**4869614**	**4870168.9**	928.52
sc-pkustk13	94893	3260967	N/A	N/A	N/A	N/A	N/A	N/A	5175636	**5176003.5**	991.49	**5174834**	5176102.0	877.71
sc-pwtk	217891	5653221	N/A	N/A	N/A	N/A	N/A	N/A	**12104727**	**12106606.2**	996.99	12110553	12112513.8	816.51
sc-shipsec1	140385	1707759	N/A	N/A	N/A	N/A	N/A	N/A	6744672	6747118.4	998.79	**6738288**	**6739918.6**	977.25
sc-shipsec5	179104	2200076	N/A	N/A	N/A	N/A	N/A	N/A	8433634	8436574.8	999.37	**8425150**	**8427230.8**	985.31
soc-BlogCatalog	88784	2093195	N/A	N/A	N/A	N/A	N/A	N/A	1179892	1180040.6	905.93	**1179888**	**1179937.7**	668.48
soc-brightkite	56739	212945	N/A	N/A	N/A	1177737	1178546.3	871.30	**1165159**	**1165219.0**	753.54	1165347	1165394.0	685.73
soc-buzznet	101163	2763066	N/A	N/A	N/A	N/A	N/A	N/A	1739461	1739667.2	983.61	**1739186**	**1739282.9**	807.72
soc-delicious	536108	1365961	N/A	N/A	N/A	N/A	N/A	N/A	4966446	4970722.0	997.65	**4909370**	**4911555.8**	995.75
soc-digg	770799	5907132	N/A	N/A	N/A	N/A	N/A	N/A	6039717	6045964.6	997.97	**5941979**	**5942894.4**	993.23
soc-dolphins	62	159	**1835**	**1835.0**	<0.01	**1835**	**1835.0**	<0.01	**1835**	**1835.0**	<0.01	**1835**	1835.9	<0.01
soc-douban	154908	327162	N/A	N/A	N/A	**516082**	516117.5	677.40	**516082**	**516082.0**	274.61	**516082**	**516082.0**	4.35
soc-epinions	26588	100120	N/A	N/A	N/A	542653	542919.8	390.81	**537932**	**537965.0**	383.37	537947	537979.0	587.77
soc-flickr	513969	3190452	N/A	N/A	N/A	N/A	N/A	N/A	8626171	8631742.6	998.50	**8566304**	**8573852.4**	996.83
soc-flixster	2523386	7918801	N/A	N/A	N/A	N/A	N/A	N/A	5828318	5839038.4	986.50	**5693085**	**5693147.5**	940.40
soc-FourSquare	639014	3214986	N/A	N/A	N/A	N/A	N/A	N/A	5293197	5312948.8	996.85	**5280145**	**5280242.8**	920.50
soc-gowalla	196591	950327	N/A	N/A	N/A	4736205	4736737.3	562.84	4689248	4689905.0	998.12	**4678748**	**4679081.0**	997.62
soc-karate	34	78	**864**	**864.0**	<0.01	**864**	**864.0**	<0.01	**864**	**864.0**	<0.01	**864**	**864.0**	<0.01
soc-lastfm	1191805	4519330	N/A	N/A	N/A	N/A	N/A	N/A	4719564	4728817.4	995.93	**4642190**	**4642222.1**	842.55
soc-livejournal	4033137	27933062	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	**108273340**	**108307151.1**	1000.00
soc-LiveMocha	104103	2193083	N/A	N/A	N/A	N/A	N/A	N/A	2462288	2462828.9	985.17	**2461356**	**2461468.6**	847.52

Computing Platform: All experiments are run on a 4-way Intel Xeon E7-8850 v2 @ 2.30GHz CPU with 1TB RAM server under CentOS 7.2.

Result Reporting Methodology: Each algorithm is executed on each instance 10 times independently within the cutoff time. The cutoff time is set to 1000s, which means each run will be terminated if 1000 seconds is reached. We report the following information: the average weight of the solutions found in all runs (”w(avg)”); the minimum weight among the solutions found in all runs (”w(min)”); and the average run-time to find the best found solution over all successful runs (”time”). If an algorithm failed to find a solution for an instance within the cutoff time, we marked it by ”N/A”.

Table 2: Experiment results on massive graphs (continued)

instance	$\|V\|$	$\|E\|$	MS-ITS			DLSWCC			ILS-VND			FASTWVC		
			w(min)	w(avg)	avg(t)	w(min)	w(avg)	avg(t)	w(min)	w(avg)	avg(t)	w(min)	w(avg)	avg(t)
soc-orkut	2997166	106349209	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	**128859528**	**128886854.5**	1000.00
soc-pokec	1632803	22301964	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	**50254441**	**50298857.8**	1000.00
soc-slashdot	70068	358647	N/A	N/A	N/A	1242765	1243175.0	962.49	**1228753**	**1228793.3**	842.59	1228945	1228979.2	760.22
soc-twitter-follows	404719	713319	N/A	N/A	N/A	**138884**	**138884.0**	204.59	**138884**	**138884.0**	16.86	**138884**	**138884.0**	2.27
soc-wiki-Vote	889	2914	22192	22200.0	1.38	**22191**	**22191.0**	0.83	**22191**	**22191.0**	0.32	**22191**	**22191.0**	14.46
soc-youtube	495957	1936748	N/A	N/A	N/A	N/A	N/A	N/A	8063822	8067762.2	999.07	**8028348**	**8037259.1**	998.44
soc-youtube-snap	1134890	2987624	N/A	N/A	N/A	N/A	N/A	N/A	15241364	15247104.9	997.04	**15171502**	**15173022.5**	339.12
socfb-A-anon	3097165	23667394	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	**22142778**	**22146868.5**	1000.00
socfb-B-anon	2937612	20959854	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	**17878163**	**17880615.8**	999.48
socfb-Berkeley13	22900	852419	N/A	N/A	N/A	1005551	1005984.7	540.13	1003423	1003890.9	822.86	**1003067**	**1003131.3**	561.14
socfb-CMU	6621	249959	292668	292932.0	2582.08	290311	290384.3	90.57	**290150**	**290188.3**	612.62	290175	290203.9	230.99
socfb-Duke14	9885	506437	459579	459599.0	5816.58	448123	448232.8	186.93	447811	447897.8	630.90	**447771**	**447827.8**	398.93
socfb-Indiana	29732	1305757	N/A	N/A	N/A	1368628	1369223.1	953.52	1365619	1366064.7	858.29	**1364177**	**1364426.4**	539.27
socfb-MIT	6402	251230	271482	271536.0	2502.44	269986	270051.7	87.87	**269834**	269861.0	532.82	269837	**269852.5**	109.81
socfb-OR	63392	816886	N/A	N/A	N/A	2105589	2106349.4	990.54	2083687	2084088.7	975.59	**2083450**	**2083567.7**	838.33
socfb-Penn94	41536	1362220	N/A	N/A	N/A	1820549	1822006.8	986.39	1809903	1810734.4	878.76	**1808434**	**1808630.8**	610.95
socfb-Stanford3	11586	568309	509513	509544.0	16658.55	496824	496945.3	244.68	**496524**	496639.2	616.69	496525	**496540.5**	185.79
socfb-Texas84	36364	1590651	N/A	N/A	N/A	1662115	1662664.4	960.25	1647418	1648445.4	910.47	**1646452**	**1646614.7**	599.57
socfb-uci-uni	58790782	92208195	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	**51364132**	**51375184.8**	1000.00
socfb-UCLA	20453	747604	911569	912087.7	24601.55	885721	886016.5	474.22	884097	884399.8	836.60	**883752**	**883797.4**	469.29
socfb-UConn	17206	604867	794985	795207.8	3437.90	773807	774393.3	336.91	773100	773300.4	847.21	**772481**	**772578.6**	254.39
socfb-UCSB37	14917	482215	676264	676731.5	14251.28	656025	656378.0	259.92	655298	655574.4	911.70	**655040**	**655099.4**	279.25
socfb-UF	35111	1465654	N/A	N/A	N/A	1603262	1604208.2	996.43	1594974	1595278.1	867.82	**1593218**	**1593364.4**	501.23
socfb-UIllinois	30795	1264421	N/A	N/A	N/A	1407508	1408412.3	951.50	1404672	1405115.8	880.27	**1403471**	**1403572.2**	578.64
socfb-Wisconsin87	23831	835946	N/A	N/A	N/A	1076027	1076534.7	539.39	1073687	1074190.3	835.48	**1073062**	**1073233.4**	472.75
tech-as-caida2007	26475	53381	N/A	N/A	N/A	198996	199217.6	235.71	**198705**	**198705.0**	25.94	**198705**	**198705.0**	30.24
tech-as-skitter	1694616	11094209	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	**29543813**	**29566128.5**	1000.00
tech-internet-as	40164	85123	N/A	N/A	N/A	313192	313540.9	375.07	**311623**	**311623.0**	79.15	311627	311647.7	95.16
tech-p2p-gnutella	62561	147878	N/A	N/A	N/A	924346	924581.4	795.94	**922551**	**922553.3**	215.86	922552	922556.3	181.03
tech-RL-caida	190914	607610	N/A	N/A	N/A	4209563	4210338.6	271.58	4177927	4179864.7	999.28	**4163391**	**4164087.3**	994.07
tech-routers-rf	2113	6632	44543	44546.5	119.02	44499	44499.5	12.73	**44495**	**44495.0**	1.62	**44495**	44495.7	54.37
tech-WHOIS	7476	56943	126740	126748.1	3654.98	126499	126528.5	75.63	**126496**	**126496.0**	1.94	**126496**	**126496.0**	56.50
web-arabic-2005	163598	1747269	N/A	N/A	N/A	N/A	N/A	N/A	**6549547**	**6550439.4**	991.93	6556962	6557980.4	968.41
web-BerkStan	12305	19500	298685	299224.8	2496.48	289132	289335.7	193.46	**288071**	**288074.2**	49.18	288133	288141.3	582.72
web-edu	3031	6474	79221	79244.6	147.78	78801	78828.6	21.75	**78726**	**78726.0**	49.51	78755	78759.9	327.74
web-google	1299	2773	**27302**	**27302.0**	3.24	**27302**	**27302.0**	0.78	**27302**	**27302.0**	0.03	**27302**	**27302.0**	0.08
web-indochina-2004	11358	47606	403612	403685.4	2439.04	399499	399835.8	164.64	**398589**	**398599.8**	393.55	398611	398616.6	483.63
web-it-2004	509338	7178413	N/A	N/A	N/A	N/A	N/A	N/A	**23771379**	23782872.4	958.23	23773139	**23774933.8**	996.00
web-polblogs	643	2280	13121	13129.4	3.68	**13107**	**13107.0**	0.03	**13107**	**13107.0**	<0.01	**13107**	**13107.0**	0.02
web-sk-2005	121422	334419	N/A	N/A	N/A	3140399	3140551.6	72.81	**3125177**	**3125623.2**	994.60	3125546	3125835.9	949.87
web-spam	4767	37375	130287	130322.4	1264.10	129946	129958.1	63.30	**129928**	**129928.0**	18.48	129932	129937.2	104.15
web-uk-2005	129632	11744049	N/A	N/A	N/A	N/A	N/A	N/A	**7561840**	**7561854.7**	42.52	7562177	7562205.2	187.92
web-webbase-2001	16062	25593	145748	145889.5	2063.89	144052	144166.2	135.15	143925	143949.8	205.04	**143911**	**143913.5**	522.97
web-wikipedia2009	1864433	4507315	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	**36116079**	**36131824.5**	1000.00

6.3. Comparison with State-of-the-art Algorithms

We compare FastWVC with two state of the art heuristic algorithms for solving MWVC, MS-ITS [20] and DLSWCC [10], as well as a state of the art local search algorithm for solving Minimum Weight Independent Set (MWIS) problem (a complementary problem of MWVC) called ILS-VND [14]. These three competitors, the MS-ITS, DLSWCC and ILS-VND are implemented in C/C++ by their authors. The binary of MS-ITS and codes of DLSWCC are kindly provided by their authors, while the codes of ILS-VND are downloded online.[1]

The results are shown in Tables 1 and 2. Although MS-ITS was effective on three types of instances: SPI, MPI, LPI [20], which means small-scale, middle-scale and large-scale instances, as reported in [10], MS-ITS failed on many massive graphs. Also, since we can only obtain the executable binary but not codes of MS-ITS, we could not set up the cutoff time. So the results for MS-ITS are not under the cutoff time in our experiments. DLSWCC performs much better than MS-ITS, but also failed to find a solution within the cutoff time for many instances . FastWVC outperformed MS-ITS and DLSWCC on almost all instances. ILS-VND worked much better than MS-ITS and DLSWCC, but was still worse than FastWVC overall. More specifically, FastWVC outperformed ILS-VND on about half of the 102 instances, and only inferior to ILS-VND on 26% of instances. In general, FastWVC is superior to MS-ITS, DLSWCC and ILS-VND on massive sparse graphs, in terms of both solution quality and computational time.

6.4. Effectiveness of ConstructWVC

In this section, we study the effectiveness of the ConstructWVC by comparing the performance of ConstructWVC with its one-pass version. The one-pass version, denoted as ConstructWVC$_1$, generates the initial vertex cover just one time, while ConstructWVC generates several vertex covers and chooses the best one as the initial vertex cover.

We performed both algorithms 10 times on each instance, and the results are shown in Table 3 and 4. The first two columns indicate the average weight and the minimum weight of the initial vertex cover found by these two functions, and the column *avg(t)* indicates the average running time.

The results show that ConstructWVC performed better than ConstructWVC$_1$ on 61 and 68 instances w.r.t. the average and best solution quality performance, respectively. As for the running time, except for several massive instances, ConstructWVC did not increase too much time consumption.

Furthermore, we studied whether the initial solution has an impact on the final solution. We conducted FastWVC and its alternative version FastWVC$_0$. The only difference between these two algorithms is that FastWVC$_0$ uses ConstructWVC$_1$ in the initial vertex cover construction phase, while FastWVC uses ConstructWVC. We summarize the comparison results between FastWVC$_0$ and FastWVC here, and do not give the detailed experimental results. The results show that in all 102 instances, FastWVC

[1] https://sites.google.com/site/nogueirabruno/software/ils-mwis.tar.gz

outperformed $FastWVC_0$ on 64 and 66 instances in terms of the average and best solution quality performance. Further observations show that, these instances for which FastWVC yields better results than $FastWVC_0$, are among the instances for which ConstructWVC obtains better initial vertex covers than those of $ConstructWVC_1$. This indicates the contribution of the ConstructWVC procedure to the FastWVC algorithm. This experiment also shows that, for massive graphs, when the local search starts from a better initial vertex cover, it is more likely to find a better solution.

Table 3: Results of ConstructWVC and $ConstructWVC_1$

instance	$\|V\|$	$\|E\|$	ConstructWVC			$ConstructWVC_1$		
			w(min)	w(avg)	avg(t)	w(min)	w(avg)	avg(t)
bio-celegans	453	2025	**14446**	**14512.6**	<0.01	14515	14515	<0.01
bio-diseasome	516	1188	**15324**	**15328.1**	<0.01	15342	15342	<0.01
bio-dmela	7393	25569	**155425**	**155597.8**	0.04	155658	155658	<0.01
bio-yeast	1458	1948	**25495**	**25535.7**	<0.01	25573	25573	<0.01
ca-AstroPh	17903	196972	**652213**	**652563.9**	0.26	652873	652873	<0.01
ca-citeseer	227320	814134	**7088569**	**7089268.5**	1.21	7092662	7092662	0.02
ca-coauthors-dblp	540486	15245729	**27192259**	**27193660.4**	19.12	27232796	27232796	0.11
ca-CondMat	21363	91286	**689864**	**690152.9**	0.14	690155	690155	<0.01
ca-CSphd	1882	1740	**30530**	30544.3	<0.01	30541	**30541**	<0.01
ca-dblp-2010	226413	716460	6689866	6690657.7	1.21	**6688859**	**6688859**	0.02
ca-dblp-2012	317080	1049866	**9066525**	**9067480.8**	1.97	9070583	9070583	0.03
ca-Erdos992	6100	7515	**27096**	27108.3	<0.01	**27096**	**27096**	<0.01
ca-GrQc	4158	13422	**123869**	**124011.2**	0.02	124123	124123	<0.01
ca-HepPh	11204	117619	**372283**	372402.9	0.13	372374	**372374**	<0.01
ca-hollywood-2009	1069126	56306653	49031985	49032743.1	67.33	**49031564**	**49031564**	0.38
ca-MathSciNet	332689	820644	**7818921**	**7819756.6**	1.70	7825591	7825591	0.03
ca-netscience	379	914	**11599**	11600.7	<0.01	**11599**	**11599**	<0.01
ia-email-EU	32430	54397	**49324**	**49340.2**	0.05	49367	49367	<0.01
ia-email-univ	1133	5451	**34133**	**34206.9**	<0.01	34585	34585	<0.01
ia-enron-large	33696	180811	**707392**	**707601.7**	0.20	707935	707935	<0.01
ia-enron-only	143	623	**5105**	5126.8	<0.01	**5105**	**5105**	<0.01
ia-fb-messages	1266	6451	**33880**	**33984.9**	<0.01	34032	34032	<0.01
ia-infect-dublin	410	2765	16540	16588.9	<0.01	**16512**	**16512**	<0.01
ia-infect-hyper	113	2196	**5451**	**5531.4**	<0.01	5590	5590	<0.01
ia-reality	6809	7680	**4439**	**4439**	<0.01	**4439**	**4439**	<0.01
ia-wiki-Talk	92117	360767	**999389**	**999684.3**	0.38	1000603	1000603	<0.01
inf-power	4941	6594	**126460**	**126681.1**	0.02	126685	126685	<0.01
inf-roadNet-CA	1957027	2760388	**58585102**	**58592209.9**	7.29	58641969	58641969	0.12
inf-roadNet-PA	1087562	1541514	**32535874**	**32540530.5**	3.94	32566343	32566343	0.07
inf-road-usa	23947347	28854312	**668964067**	**668988654.3**	85.38	669116462	669116462	1.49
rec-amazon	91813	125704	**2694690**	**2695800.6**	0.29	2703863	2703863	<0.01
rt-retweet	96	117	**2090**	**2090**	<0.01	**2090**	**2090**	<0.01
rt-retweet-crawl	1112702	2278852	**4861763**	**4861763**	4.52	**4861763**	**4861763**	0.06
rt-twitter-copen	761	1029	**12962**	12969.8	<0.01	**12962**	**12962**	<0.01
sc-ldoor	952203	20770807	**49589199**	**49589901.5**	27.21	49608461	49608461	0.12
sc-msdoor	415863	9378650	**22084175**	**22085467.7**	12.13	22094751	22094751	0.06
sc-nasasrb	54870	1311227	**3029252**	**3030187.6**	1.52	3030396	3030396	<0.01
sc-pkustk11	87804	2565054	**4916106**	**4916106**	3.02	**4916106**	**4916106**	0.01
sc-pkustk13	94893	3260967	**5229162**	**5229162**	3.86	**5229162**	**5229162**	0.02
sc-pwtk	217891	5653221	**12280594**	**12282604.8**	6.78	12283287	12283287	0.03
sc-shipsec1	140385	1707759	6989036	6992263.3	2.30	**6984295**	**6984295**	0.01
sc-shipsec5	179104	2200076	**8816948**	8820596.6	3.03	8817803	**8817803**	0.02
soc-BlogCatalog	88784	2093195	**1237127**	**1237882**	2.20	1238677	1238677	0.02
soc-brightkite	56739	212945	**1224647**	**1225164.7**	0.26	1226767	1226767	<0.01
soc-buzznet	101163	2763066	**1834391**	1835156.2	3.34	1834968	**1834968**	0.03
soc-delicious	536108	1365961	**5261622**	**5264177.9**	2.07	5269011	5269011	0.03
soc-digg	770799	5907132	**6222667**	**6224983.2**	7.42	6228670	6228670	0.09
soc-dolphins	62	159	**1840**	**1840**	<0.01	**1840**	**1840**	<0.01
soc-douban	154908	327162	**518539**	518555.1	0.40	**518539**	**518539**	<0.01
soc-epinions	26588	100120	**565448**	**565733.9**	0.12	567196	567196	<0.01
soc-flickr	513969	3190452	**8892148**	**8892972.5**	4.14	8901650	8901650	0.06
soc-flixster	2523386	7918801	**5741200**	**5741410.6**	9.05	5741959	5741959	0.13
soc-FourSquare	639014	3214986	**5350403**	**5351272.2**	4.51	5359433	5359433	0.05
soc-gowalla	196591	950327	**4903718**	**4906161.6**	1.34	4913493	4913493	0.02
soc-karate	34	78	**887**	**895.4**	<0.01	899	899	<0.01
soc-lastfm	1191805	4519330	**4741459**	**4742182.2**	5.64	4742994	4742994	0.08
soc-livejournal	4033137	27933062	**109748624**	**109751947.1**	54.99	109916372	109916372	0.64
soc-LiveMocha	104103	2193083	**2612601**	**2612601**	2.67	**2612601**	**2612601**	0.03

6.5. *Effectiveness of The New Exchange Step*

In this subsection , we evaluated the effectiveness of the new exchange step, which suggests removing two vertices instead of just one in the vertex removing phase. As

Table 4: Results of ConstructWVC and ConstructWVC$_1$ (continued)

instance	$\|V\|$	$\|E\|$	ConstructWVC			ConstructWVC$_1$		
			w(min)	w(avg)	avg(t)	w(min)	w(avg)	avg(t)
soc-orkut	2997166	106349209	**130226871**	**130240678.2**	163.026	130361414	130361414	1.57
soc-pokec	1632803	22301964	**52618441**	**52636603.1**	37.13	52695846	52695846	0.44
soc-slashdot	70068	358647	**1304269**	**1304635.7**	0.42	1306240	1306240	<0.01
soc-twitter-follows	404719	713319	**141032**	**141032**	1.05	**141032**	**141032**	0.01
soc-wiki-Vote	889	2914	**23505**	23566.2	<0.01	**23505**	**23505**	<0.01
soc-youtube	495957	1936748	**8499289**	**8500775.1**	3.34	8504818	8504818	0.04
soc-youtube-snap	1134890	2987624	**15967951**	**15969135.2**	5.52	15978324	15978324	0.08
socfb-A-anon	3097165	23667394	25061703	25072855.5	41.55	**25017688**	**25017688**	0.54
socfb-B-anon	2937612	20959854	20222943	20232390.7	36.74	**20194581**	**20194581**	0.50
socfb-Berkeley13	22900	852419	**1032723**	**1032723**	0.91	**1032723**	**1032723**	<0.01
socfb-CMU	6621	249959	**298480**	**298633**	0.25	298696	298696	<0.01
socfb-Duke14	9885	506437	**459054**	**459054**	0.48	**459054**	**459054**	<0.01
socfb-Indiana	29732	1305757	**1399968**	**1399968**	1.44	**1399968**	**1399968**	<0.01
socfb-MIT	6402	251230	**278253**	**278253**	0.26	**278253**	**278253**	<0.01
socfb-OR	63392	816886	**2167213**	**2168732**	0.95	2174075	2174075	<0.01
socfb-Penn94	41536	1362220	**1864409**	**1864409**	1.58	**1864409**	**1864409**	<0.01
socfb-Stanford3	11586	568309	**508872**	**509455.2**	0.56	509778	509778	<0.01
socfb-Texas84	36364	1590651	**1694107**	**1694107**	1.79	**1694107**	**1694107**	0.01
socfb-uci-uni	58790782	92208195	**51583996**	51584124.1	210.19	51584036	**51584036**	3.14
socfb-UCLA	20453	747604	**910166**	**910166**	0.78	**910166**	**910166**	<0.01
socfb-UConn	17206	604867	**793217**	**793217**	0.59	**793217**	**793217**	<0.01
socfb-UCSB37	14917	482215	**674585**	**674585**	0.47	**674585**	**674585**	<0.01
socfb-UF	35111	1465654	**1637651**	**1637651**	1.58	**1637651**	**1637651**	0.01
socfb-UIllinois	30795	1264421	**1441683**	**1441683**	1.36	**1441683**	**1441683**	<0.01
socfb-Wisconsin87	23831	835946	**1103432**	**1103432**	0.85	**1103432**	**1103432**	<0.01
tech-as-caida2007	26475	53381	**208678**	**208725**	0.07	208830	208830	<0.01
tech-as-skitter	1694616	11094209	**31375664**	**31379545.9**	17.81	31384428	31384428	0.20
tech-internet-as	40164	85123	**328644**	**328743.2**	0.13	328835	328835	<0.01
tech-p2p-gnutella	62561	147878	**935090**	**935245.8**	0.22	935514	935514	<0.01
tech-RL-caida	190914	607610	**4437938**	**4439585.6**	1.05	4447833	4447833	0.02
tech-routers-rf	2113	6632	**46307**	**46367.1**	0.02	46433	46433	<0.01
tech-WHOIS	7476	56943	**130541**	**130727.3**	0.06	131177	131177	<0.01
web-arabic-2005	163598	1747269	**6628147**	**6628147**	2.29	**6628147**	**6628147**	0.01
web-BerkStan	12305	19500	**300329**	**300608.5**	0.05	301229	301229	<0.01
web-edu	3031	6474	**82875**	83000.8	0.01	82979	**82979**	<0.01
web-google	1299	2773	**27879**	**27891.8**	<0.01	27916	27916	<0.01
web-indochina-2004	11358	47606	**405620**	405929.7	0.07	405760	**405760**	<0.01
web-it-2004	509338	7178413	**23830820**	**23830820**	9.62	**23830820**	**23830820**	0.04
web-polblogs	643	2280	**13624**	**13646.9**	<0.01	13701	13701	<0.01
web-sk-2005	121422	334419	**3240553**	**3241324.1**	0.53	3242621	3242621	<0.01
web-spam	4767	37375	**136146**	**136334.1**	0.04	136355	136355	<0.01
web-uk-2005	129632	11744049	**7564548**	**7564548**	12.59	7564549	7564549	0.05
web-webbase-2001	16062	25593	**147394**	**147474.4**	0.06	147476	147476	<0.01
web-wikipedia2009	1864433	4507315	**37403142**	**37405146.1**	12.69	37437560	37437560	0.16

mentioned in Section 4, this strategy is designed for massive sparse graphs. We compared FastwWVC with two alternative algorithms $\mathrm{FastWVC}_1$ and $\mathrm{FastWVC}_2$. These two algorithms work the same as FastWVC except that $\mathrm{FastWVC}_1$ removes one vertex in the vertex removing phase and $\mathrm{FastWVC}_2$ removes three. The comparison results of FastWVC and $\mathrm{FastWVC}_1$ and $\mathrm{FastWVC}_2$ are recorded in Tables 5 and 6 . Note that we only report the instances on which the three algorithms find different solution qualities.

Table 5: Comparison results of FastWVC, $\mathrm{FastWVC}_1$ and $\mathrm{FastWVC}_2$

instance	FastWVC			$\mathrm{FastWVC}_1$			$\mathrm{FastWVC}_2$		
	w(min)	w(avg)	avg(t)	w(min)	w(avg)	avg(t)	w(min)	w(avg)	avg(t)
ca-AstroPh	643007	643011.2	418.77	644716	644946.2	923.01	**643000**	**643005.1**	533.97
ca-citeseer	**7016968**	**7018567.5**	993.79	7041226	7041818.8	973.33	7020178	7021632.3	960.65
ca-coauthors-dblp	**27111167**	**27112456.6**	589.15	27153236	27155296.1	956.19	27112928	27114165.4	323.31
ca-CondMat	679102	679108.3	393.15	681813	681958.4	309.90	**679084**	**679089.4**	685.68
ca-dblp-2010	**6605746**	**6606471.6**	996.66	6633063	6633824.1	958.62	6608252	6609178.4	992.35
ca-dblp-2012	**8953523**	**8956561**	997.49	8983963	8985445.8	973.20	8956849	8958055.6	994.52
ca-HepPh	**365470**	365475.2	297.90	366647	366713.6	705.15	365471	**365472.2**	586.25
ca-hollywood-2009	48930484	48934214.2	1000.00	48991371	48997538.1	972.12	**48885785**	**48893078.5**	999.23
ca-MathSciNet	**7614812**	7619427.4	997.14	7658474	7659193.6	981.08	7616796	**7617979.1**	994.40
ia-enron-large	**692150**	**692161.3**	613.55	695426	695599.7	628.65	692163	692165.3	814.20
ia-wiki-Talk	946099	946105.4	694.19	951144	951425.4	906.30	**946088**	**946104.7**	788.43
inf-roadNet-CA	57012891	57042381.2	1000.00	56873128	57002680.1	1000.00	**56696721**	**56784355.7**	1000.00
inf-roadNet-PA	31240366	31257976.7	999.86	**31096046**	**31218453.8**	998.02	31214598	31250325.5	999.95
inf-road-usa	668606851	668629969.7	1000.00	668596499	668655028.6	1000.00	**668429187**	**668479678.6**	1000.00
rec-amazon	**2572976**	**2573190.1**	971.14	2599715	2600931.3	970.41	2573682	2574164.8	992.08
rt-retweet-crawl	4729503	4729765.8	893.50	4759339	4760277.1	35.98	**4729192**	**4729438.1**	981.80
sc-ldoor	**49478677**	49481350.5	999.77	49523049	49528247.5	980.54	49478929	**49480274.6**	761.07
sc-msdoor	**22026919**	**22027981.7**	283.93	22041463	22042092	933.93	22031058	22031891.7	126.84
sc-nasasrb	**2979810**	**2980134.8**	891.93	2990602	2990995.9	472.11	2980281	2980596.9	886.05
sc-pkustk11	**4869614**	**4870168.9**	928.52	4874214	4874446	918.07	4870350	4870672	929.90
sc-pkustk13	**5174834**	**5176102**	877.71	5194547	5194992.1	534.18	5176605	5177131.6	889.80
sc-pwtk	**12110553**	**12112513.8**	816.51	12112445	12114896.3	947.22	12118043	12121128	994.24
sc-shipsec1	**6738288**	**6739918.6**	977.25	6787059	6788849	919.14	6740405	6742112.1	968.99
sc-shipsec5	**8425150**	**8427230.8**	985.31	8489312	8492829.2	984.06	8427395	8429188.8	985.36
soc-BlogCatalog	**1179888**	**1179937.7**	668.48	1187103	1187290.6	722.02	1179919	1179950.4	713.77
soc-brightkite	**1165347**	**1165394**	685.73	1172415	1172559.8	785.22	1165359	1165415.8	849.14
soc-buzznet	**1739186**	**1739282.9**	807.72	1751055	1751379.2	798.68	1739227	1739283	827.86
soc-delicious	**4909370**	**4911555.8**	995.75	4943773	4945450.6	871.55	4912816	4914340.3	987.92
soc-digg	**5941979**	**5942894.4**	993.23	5980802	5981686.1	255.69	5942519	5943287	979.85
soc-epinions	**537947**	537979	587.77	540630	540868.4	402.77	537957	**537973.5**	607.85
soc-flickr	**8566304**	8573852.4	996.83	8615748	8617927.5	952.57	8566883	**8569511.9**	998.35
soc-flixster	5693085	5693147.5	940.40	5718898	5720033.1	17.31	**5693002**	**5693079**	915.30
soc-FourSquare	**5280145**	**5280242.8**	920.50	5314168	5316013.3	15.87	5280258	5280363.8	942.12
soc-gowalla	**4678748**	**4679081**	997.62	4716347	4717217.7	967.46	4680528	4681059.4	966.03
soc-lastfm	**4642190**	4642222.1	842.55	4665739	4667050.1	20.53	4642194	**4642211.9**	788.63
soc-livejournal	108273340	108307151.1	1000.00	108387512	108477764.9	1000.00	**107823867**	**107886992.6**	1000.00
soc-LiveMocha	**2461356**	**2461468.6**	847.52	2479201	2479572.1	387.12	2461433	2461511	941.08
soc-orkut	128859528	128886854.5	1000.00	128896871	128959752.5	1000.00	**128529097**	**128579169.6**	1000.00
soc-pokec	50254441	50298857.8	1000.00	**49827108**	**50099416**	1000.00	50070489	50121847	1000.00
soc-slashdot	1228945	**1228979.2**	760.22	1236199	1236541.9	891.91	**1228906**	1228982.6	852.18
soc-youtube	**8028348**	8037259.1	998.44	8087844	8092021.3	980.55	8028493	**8033043.5**	998.45
soc-youtube-snap	15171502	15173022.5	339.12	15229397	15253055.6	993.46	**15152226**	**15163110.5**	677.96

Overall, FastWVC and $\mathrm{FastWVC}_2$ find better solutions on most instances than $\mathrm{FastWVC}_1$, indicating that removing two or three vertices is better than just removing one vertex in each step for sparse graphs. On the other hand, FastWVC and $\mathrm{FastWVC}_2$ are competitive with each other, but FastWVC is slightly better. In terms of best found solution, FastWVC performs better on 40 instances, while $\mathrm{FastWVC}_2$ performs better on 34 instances. We then compare their run time. For the average running time, FastWVC outperformed $\mathrm{FastWVC}_2$ on 60 instances. Taking into account both solution quality and time consumption, FastWVC is better than $\mathrm{FastWVC}_2$.

From the results, we can see that compared with removing just one vertex, the larger

Table 6: Comparison results of FastWVC, FastWVC_1 and FastWVC_2 (continued)

instance	FastWVC			FastWVC_1			FastWVC_2		
	w(min)	w(avg)	avg(t)	w(min)	w(avg)	avg(t)	w(min)	w(avg)	avg(t)
socfb-A-anon	22142778	**22146868.5**	1000.00	22194430	22197007.5	845.94	**22133288**	22152286.3	1000.00
socfb-B-anon	17878163	17880615.8	999.48	17944445	17945907.7	503.66	**17870143**	**17875291.6**	999.30
socfb-Berkeley13	**1003067**	**1003131.3**	561.14	1009566	1009830.1	477.66	1003103	1003159.1	554.19
socfb-CMU	290175	290203.9	230.99	291663	291720.2	349.28	**290155**	**290186.9**	309.55
socfb-Duke14	447771	447827.8	398.93	450048	450195.2	441.87	**447766**	**447810.3**	395.50
socfb-Indiana	**1364177**	1364426.4	539.27	1372867	1373107.4	746.55	1364231	**1364340.1**	722.30
socfb-MIT	269837	269852.5	109.81	271278	271356.6	124.09	**269834**	**269838.3**	394.55
socfb-OR	**2083450**	**2083567.7**	838.33	2098596	2099017.4	531.05	2083524	2083609.1	868.34
socfb-Penn94	**1808434**	**1808630.8**	610.95	1821512	1821736.9	707.34	1808485	1808745.8	808.35
socfb-Stanford3	496525	496540.5	185.79	499608	499772.3	191.16	**496522**	**496533.4**	351.33
socfb-Texas84	1646452	1646614.7	599.57	1657317	1657614.9	644.31	**1646309**	**1646457.4**	753.32
socfb-uci-uni	51364132	51375184.8	1000.00	51487637	51503733.2	1000.00	**51329084**	**51351103.2**	1000.00
socfb-UCLA	883752	883797.4	469.29	889413	889528.4	483.86	**883680**	**883762.3**	453.22
socfb-UConn	**772481**	772578.6	254.39	776528	777021.7	536.42	772521	**772573.6**	593.86
socfb-UCSB37	**655040**	**655099.4**	279.25	659083	659235.8	399.55	655075	655102.9	593.72
socfb-UF	1593218	1593364.4	501.23	1603705	1604106.2	670.48	**1593129**	**1593252.6**	634.39
socfb-UIllinois	1403471	1403572.2	578.64	1412764	1412947.3	593.60	**1403407**	**1403553.6**	774.98
socfb-Wisconsin87	**1073062**	1073233.4	472.75	1079729	1080182.1	647.01	1073132	**1073195.9**	527.65
tech-as-skitter	29543813	29566128.5	1000.00	29680644	29789038.7	1000.00	**29494138**	**29509244.2**	999.89
tech-internet-as	311627	311647.7	95.16	312846	312896.2	792.62	**311624**	**311624**	618.25
tech-p2p-gnutella	**922552**	**922556.3**	181.03	927363	927810	2.65	922557	922558.5	127.54
tech-RL-caida	**4163391**	**4164087.3**	994.07	4198002	4198720.5	947.21	4164818	4165711.2	990.44
web-arabic-2005	**6556962**	**6557980.4**	968.41	6594380	6594914.7	988.43	6559878	6561040	763.73
web-BerkStan	288133	288141.3	582.72	289282	289510.3	889.27	**288110**	**288115.8**	742.49
web-indochina-2004	**398611**	398616.6	483.63	400517	400706.9	823.09	**398611**	**398616.1**	665.16
web-it-2004	23773139	23774933.8	996.00	23810934	23812156.7	934.71	**23772688**	**23773428.6**	543.54
web-sk-2005	**3125546**	**3125835.9**	949.87	3161423	3162309.2	985.16	3126168	3126484.6	975.03
web-uk-2005	**7562177**	**7562205.2**	187.92	7564358	7564410.5	907.54	7562179	7562208.7	115.51
web-wikipedia2009	36116079	36131824.5	1000.00	36286180	36385350.6	999.88	**36061563**	**36079360.5**	987.14

search range resulted by removing two vertices can improve solution quality. This is because removing more vertices means more vertices can be taken into consideration when choosing adding vertices, increasing the possibility of generating a better solution. However, although removing three vertices can make the search area even larger, FastWVC_2 did not perform better than FastWVC, which may be due to the increased overhead in each iteration.

6.6. *Effectiveness of The Cost-effective Vertex Selection Strategy*

This subsection is to verify the effectiveness of the cost-effective vertex selection strategy. As described in Section 4, when selecting adding vertices, we search in the closed neighborhood of the two newly removed vertices. To show how much this strategy contributes to the performance of FastWVC, we made a comparison between FastWVC and an alternative algorithm FastWVC_3, which chooses adding vertices by scanning all vertices not in the current candidate solution. Experiment results are shown in Tables 7 and 8 . We only report the instances on which the two algorithms returns different quality solutions.

The results show that the performance of FastWVC is significantly better than FastWVC_3. From the aspect of the best performance and the average performance, in all 102 instances, more than 70 solutions found by FastWVC are better than those found by FastWVC_3, and about 20 solutions found by them are the same. From the aspect of the average running time, FastWVC outperformed FastWVC_3 on 90 instances, moreover, FastWVC spent less than half of the time spent by FastWVC_3 on 29 instances. The results are consistent with our conjecture, that is, FastWVC can reduce

Table 7: Comparison results of FastWVC and FastWVC3

instance	FastWVC			$FastWVC_3$		
	w(min)	w(avg)	avg(t)	w(min)	w(avg)	avg(t)
ca-citeseer	**7016968**	**7018567.5**	993.79	7024113	7024682.8	588.62
ca-coauthors-dblp	**27111167**	**27112456.6**	589.15	27155186	27158263.4	1000.00
ca-dblp-2010	**6605746**	**6606471.6**	996.66	6618405	6618900.3	745.38
ca-dblp-2012	**8953523**	**8956561**	997.49	8961363	8963439	997.91
ca-hollywood-2009	**48930484**	**48934214.2**	1000.00	48993531	48996195.8	1000.00
ca-MathSciNet	**7614812**	**7619427.4**	997.14	7645516	7646637.8	590.41
ia-wiki-Talk	**946099**	**946105.4**	694.19	946412	946486.7	940.85
inf-roadNet-CA	**57012891**	**57042381.2**	1000.00	58194217	58204066	1000.00
inf-roadNet-PA	**31240366**	**31257976.7**	999.86	31968325	31984022.8	1000.00
inf-road-usa	**668606851**	**668629969.7**	1000.00	668930334	668955760.9	1000.00
rec-amazon	**2572976**	**2573190.1**	971.14	2586498	2588415.5	991.47
rt-retweet-crawl	**4729503**	**4729765.8**	893.50	4744591	4745931.6	993.95
sc-ldoor	**49478677**	**49481350.5**	999.77	49515346	49518140.1	1000.00
sc-msdoor	**22026919**	**22027981.7**	283.93	22032036	22033292.5	998.78
sc-nasasrb	**2979810**	**2980134.8**	891.93	2981957	2982574.4	941.05
sc-pkustk11	**4869614**	**4870168.9**	928.52	4875716	4876338.8	790.64
sc-pkustk13	**5174834**	**5176102**	877.71	5182608	5184234.1	968.66
sc-pwtk	**12110553**	**12112513.8**	816.51	12147216	12152164.1	997.77
sc-shipsec1	**6738288**	**6739918.6**	977.25	6777157	6783384.1	991.65
sc-shipsec5	**8425150**	**8427230.8**	985.31	8506220	8518455.7	998.27
soc-BlogCatalog	**1179888**	**1179937.7**	668.48	1180051	1180134.8	907.94
soc-brightkite	**1165347**	**1165394**	685.73	1165667	1165819.4	944.52
soc-buzznet	**1739186**	**1739282.9**	807.72	1740096	1740492	977.14
soc-delicious	**4909370**	**4911555.8**	995.75	4959414	4962941.8	999.29
soc-digg	**5941979**	**5942894.4**	993.23	5987597	5990709.4	995.94
soc-flickr	**8566304**	**8573852.4**	996.83	8612989	8614241.3	374.38
soc-flixster	**5693085**	**5693147.5**	940.40	5699309	5700160.3	984.81
soc-FourSquare	**5280145**	**5280242.8**	920.50	5289756	5291003.7	992.34
soc-gowalla	**4678748**	**4679081**	997.62	4703181	4708001.7	997.38
soc-lastfm	**4642190**	**4642222.1**	842.55	4644949	4645958.7	982.87
soc-livejournal	**108273340**	**108307151.1**	1000.00	109266118	109320007.7	1000.00
soc-LiveMocha	**2461356**	**2461468.6**	847.52	2462581	2463007.9	978.52
soc-orkut	**128859528**	**128886854.5**	1000.00	129701389	129783228.2	1000.00
soc-pokec	**50254441**	**50298857.8**	1000.00	51238688	51420236.2	1000.00
soc-slashdot	**1228945**	**1228979.2**	760.22	1229241	1229348.4	963.78
soc-youtube	**8028348**	**8037259.1**	998.44	8088892	8092088	847.16
soc-youtube-snap	**15171502**	**15173022.5**	339.12	15181469	15232887.7	1000.00
socfb-A-anon	**22142778**	**22146868.5**	1000.00	23356281	23487555	1000.00

Table 8: Comparison results of FastWVC and FastWVC3 (continued)

instance	FastWVC			$FastWVC_3$		
	w(min)	w(avg)	avg(t)	w(min)	w(avg)	avg(t)
socfb-B-anon	**17878163**	**17880615.8**	999.48	18463174	18580581.8	1000.00
socfb-OR	**2083450**	**2083567.7**	838.33	2084141	2084480.9	897.14
socfb-Penn94	**1808434**	**1808630.8**	610.95	1808835	1808941.9	822.94
socfb-Texas84	**1646452**	**1646614.7**	599.57	1646664	1646841.6	816.50
socfb-uci-uni	**51364132**	**51375184.8**	1000.00	51563097	51567783.2	1000.00
socfb-UF	**1593218**	**1593364.4**	501.23	1593477	1593580.2	767.10
socfb-UIllinois	**1403471**	**1403572.2**	578.64	1403578	1403724.9	715.09
socfb-Wisconsin87	**1073062**	**1073233.4**	472.75	1073181	1073366.9	670.38
tech-as-skitter	**29543813**	**29566128.5**	1000.00	30213857	30372644.8	1000.00
tech-RL-caida	**4163391**	**4164087.3**	994.07	4181319	4186871.5	998.31
web-arabic-2005	**6556962**	**6557980.4**	968.41	6568314	6569498.7	590.57
web-it-2004	**23773139**	**23774933.8**	996.00	23787300	23791873.2	996.50
web-sk-2005	**3125546**	**3125835.9**	949.87	3131312	3132625	950.72
web-wikipedia2009	**36116079**	**36131824.5**	1000.00	36588535	36644916.7	1000.00

the search time while guaranteeing the quality of the solution.

7. Conclusions and Future Work

In this paper, we developed a new local search algorithm for MWVC problem called FastWVC, which works particularly well for massive graphs. A new construction procedure, namely *ConstructWVC*, was proposed to produce a competitive initial candidate solution. A new exchange step was introduced for making the search more effective on massive sparse graphs.

We carried out extensive experiments to compare FastWVC with state-of-the-art algorithms on a board range of benchmarks from real world networks. The results showed that FastWVC performs better on most of the instances. In the future, we would like to further improve the algorithm by low complexity strategies for large graphs.

Acknowledgement

This work is supported by National Natural Science Foundation of China 61502464. Shaowei Cai is also supported by Youth Innovation Promotion Association, Chinese Academy of Sciences.

References

References

[1] Bouamama, S., Blum, C., & Boukerram, A. (2012). A population-based iterated greedy algorithm for the minimum weight vertex cover problem. *Appl. Soft Comput.*, *12*, 1632–1639.

[2] Cai, S. (2015). Balance between complexity and quality: local search for minimum vertex cover in massive graphs. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015* (pp. 747–753).

[3] Cai, S., Su, K., Luo, C., & Sattar, A. (2013). NuMVC: an efficient local search algorithm for minimum vertex cover. *Journal of Artificial Intelligence Research*, *46*, 687–716.

[4] Cai, S., Su, K., & Sattar, A. (2011). Local search with edge weighting and configuration checking heuristics for minimum vertex cover . *Artificial Intelligence*, *175*, 1672–1696.

[5] Chvatal, V. (1979). A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, *4*, 233–235.

[6] Dinur, I., & Safra, S. (2005). On the hardness of approximating minimum vertex cover. *Annals of Mathematics*, *162*, 439–485.

[7] Glover, F. (1989). Tabu search – part i. *ORSA Journal on Computing*, *1*, 190–206.

[8] Jovanovic, R., & Tuba, M. (2011). An ant colony optimization algorithm with improved pheromone correction strategy for the minimum weight vertex cover problem . *Applied Soft Computing*, *11*, 5360–5366.

[9] Katzmann, M., & Komusiewicz, C. (2017). Systematic exploration of larger local search neighborhoods for the minimum vertex cover problem. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.* (pp. 846–852).

[10] Li, R., Hu, S., Zhang, H., & Yin, M. (2016). An efficient local search framework for the minimum weighted vertex cover problem. *Information Sciences*, *372*, 428–445.

[11] Li, Y., Cai, S., & Hou, W. (2017). An efficient local search algorithm for minimum weighted vertex cover on massive graphs. In *Proceedings of 11th International Conference of Simulated Evolution and Learning, SEAL 2017, Shenzhen, China, November 10-13, 2017,* (pp. 145–157).

[12] Ma, Z., Fan, Y., Su, K., Li, C., & Sattar, A. (2016). Random walk in large real-world graphs for finding smaller vertex cover. In *IEEE International Conference on TOOLS with Artificial Intelligence* (pp. 686–690).

[13] Ni, Y. (2012). Minimum weight covering problems in stochastic environments. *Information Sciences*, *214*, 91–104.

[14] Nogueira, B., Pinheiro, R. G. S., & Subramanian, A. (2018). A hybrid iterated local search heuristic for the maximum weight independent set problem. *Optimization Letters*, *12*, 567–583.

[15] Richter, S., Helmert, M., & Gretton, C. (2007). A stochastic local search approach to vertex cover. In *Proceedings of KI-07* (pp. 412–426).

[16] Rossi, R. A., & Ahmed, N. K. (2015). The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*. URL: `http://networkrepository.com`.

[17] Shyu, S. J., Yin, P. Y., & Lin, B. M. T. (2004). An ant colony optimization algorithm for the minimum weight vertex cover problem. *Annals of Operations Research*, *131*, 283–304.

[18] Singh, A., & Gupta, A. K. (2011). A hybrid heuristic for the minimum weight vertex cover problem. *Asia-Pacific Journal of Operational Research (APJOR)*, *23*, 273–285.

[19] Wagner, M., Friedrich, T., & Lindauer, M. (2017). Improving local search in a minimum vertex cover solver for classes of networks. In *2017 IEEE Congress on Evolutionary Computation, CEC 2017, Donostia, San Sebastián, Spain, June 5-8, 2017* (pp. 1704–1711).

[20] Zhou, T., Zhipeng, Wang, Y., Ding, J., & Peng, B. (2016). Multi-start iterated tabu search for the minimum weight vertex cover problem. *Journal of Combinatorial Optimization*, *32*, 368–384.