Improving Local Search for Minimum Weight Vertex Cover by Dynamic Strategies

Shaowei Cai[1,2*], **Wenying Hou**[3], **Jinkun Lin**[1] and **Yuanjie Li**[1,2]
[1]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
[2]School of Computer and Control Engineering, University of Chinese Academy of Sciences
[3]School of Information Technology and Management, University of International Business and Economics
caisw@ios.ac.cn

Abstract

The minimum weight vertex cover (MWVC) problem is an important combinatorial optimization problem with various real-world applications. Due to its NP hardness, most works on solving MWVC focus on heuristic algorithms that can return a good quality solution in reasonable time. In this work, we propose two dynamic strategies that adjust the behavior of the algorithm during search, which are used to improve a state of the art local search for MWVC named FastWVC, resulting in two local search algorithms called DynWVC1 and DynWVC2. Previous MWVC algorithms are evaluated on graphs with random or hand crafted weights. In this work, we evaluate the algorithms on the vertex weighted graphs that obtained from an important real world problem, the map labeling problem. Experiments show that our algorithm obtains better results than previous algorithms for MWVC and maximum weight independent set (MWIS) on these real world instances. We also test our algorithms on massive graphs studied in previous works, and show significant improvements there.

1 Introduction

The minimum weight vertex cover problem (MWVC), an extension of the minimum vertex cover problem (MVC), is a well-known combinatorial optimization problem. Given a graph G, a *vertex cover* is a subset of vertices that contains at least one incident vertex of each edge. The MVC problem is to find a vertex cover with the minimum size. When extended to MWVC, each vertex in G has a positive weight, and the task is to find a vertex cover with the smallest total weight. The MWVC problem has been applied to various real-world problems. In particular, with the rise of dynamic digital maps, the dynamic map labeling problem has attracted more and more attention [Been *et al.*, 2006; 2010; Liao *et al.*, 2016; Barth *et al.*, 2016]. This problem can be naturally encoded as the Maximum Weight Independent Set problem (MWIS), and can also be directly solved by MWVC algorithms, as vertex cover and independent set are two complementary concepts in graphs.

MVC is an NP-hard problem. Moreover, it is NP-hard to be approximated within any factor smaller than 1.3606 [Dinur and Safra, 2005]. Therefore, it is generally not efficiently solvable. Most studies on MVC focus on heuristic algorithms, which return solutions of acceptable quality in reasonable time. Heuristic MVC algorithms are usually based on local search, including the widely acknowledged state of the art algorithm NuMVC [Cai *et al.*, 2013]. Recently, there is increasing interest in solving MVC on massive graphs. Since the introduction of FastVC [Cai, 2015], several local search algorithms have been proposed for solving MVC on massive graphs, and the latest state of the art is established by improved versions of FastVC [Cai *et al.*, 2017].

Most studies on solving MWVC are devoted to heuristic algorithms to find near-optimal solutions in reasonable time. The ideas used in these algorithms include ant colony [Shyu *et al.*, 2004; Tuba and Jovanovic, 2009; Jovanovic and Tuba, 2011], simulated annealing [Voß and Fink, 2012] and asymmetric game [Tang *et al.*, 2017]. A tabu search algorithm named Multi-Start Iterated Tabu Search (MS-ITS) [Zhou *et al.*, 2016] achieved state-of-the-art performance on a broad range of small and middle sized benchmarks. As for solving MWVC on massive graphs, two recent local search algorithms DLSWCC [Li *et al.*, 2016] and FastWVC [Li *et al.*, 2017] made significant improvements, where FastWVC performs better.

MWVC are related to Maximum Weight Clique Problem (MWCP) on the complementary graphs. Recently, there has been significant progress in solving MWCP on sparse massive graphs [Wang *et al.*, 2016; Cai and Lin, 2016; Jiang *et al.*, 2017]. But these algorithms may not be efficient to find an MWVC in these graphs, because finding the MWVC is equivalent to finding the MWC in the very dense complement graphs, which also motivates dedicated approaches for MWVC.

Although it is interesting to evaluate MWVC algorithms on massive graphs, a drawback of the current evaluations of MWVC algorithms is that the weights are hand crafted or randomly generated, and do not have real meanings in

*Corresponding author

applications. Indeed, this drawback has been pointed out in evaluations of MWCP algorithms [McCreesh *et al.*, 2017]. It is more desirable to evaluate MWVC algorithms on real application benchmarks, where both the graphs and weights are transferred from real world problems. In this work, we evaluate our algorithms on instances arisen from the map labeling problem — an important problem in digital maps. To show the robustness of our algorithms, we also evaluate them on massive sparse graphs used in previous works.

Our algorithm is built upon a baseline local search algorithm that is abstracted and modified from FastWVC [Li *et al.*, 2017], which maintains the validity of the candidate solution at the end of each iteration. To improve the algorithm, we propose two dynamic strategies, both of which are used in choosing vertex to be removed. The first one combines two heuristics that use different scoring functions. This is different from previous algorithms, which use only one scoring function during the search. The cooperation of the two heuristics are controlled by a parameter that counts the number of non-improving steps w.r.t. solution quality. By using this dynamic strategy, we develop a new local search algorithm named DynWVC1. Experiments show that DynWVC1 outperforms previous heuristic MWVC algorithms on both map labeling graphs and massive graphs.

The second idea is a dynamic strategy for deciding the number of removed vertices. Previous studies mainly remove one vertex in the removing phase of each local search step, FastWVC suggests to remove two vertices. This strategy produces more uncovered edges and thus makes the search region larger. As it turns out, this two-removal strategy has a significant contribution to the good performance of FastWVC. However, the degrees of vertices vary a lot, and removing two vertices might still be not sufficient if the two selected vertices have small degrees. We suggest to remove one more vertex if the total degree of the two removed vertices is not large enough. This idea is related to ideas of exploring large neighborhoods, including k-swap in local search for unweighted vertex cover [Katzmann and Komusiewicz, 2017], and $(k-1,k)$-swap for unweighted independent set [Andrade *et al.*, 2012]. However, in our strategy, this is decided dynamically, according to a criterion on the total degree of removed vertices. We improve DynWVC1 using this strategy, leading to the DynWVC2 algorithm. DynWVC2 further improves DynWVC1 on massive sparse graphs and is currently the best MWVC algorithm on these sparse graphs.

The reminder of this paper is organized as follows. Section 2 presents basic definitions. Section 3 presents the baseline algorithm. Section 4 introduces the dynamic strategy that combines two heuristics to choose vertex to remove, as well as the DynWVC1 algorithm. Section 5 explains the dynamic strategy that determines the number of vertices to remove, and presents the DynWVC2 algorithm. Experiment results are shown in Section 6. Section 7 gives concluding remarks.

2 Preliminaries

2.1 Basic Definitions and Notation

For a weighted graph $G = (V, E)$, V is the set of vertices, E is the set of edges, and each vertex $v \in V$ has a positive weight $w(v)$. Each edge consists of two vertices, which are called *endpoints* of the edge. Two vertices are neighbors if they belong to the same edge, $N(v) = \{u \in V | (u, v) \in E\}$ is the set of neighbors of v, and $degree(v) = |N(v)|$. For a vertex set S, $N(S) = \bigcup_{v \in S} N(v)$.

A *candidate solution* is a subset of V. An edge is *covered* by a candidate solution if at least one of its endpoints belongs to it. The current candidate solution is denoted as C, and we use $w(C)$ to denote the total weight of vertices in C. Further, we use $s_v = \{1, 0\}$ to denote the state of a vertex. If $v \in C$, then $s_v = 1$, and we call v a covering vertex. If $v \notin C$, then $s_v = 0$ and we call v an uncovering vertex. The age of a vertex, denoted as $age(v)$, is the number of steps that have happened since v last changed its state.

A *vertex cover* of a graph is a subset of V that covers all edges, and 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. The MWVC problem is to find a vertex cover C^* such that $w(C^*)$ is minimum, which is equivalent to the Maximum Weight Independent Set (MWIS) problem, i.e., seeking for an independent set with the largest weight.

2.2 Scoring Functions

As with most local search algorithms for MVC and MWVC, our algorithms also use an edge weighting mechanism. Each edge $e \in E$ is associated with a positive number $edge\_w(e)$ as its weight. The cost of C, denoted by $cost(C)$, is the total weight of edges uncovered by C.

The change of *cost* caused by changing the state of a vertex v is denoted as $dscore(v)$, i.e.,

$$dscore(v) = cost(C) - cost(C'),$$

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

Two scoring functions $gain$ and $loss$ are used to measure how much contribution (or damage) a vertex will make to the solution by changing its state, considering both edge weighting mechanism and vertex weights. They are formally defined as

$$gain(v) = \frac{dscore(v)}{w(v)}, v \notin C$$

$$loss(v) = \frac{|dscore(v)|}{w(v)}, v \in C$$

Most local search algorithms for MWVC keep the candidate solution valid at the end of each step. Suppose in a step, a vertex v is removed, making some edges uncovered, then some vertices are added to cover those edges to keep the candidate solution valid. Based on this observation, a scoring function for vertices in C is as follows [Zhou *et al.*, 2016].

$$valid\_score(v) = \sum_{u \in V_r(v)} w(u) - w(v), \text{for } v \in C$$

where $V_r(v)$ denotes the set of adjacent uncovering vertices of v.

The BMS (Best from Multiple Selection) heuristic is used to choose a good-quality element from a large set [Cai, 2015]. It randomly picks k elements and returns the best one w.r.t. some criterion. Our algorithms utilize the BMS heuristic, and parameter k is always set to 50, which guarantees the returned vertex is among the top 10% w.r.t. the criterion with a probability greater than 0.99, as proved in [Cai, 2015].

2.3 Map Labeling Problem

Map labeling is a research area in cartography and computational geometry. Components in a graph such as geographic places and points of interests should be labeled in order to be meaningful to the users. Map labeling involves the selection and placement of labels in maps and one constraint is that two labels should not overlap each other. The problem of eliminating label conflicts and maximizing the importance of labels displayed can be modeled as the Maximum Weight Independent Set (MWIS) problem [Barth *et al.*, 2016]. More specifically, for static graphs, each label can be viewed as a vertex, each vertex is assigned a weight based on its importance, there is an edge between two vertices if they overlap each other. For dynamic graphs, the set of labels displayed changes over time based on operations such as zooming, panning and rotating. Therefore, $[a, b]_l$, which denotes label l displays in time interval $[a, b]$, is viewed as a vertex, each vertex is assigned a weight based on the weight of its label and the length of its time interval. Two vertices are connected by an edge if they conflict with each other. The purpose of maximizing the overall display time of labels can be viewed as finding an independent set that has the maximum weight, which is equal to finding a vertex cover with the minimum weight.

3 Baseline Algorithm

In this section, we introduce a baseline algorithm. We propose a local search framework for MWVC, and then present the important heuristics adopted in our baseline algorithm.

3.1 A New Local Search Framework for MWVC

We introduce a local search algorithmic framework for MWVC (Algorithm 1). This framework is abstracted from FastWVC and is modified to make it more efficient.

An initial vertex cover *C* is firstly generated by a procedure *Construct*. Then the algorithm enters the local search process. Each step consists of a removing phase and an adding phase. In removing phase, vertices are removed by the *RemoveVertices* function (line 5). In adding phase, vertices are added to *C* until it becomes a vertex cover again. After that, redundant vertices are detected and removed (line 10). If the newly found solution C is better than the best found solution C^*, C^* is updated to *C*. At the end of the algorithm, C^* is returned.

A remark on the way to add vertices: All uncovered edges are incident to at least one removed vertex in the removing phase of this step. So, in order to make C valid again, the vertices to be added must be in the set $N(R)$, where R is the set of removed vertices in this step. In our algorithm, adding vertices are selected from $N(R)$.

Algorithm 1: A Local Search Framework for MWVC

```
   Input: A vertex weighted graph, the cutoff time
   Output: A vertex cover of G
1  begin
2      C ← Construct();
3      C* ← C;
4      while elapsed_time < cutoff do
5          RemoveVertices(C);
6          R ← {the vertices removed in this step};
7          while some edge is uncovered by C do
8              choose a vertex v from N(R);
9              C ← C ∪ {v};
10         remove redundant vertices from C;
11         if w(C) < w(C*) then C* ← C;
12     return C*;
```

3.2 Construct Function

Our baseline algorithm employs the construct function in FastWVC. Specifically, the function consists of two phases: a repeated extending phase and a shrinking phase. The extending phase generates k vertex covers and returns the best one according to BMS heuristic. Each vertex cover is constructed by scanning an uncovered edge each time and adding the endpoint with the greater $gain$ value. Different random scanning orders are used so that the k vertex covers are different, and the best one among them is handed to the shrinking phase, where redundant vertices are removed.

3.3 RemoveVertices Function

Another important function in the local search framework is the *RemoveVertices* function. For our baseline algorithm, we adopt the one in FastWVC. Two vertices are removed. Firstly, a vertex with the minimum $loss$ value is selected and removed from C. Then, the second vertex to remove is selected by using the BMS heuristic w.r.t. the $loss$ value.

4 DynWVC1 Algorithm

We propose two dynamic strategies to improve the baseline algorithm, both strategies focus on the *RemoveVertices* function. This section presents the first strategy, as well as the resulting algorithm improved by it (called DynWVC1).

4.1 Choosing Scoring Functions Dynamically

Vertex selection heuristic is critical in local search algorithms for MWVC. To select vertices to remove from *C*, various scoring functions have been proposed. Two effective scoring functions are *loss* and *valid_score* introduced in Section 2.

The *loss* and *valid_score* functions have essentially different impact on the behavior of the algorithm. Vertex selection using $loss$ function is an "exploratory" selection; in other words, there is a good chance that such a selected vertex is good for solution quality, but we can not be certain of that. Different from "exploratory" vertex selection, *valid_score* is a "deterministic" one, that is, we can know exactly whether the removal of a vertex will have a positive effect on solution quality. For example, if the *valid_score* value of a vertex is negative, it means that, if we remove this vertex and add

Algorithm 2: DynamicChoose($no\_improve$, α)

Input: parameter $no\_improve$ and α
Output: a vertex to be removed

```
1  begin
2      if no_improve < α then
3          choose a vertex u with minimum valid_score from C,
           breaking ties in favor of the oldest one;
4      else
5          choose a vertex u with small loss from C according to
           BMS strategy,breaking ties in favor of the oldest one;
6          no_improve ← 0;
7      return u;
```

its adjacent uncovering vertices, we can get a vertex cover with lower cost than the current one. However, just like other deterministic heuristics, local search using *valid_score* alone easily gets trapped in local optima.

We propose a dynamic vertex selection strategy, which consists of a primary vertex scoring function and a secondary scoring function. The primary function is *valid_score* and the secondary is *loss*. We put the implementation of this strategy into a function called *DynamicChoose* (Algorithm 2), which is used to select the second removing vertex. The parameter *no_improve* denotes the number of iterations in which the *valid_score* function fails to decrease the weight of vertex cover. If *no_improve* is smaller than a predetermined parameter α, a vertex with the smallest *valid_score* will be removed; if *no_improve* achieves α, which means *valid_score* loses its effectiveness for a good number of steps, then the secondary function *loss* is activated to pick the vertex according to BMS heuristic, in the same way as FastWVC. In both cases, ties are broken by preferring the oldest vertex.

The *no_improve* is initialized as 0 at the beginning of the algorithm. When the solution found in this iteration is no better than the solution in last iteration, it shows that the removing vertex selected by *valid_score* does not contribute well to the solution quality, then *no_improve* is increased by 1. Each time using the secondary function, *no_improve* is reset to 0, so that *DynamicChoose* switches back to the main scoring function.

The intuition of this strategy is to make *valid_score* and *loss* complement each other. The *valid_score* chooses the most valuable removed vertex in the current step, and when it loses effectiveness, the *loss* function guides the search to a new area and creates more valuable vertices for *valid_score* to select.

4.2 Description of DynWVC1 Algorithm

We use the *DynamicChoose* function to improve the baseline algorithm, leading to a local search algorithm named DynWVC1 (Algorithm 3). In the beginning, the initial vertex cover C is generated by the construct function as shown in the baseline algorithm. Edge weights are all initialized as 1. Then, $gain$ of vertices in $V \setminus C$, $loss$ and $valid\_score$ of vertices in C are calculated. Parameter $no\_improve$ is set to 0. We use C^* and C' to denote the best found solution and the solution found in last iteration of local search.

In the main loop, each iteration consists of a removing phase and an adding phase. In the removing phase, two vertices are removed from C. Firstly, a vertex w with the minimum $loss$ is first removed from C. Then, the second vertex u is selected by *DynamicChoose* function introduced in Section 4.1. After the removal of w and u, some of their incident edges may become uncovered.

Algorithm 3: DynWVC1

Input: A weighted graph $G = (V, E, W)$, *cutoff*, α
Output: A vertex cover of G

```
1  begin
2      C ← Construct();
3      C*, C' ← C;
4      for each e ∈ E, edge_w(e) ← 1;
5      calculate gain, loss and valid_score of vertices;
6      no_improve ← 0;
7      while elapsed_time < cutoff do
8          choose a vertex w with minimum loss from C,
           breaking ties in favor of the oldest one;
9          C ← C \ {w};
10         u ← DynamicChoose(no_improve, α);
11         C ← C \ {u};
12         R ← {w, u};
13         while some edge is uncovered by C do
14             choose a vertex v with maximum gain from
               N(R), breaking ties in favor of the oldest one;
15             C ← C ∪ {v};
16             edge_w(e) ← edge_w(e) + 1 for each uncovered
               edge;
17             remove redundant vertices from C;
18         if w(C) < w(C*) then C* ← C;
19         if w(C') ≤ w(C) then no_improve++;
20         C' ← C;
21     return C*;
```

In the adding phase, vertices are added into C until C becomes a vertex cover. As explained in the baseline algorithm, we only need to consider the vertices in $N(R) = N(w) \cup N(u)$. Specifically, while C is not yet a vertex cover, a vertex $v \in N(R)$ with the maximum $gain$ is chosen and added to C, breaking ties by preferring the oldest one (line 14-15). Then weights of all uncovered edges are increased by 1. At the end of each adding operation, redundant vertices whose neighbors are all covering vertices are removed. Note that this is different from previous algorithms, which remove redundant vertices by scanning all vertices in C after all adding operations are done. In fact, only neighbors of the adding vertex may become redundant vertices, and thus we only need to check the neighbors of the vertex just added.

If C is better than C^*, C^* is updated to C. If the solution found in this step is no better than the solution in last step, i.e., C is no better than C', $no\_improve$ is increased by 1. Finally, C' is set to C. At the end of the algorithm, the best found solution C^* is returned.

5 DynWVC2 Algorithm

In most local search algorithms for MVC and MWVC, one vertex is removed in the removing phase, FastWVC [Li *et*

Instance	$\|V\|$	$\|E\|$	PLS_WIS $w_{max}(w_{avg})$	MSITS $w_{max}(w_{avg})$	DLSWCC $w_{max}(w_{avg})$	FastWVC $w_{max}(w_{avg})$	DynWVC1 $w_{max}(w_{avg})$
alabama-AM2	1164	19386	154265(152437.6)	174017(174017)	173969(173883.1)	**174297(174291.9)**	174243(174217.6)
alabama-AM3	3504	309664	150548(146911.9)	184135(184135)	182322(181655.3)	185484(185411.9)	**185590(185521.4)**
columbia-AM1	2500	24651	193904(193779.6)	192626(192626)	196418(196391.3)	196466(196460.1)	**196475(196475)**
columbia-AM2	13597	1609795	160892(159906.7)	184140(184140)	191755(191127.3)	208964(208931.1)	**208973(208949.7)**
columbia-AM3	46221	27729137	161950(159907.1)	N/A(N/A)	N/A(N/A)	223870(223367.9)	**225494(225068.7)**
florida-AM2	1254	16936	211590(209854.3)	228849(228849)	230537(230478.3)	**230595**(230594.4)	**230595(230595)**
florida-AM3	2985	154043	180223(176676.6)	236460(236460)	236111(235650)	**237283(237268.6)**	237271(237098.7)
georgia-AM3	1680	74126	187605(185334.9)	221354(221354)	221531(221255.3)	222651(222646.9)	**222652(222652)**
greenland-AM3	4986	3652361	11940(11496.2)	N/A(N/A)	12110(12110)	14003(13990.5)	**14012(14009.4)**
hawaii-AM2	2875	265158	109338(107426.7)	125007(125007)	124206(123476.3)	125244(125223.5)	**125278(125272.9)**
hawaii-AM3	28006	49444921	100163(98118.4)	N/A(N/A)	N/A(N/A)	138015(137680.7)	**140718(140666.3)**
idaho-AM3	4064	3924080	70018(68488.9)	N/A(N/A)	74530(74530)	77123(77115.6)	**77145(77145)**
kansas-AM3	2732	806912	74702(72976.2)	87812(87812)	87584(86638.2)	87951(87944.8)	**87976(87974.8)**
kentucky-AM2	2453	643428	87058(86615)	**97397(97397)**	95725(95497.1)	**97397(97397)**	**97397(97397)**
kentucky-AM3	19095	59533630	82387(79291.7)	N/A(N/A)	N/A(N/A)	99432(99335.7)	**100474(100462.1)**
louisiana-AM3	1162	37077	53182(52575)	59922(59922)	59404(59351.1)	**60024(60024)**	60005(60005)
maryland-AM3	1018	95415	40480(39928.6)	**45496(45496)**	45400(45359.7)	**45496**(45495.2)	**45496(45496)**
massachusetts-AM2	1339	35449	135506(134827.9)	140071(140071)	140035(140022.2)	**140095(140095)**	**140095(140095)**
massachusetts-AM3	3703	551491	128952(127317.1)	145108(145108)	142638(142602.8)	145773(145744)	**145865(145850.3)**
mexico-AM3	1096	47131	90383(89059.4)	97571(97571)	97382(97296.4)	97660(97654.8)	**97663(97663)**
new-hampshire-AM3	1107	18021	103494(101645.5)	116002(116002)	114721(114574)	**116060(116060)**	**116060**(116057.6)
north-carolina-AM3	1557	236739	45877(45239.5)	49652(49652)	49095(48781.8)	**49720(49719.5)**	**49720**(49709)
oregon-AM2	1325	57517	149883(148166.3)	**165047(165047)**	164834(164824.2)	**165047(165047)**	**165047(165047)**
oregon-AM3	5588	2912701	131420(130517.2)	174200(174200)	164953(164928.2)	174912(174873.5)	**175064(175049.8)**
pennsylvania-AM3	1148	26464	117980(116745.1)	143867(143867)	142914(142760.2)	**143870(143870)**	**143870(143870)**
rhode-island-AM2	2866	295488	159295(157745.7)	183624(183624)	183870(183556.6)	**184596(184591.2)**	184545(184532.7)
rhode-island-AM3	15124	12622219	145333(142756.2)	N/A(N/A)	N/A(N/A)	199223(198937.3)	**201477(201433.8)**
utah-AM3	1339	42872	84397(83214.7)	98628(98628)	98230(98171)	**98847(98845.6)**	98802(98799.8)
vermont-AM3	3436	1136164	53411(52764.5)	63195(63195)	61225(60728.7)	62909(62868.7)	**63254(63236.8)**
virginia-AM2	2279	60040	243531(241229.3)	295055(295055)	295690(295634.2)	**295867(295867)**	295847(295672.2)
virginia-AM3	6185	665903	231241(227065.2)	301612(301612)	299523(297636.6)	307644(307589.3)	**307903(307805.3)**
washington-AM2	3025	152449	248720(243684.5)	302004(302004)	304914(304759.1)	305574(305565.1)	**305619(305614)**
washington-AM3	10022	2346213	230433(223661.6)	305342(305342)	297413(297258.7)	312767(312664.8)	**313693(313626)**
west-virginia-AM3	1185	125620	44895(44540.5)	47826(47826)	46991(46880.3)	47881(47881)	**47927(47927)**

Table 1: Experiment results on map labeling benchmark. **We report the weight of the returned independent set, the larger the better.**

al., 2017] proposes to remove two vertices at one iteration to enlarge the search region. It is an effective strategy especially for massive sparse graphs. However, the degrees of vertices vary a lot, the search region produced by removing two vertices is not always sufficient for the search process. From the overall point of view, removing two vertices is an effective method. But from the view of each step, the search region produced by it may be still not enough for some steps.

To make the algorithm more efficient on massive sparse graphs, we incorporate a new strategy into DynWVC1 for deciding the number of removed vertices, resulting in the DynWVC2 algorithm. In DynWVC2, the first two removing vertices are selected and removed in the same way as DynWVC1. After removing two vertices, if the total degree of the removed vertices does not reach a predetermined value (which is set to 2 times average degree of the graph), then the algorithm removes one more vertex with small *loss* based on BMS strategy. In this way, when the search region produced by removing two vertices is too small, which limits our options in the adding phase, it can be expanded by removing one more vertex. If the search region produced by removing two vertices is large enough, to get a balance between search time and search quality, we do not remove the third one.

6 Experiments

We evaluate our algorithms on map labeling instances and massive graphs, compared with state of the art algorithms.

6.1 Experiment Preliminaries

Setup: DynWVC1 and DynWVC2 are implemented in C++, and compiled by g++ with '-O3' option. The parameter in our algorithms α is set to 5. All experiments are run on a 4-way Intel Xeon E7-8850 v2 @ 2.30GHz CPU with 1TB RAM server under CentOS 7.2.

Competitors: For the map labeling benchmark, we compare with three local search algorithms for MWVC namely MS-ITS [Zhou *et al.*, 2016], DLSWCC [Li *et al.*, 2016] and FastWVC [Li *et al.*, 2017], and a local search algorithm for MWIS namely PLS_MWIS [Pullan, 2009], which is selected as the best MWIS algorithm for the map labeling benchmark [Barth *et al.*, 2016]. For the massive graphs, FastWVC performs significantly better than MS-ITS and DLSWCC [Li *et al.*, 2017], so we only compare with FastWVC. All competitors are implemented by their authors using C++ and complied by g++ with '-O3' option.

Results Reporting Methodology: All algorithms are executed 10 times on each instance independently with a cutoff time of 1000 seconds. We report the following information: the average weight of the solutions found in 10 runs ("w_{avg}"), the best solution found in 10 runs. For map labeling, we treat it as MWIS problem, and report the weight of independent set (the larger the better). For massive graphs MWVC instances, the smaller weight the better. If an algorithm failed to find a solution within time limit, we marked it by "N/A".

Instance	FastWVC $w_{min}(w_{avg})$	DynWVC1 $w_{min}(w_{avg})$	DynWVC2 $w_{min}(w_{avg})$	Instance	FastWVC $w_{min}(w_{avg})$	DynWVC1 $w_{min}(w_{avg})$	DynWVC2 $w_{min}(w_{avg})$
ca-AstroPh	643098(643111.9)	643016(643021.1)	**643010(643013.6)**	soc-livejournal	108354007(108376264.8)	**107109272**(107175170.3)	107119513(**107171075.4**)
ca-citeseer	7071275(7072174.2)	7004514(**7004613**)	**7004443**(7004760.4)	soc-LiveMocha	2462223(2462424.1)	2461311(2461444.8)	**2461134(2461234.3)**
ca-author-dblp	27177143(27178701.2)	**27086884(27087172)**	27087067(27087258.2)	soc-orkut	128907034(128921417.8)	**127809146(127851626.7)**	127852422(127880203.9)
ca-CondMat	679176(679193.5)	679099(679104.7)	**679089(679097.4)**	soc-pokec	50055454(50068176.5)	**49012281(49073735.7)**	49051954(49110504.9)
ca-dblp-2010	6625143(6649380.4)	6594945(**6595074.4**)	**6594885**(6595099.9)	soc-slashdot	1228968(1229005.1)	1228898(1228903.9)	**1228811(1228819.9)**
ca-dblp-2012	9029616(9030774)	**8930269(8930691)**	8930674(8931272.9)	soc-twitter-follow	**138884(138884)**	**138884(138884)**	**138884(138884)**
ca-HepPh	365487(365496.6)	365471(365474.2)	**365469(365469)**	soc-youtube	8160145(8200371.2)	7992738(**7993675.1**)	**7992494**(7994238.9)
ca-holywod-09	49016625(49017602)	48851286(48851930.7)	**48850043(48851087.4)**	soc-youtube-snap	15416414(15418150.1)	**15075956(15077176.3)**	15127993(15129877.9)
ca-MathSciNet	7729148(7730475.5)	7595513(7595824.9)	**7595452(7595753.8)**	socfb-A-anon	22236342(22291292.3)	**22090572(22091793.8)**	22104015(22105054.9)
ia-email-EU	**48447(48447)**	**48447(48447)**	**48447(48447)**	socfb-B-anon	17972088(17975652.3)	**17844202(17852798.5)**	17845087(17863627.5)
ia-enron-large	692227(692234.1)	692153(692157.2)	**692150(692152.9)**	socfb-Berkeley13	1003741(1003868.5)	1003122(1003247.2)	**1003040(1003135.7)**
ia-wiki-Talk	946156(946170.7)	946120(946126.2)	**946081(946091.2)**	socfb-Indiana	1365785(1366024.6)	1364526(1364658.1)	**1364435(1364597.9)**
inf-roadNet-CA	56922074(56947699.5)	55740241(**55765382.3**)	**55730992**(55767520.6)	socfb-OR	2084279(2084490.4)	2083440(2083527.4)	**2083230(2083346.8)**
inf-roadNet-PA	31481091(31486345.1)	**30701298(30705976.4)**	**30701298(30705976.4)**	socfb-Penn94	1810396(1810689.2)	1808768(1808896.5)	**1808672(1808802.3)**
inf-road-usa	668594656(668627425.9)	668032689(**668066207.3**)	**668023952**(668068870.6)	socfb-Stanford3	496577(496657.9)	496514(496525.2)	**496510(496513.1)**
rec-amazon	2574487(2575227.3)	**2571527(2571557.8)**	**2571527(2571557.8)**	socfb-Texas84	1648374(1648625.6)	1646886(1647079.3)	**1646603(1646855.3)**
rt-retweet-crawl	4732837(4734284)	4728891(4728902.8)	**4728868(4728887)**	socfb-uci-uni	51468147(51473502.1)	**51218381(51221409.6)**	51228972(51230733.2)
sc-ldoor	49550246(49551537)	49428447(**49429403.5**)	**49427306**(49430888.2)	socfb-UCLA	884262(884333.8)	883749(883825.7)	**883666(883752.9)**
sc-msdoor	22063493(22064239.4)	22002649(**22003068.6**)	**22001876**(22003364.2)	socfb-UConn	772903(773124.3)	772526(772617.3)	**772468(772573.6)**
sc-nasasrb	2984150(2985056.9)	**2979327(2979409.2)**	2979367(2979464.1)	socfb-UCSB37	655343(655418.8)	655066(655132.8)	**655009(655057.7)**
sc-pkustk11	4876622(4878352.1)	4867503(4867632.2)	**4867392(4867573.5)**	socfb-UF	1594572(1595017.6)	1593400(1593597.5)	**1593376(1593494.8)**
sc-pkustk13	5191545(5192638.6)	5173828(**5174100.7**)	**5173806**(5174114.9)	socfb-UIllinois	1404688(1404889.3)	1403731(1403806.1)	**1403428(1403674.6)**
sc-pwtk	12151340(12155895.5)	**12085553**(12086438.5)	12085826(**12086426.8**)	socfb-Wiconsin87	1073751(1074010.9)	**1073102**(1073312.1)	1073136(**1073210.2**)
sc-shipsec1	6786381(6790277.5)	**6726861(6728121.1)**	6727779(6728873.6)	tech-as-caida2007	198710(198710)	**198705(198705)**	**198705(198705)**
sc-shipsec5	8477677(8484227.7)	8405621(8406534.5)	**8404554(8406087.6)**	tech-as-skitter	29888271(29910900)	**29222525(29223901)**	29386910(29388955.6)
soc-BlogCatlog	1180108(1180119.3)	1179906(1179920.9)	**1179870(1179884.2)**	tech-internet-as	311624(311624)	**311623(311623)**	**311623**(311623.2)
soc-brightkite	1165377(1165401.8)	1165253(1165278.7)	**1165217(1165238.5)**	tech-p2p-gnutella	922568(922570.8)	922552(922553.5)	**922551(922552.4)**
soc-buzznet	1739328(1739422.9)	1738907(1738945)	**1738808(1738871.8)**	tech-RL-caida	4158849(4165235)	4149301(4149603.5)	**4149072(4149207.1)**
soc-delicious	4921290(4927118.5)	4898568(4900311.2)	**4895133(4897942.3)**	web-arabic-2005	6600438(6606173.1)	**6548677(6549252.5)**	6549005(6549322)
soc-digg	5967941(5976630.6)	5938917(5938946.7)	**5938590(5938669.7)**	web-BerkStan	288172(288188.5)	**288144(288156.8)**	288146(288157)
soc-douban	**516082(516082)**	**516082(516082)**	**516082(516082)**	web-indochina-04	398644(398649.1)	**398603**(398609.9)	398606(**398609**)
soc-epinions	537998(538024.7)	537976(537989.2)	**537957(537965.9)**	web-it-2004	23824127(23825659.3)	**23765753(23766387.6)**	23765925(23766802.4)
soc-flickr	8705430(8706624.6)	8539127(8539501.2)	**8537070(8538040)**	web-sk-2005	3127008(3127230.7)	**3124138(3124236.6)**	3124154(3124319.2)
soc-flixster	5693856(5694379.4)	5692954(5692959.7)	**5692945(5692952.7)**	web-uk-2005	7563135(7563202.4)	**7561840(7561840)**	**7561840(7561840)**
soc-FourSquare	5281828(5282156.1)	**5279869(5279912.4)**	5279873(5279919)	web-webbase-01	143922(143927.2)	143918(143921.4)	**143917(143920.7)**
soc-gowalla	4682631(4703023.1)	4675213(4675386.7)	**4674975(4675054.5)**	web-wikipedia09	36460676(36465150.1)	**35717178(35723311.3)**	35747391(35750347.8)
soc-lastfm	4642497(4642613.3)	4642174(4642184)	**4642164(4642169.6)**				

Table 2: Experiment results on massive graphs. **We report the weight of the returned vertex cover, the smaller the better.**

6.2 Experiments on Map Labeling Benchmark

We download the map files for North America from OpenStreetMap[1] and generate conflict graphs from them using the software[2] developed by Barth et al. [Barth *et al.*, 2016], which are used as the benchmark. We consider almost all the POIs[3] to label during the generating procedure. For each map file, there are three resulting conflict graphs corresponding to the Activity Model chosen, namely AM1, AM2 and AM3 [Barth *et al.*, 2016]. We do not report results on instances with fewer than 1000 vertices, which are easy to solve.

Experiment results on these instances are shown in Table 1. DynWVC1 finds better solutions than other algorithms on most instances. Also, DynWVC1 is much faster than other algorithms: its average run time is 219s, while this figure is 498s for PLS_WIS, 593s for MS-ITS, 329s for DLSWCC and 443 for FastWVC. Also, DynWVC2 has almost the same performance as DynWVC1, and is not reported.

6.3 Experiments on Massive Graphs

We also evaluate our algorithms on massive graphs from Network Data Repository [Rossi and Ahmed, 2015]. Many of them have 100 thousands to millions of vertices, and dozens of millions of edges. These instances become popular in recent heuristic works for MWVC. The weight of each vertex is assigned to a value from [20,100] uniformly at random, as adopted in testing DLSWCC and FastWVC [Li *et al.*, 2017].

Table 2 reports results on massive graphs with more than 10,000 vertices. DynWVC1 and DynWVC2 find better solutions than FastWVC on most instances, pushing forward the state of the art in solving MWVC for massive sparse graphs. Additionally, DynWVC2 improves DynWVC1 on these instances. For 37 out of 73 instances, DynWVC2 has better performance than DynWVC1 in terms of both best and averaged solution quality, while DynWVC1 has better performance on 14 instances. For the remained instances, neither of them dominates on both measurements.

6.4 Effectiveness of *DynamicChoose*

To study the effectiveness of the dynamic strategy for choosing scoring function, we compare DynWVC1 with its simplified version that does not use the *DynamicChoose* function, namely SimpWVC. We report the difference between the best solutions and difference between averaged solutions found by DynWVC1 and SimpWVC. We only present results on map labeling instances with more than 1000 vertices and massive graphs with more than 10000 vertices. Experiment results are shown in Table 3 and 4, respectively. Almost all Δ values are negative, indicating that DynWVC1 finds better solutions than SimpWVC on almost all instances. We conclude that the *DynamicChoose* function effectively improves the algorithm.

[1] http://download.geofabrik.de/north-america.html

[2] https://github.com/kit-algo/temporalmaplabeling

[3] https://wiki.openstreetmap.org/wiki/Map_Features

Instance	SimpWVC-DynWVC1		Instance	SimpWVC-DynWVC1	
	Δ_{max}	Δ_{avg}		Δ_{max}	Δ_{avg}
alabama-AM2	-32	-63.4	massachusetts-AM2	-1	-2.8
alabama-AM3	-2331	-2384.3	massachusetts-AM3	-1147	-1177
columbia-AM1	-14	-20.2	mexico-AM3	-166	-205.3
columbia-AM2	-1658	-1665.7	new-hampshire-AM3	-245	-269.7
columbia-AM3	-10308	-10278.2	north-carolina-AM3	-6	-1.1
florida-AM2	-13	-27.7	oregon-AM2	-64	-75
florida-AM3	-534	-490.1	oregon-AM3	-1412	-1664
georgia-AM3	-606	-711.4	pennsylvania-AM3	-349	-430.4
greenland-AM3	-196	-282.9	rhode-island-AM2	-256	-386.8
hawaii-AM2	-533	-583.9	rhode-island-AM3	-9754	-10226.8
hawaii-AM3	-7757	-8111.5	utah-AM3	-241	-263
idaho-AM3	-198	-236.8	vermont-AM3	-983	-1030.1
kansas-AM3	-119	-133.3	virginia-AM2	14	182
kentucky-AM2	-66	-85.4	virginia-AM3	-8157	-8352.5
kentucky-AM3	-2556	-2936.6	washington-AM2	-401	-442.3
louisiana-AM3	-31	-38.2	washington-AM3	-12326	-12777.9
maryland-AM3	-13	-17	west-virginia-AM3	-154	-195.1

Table 3: Results of SimpWVC-DynWVC1 on map labeling

Instance	DynWVC1-SimpWVC		Instance	DynWVC1-SimpWVC	
	Δ_{min}	Δ_{avg}		Δ_{min}	Δ_{avg}
ca-AstroPh	-72	-79.3	soc-livejournal	-602156	-588590.3
ca-citeseer	-7519	-7638.3	soc-LiveMocha	-655	-684.3
ca-coauthors-dblp	-90109	-90921.3	soc-orkut	-559630	-576350.7
ca-CondMat	-85	-85.7	soc-pokec	-452147	-420907.4
ca-dblp-2010	-5150	-5289.3	soc-slashdot	-79	-90.6
ca-dblp-2012	-12685	-13096.9	soc-twitter-follows	0	0
ca-HepPh	-19	-21.8	soc-youtube	-1528	-1607.6
ca-hollywood-09	-165838	-166004.9	soc-youtube-snap	-58603	-112590.3
ca-MathSciNet	-3301	-3330.5	socfb-A-anon	13035	-40526.5
ia-email-EU	0	0	socfb-B-anon	-42	-6404.6
ia-enron-large	-65	-68.3	socfb-Berkeley13	-576	-525.1
ia-wiki-Talk	-36	-35.4	socfb-Indiana	-1090	-1191.8
inf-roadNet-CA	-997607	-980812.8	socfb-OR	-537	-588.3
inf-roadNet-PA	-778138	-781780	socfb-Penn94	-852	-1315.6
inf-road-usa	-219899	-226239.6	socfb-Stanford3	-77	-136.9
rec-amazon	-850	-988.4	socfb-Texas84	-1507	-1470
rt-retweet-crawl	-48	-56.2	socfb-uci-uni	-251521	-253746.9
sc-ldoor	-122090	-123726.9	socfb-UCLA	-381	-431.7
sc-msdoor	-61349	-61790.1	socfb-UConn	-329	-462.5
sc-nasasrb	-2054	-2197.4	socfb-UCSB37	-212	-218.2
sc-pkustk11	-3717	-3932.1	socfb-UF	-1038	-1213.7
sc-pkustk13	-8546	-8981.3	socfb-UIllinois	-708	-945.9
sc-pwtk	-23457	-24073	socfb-Wisconsin87	-760	-697.5
sc-shipsec1	-10851	-10559.3	tech-as-caida2007	-4	-4.3
sc-shipsec5	-11889	-11919.7	tech-as-skitter	-663233	-666506.9
soc-BlogCatalog	-166	-170.6	tech-internet-as	-1	-1
soc-brightkite	-111	-121.1	tech-p2p-gnutella	-14	-15.1
soc-buzznet	-281	-284.7	tech-RL-caida	-616	-508.6
soc-delicious	3515	4002.7	web-arabic-2005	-22007	-23188.2
soc-digg	-879	-923.8	web-BerkStan	-35	-31.9
soc-douban	0	0	web-indochina-2004	-38	-38.6
soc-epinions	-44	-34.9	web-it-2004	-59299	-59603.7
soc-flickr	-546	-1457.6	web-sk-2005	-1881	-1917.7
soc-flixster	-143	-151.4	web-uk-2005	-1833	-1876.3
soc-FourSquare	-1070	-1124.8	web-webbase-2001	-4	-2.1
soc-gowalla	-980	-1053.1	web-wikipedia2009	-745191	-744083.8
soc-lastfm	-106	-106.6			

Table 4: Results of DynWVC1-SimpWVC on massive graphs

7 Conclusions and Future Work

This paper focused on efficient heuristic algorithms for MWVC. We introduced a local search framework, and proposed two dynamic strategies to improve the baseline algorithm, leading to two local search algorithms for MWVC.

We evaluated our algorithms on the instances transferred from the real world map labeling problems and massive sparse graphs. Our algorithms significantly outperform previous heuristic algorithms. We would like to improve our algorithms for more real world problems.

References

[Andrade *et al.*, 2012] Diogo Vieira Andrade, Mauricio G. C. Resende, and Renato Fonseca F. Werneck. Fast local search for the maximum independent set problem. *J. Heuristics*, 18(4):525–547, 2012.

[Barth *et al.*, 2016] Lukas Barth, Benjamin Niedermann, Martin Nöllenburg, and Darren Strash. Temporal map labeling: a new unified framework with experiments. In *Proc. of GIS-2016*, pages 23:1–23:10, 2016.

[Been *et al.*, 2006] Ken Been, Eli Daiches, and Chee-Keng Yap. Dynamic map labeling. *IEEE Trans. Vis. Comput. Graph.*, 12(5):773–780, 2006.

[Been *et al.*, 2010] Ken Been, Martin Nöllenburg, Sheung-Hung Poon, and Alexander Wolff. Optimizing active ranges for consistent dynamic map labeling. *Comput. Geom.*, 43(3):312–328, 2010.

[Cai and Lin, 2016] Shaowei Cai and Jinkun Lin. Fast solving maximum weight clique problem in massive graphs. In *Proc. of IJCAI 2016*, pages 568–574, 2016.

[Cai *et al.*, 2013] Shaowei Cai, Kaile Su, Chuan Luo, and Abdul Sattar. Numvc: An efficient local search algorithm for minimum vertex cover. *J. Artif. Intell. Res.*, 46:687–716, 2013.

[Cai *et al.*, 2017] Shaowei Cai, Jinkun Lin, and Chuan Luo. Finding A small vertex cover in massive sparse graphs: Construct, local search, and preprocess. *J. Artif. Intell. Res.*, 59:463–494, 2017.

[Cai, 2015] Shaowei Cai. Balance between complexity and quality: Local search for minimum vertex cover in massive graphs. In *Proc. of IJCAI-2015*, pages 747–753, 2015.

[Dinur and Safra, 2005] Irit Dinur and Samuel Safra. On the hardness of approximating minimum vertex cover. *Annals of Mathematics*, 162(1):439–485, 2005.

[Jiang *et al.*, 2017] Hua Jiang, Chu-Min Li, and Felip Manyà. An exact algorithm for the maximum weight clique problem in large graphs. In *Proc. of AAAI 2017*, pages 830–838, 2017.

[Jovanovic and Tuba, 2011] Raka Jovanovic and Milan Tuba. An ant colony optimization algorithm with improved pheromone correction strategy for the minimum weight vertex cover problem. *Appl. Soft Comput.*, 11(8):5360–5366, 2011.

[Katzmann and Komusiewicz, 2017] Maximilian Katzmann and Christian Komusiewicz. Systematic exploration of larger local search neighborhoods for the minimum vertex cover problem. In *Proc. AAAI 2017*, pages 846–852, 2017.

[Li *et al.*, 2016] Ruizhi Li, Shuli Hu, Haochen Zhang, and Minghao Yin. An efficient local search framework for the minimum weighted vertex cover problem. *Inf. Sci.*, 372:428–445, 2016.

[Li *et al.*, 2017] Yuanjie Li, Shaowei Cai, and Wenying Hou. An efficient local search algorithm for minimum weighted vertex cover on massive graphs. In *Proc. of SEAL-2017*, pages 145–157, 2017.

[Liao *et al.*, 2016] Chung-Shou Liao, Chih-Wei Liang, and Sheung-Hung Poon. Approximation algorithms on consistent dynamic map labeling. *Theor. Comput. Sci.*, 640:84–93, 2016.

[McCreesh *et al.*, 2017] Ciaran McCreesh, Patrick Prosser, Kyle Simpson, and James Trimble. On maximum weight clique algorithms, and how they are evaluated. In *Proc. of CP-2017*, pages 206–225, 2017.

[Pullan, 2009] Wayne Pullan. Optimisation of unweighted/weighted maximum independent sets and minimum vertex covers. *Discrete Optimization*, 6(2):214–219, 2009.

[Rossi and Ahmed, 2015] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *Proc. of AAAI-2015*, pages 4292–4293, 2015.

[Shyu *et al.*, 2004] Shyong Jian Shyu, Peng-Yeng Yin, and Bertrand M. T. Lin. An ant colony optimization algorithm for the minimum weight vertex cover problem. *Annals OR*, 131(1-4):283–304, 2004.

[Tang *et al.*, 2017] C. Tang, A. Li, and X. Li. Asymmetric game: A silver bullet to weighted vertex cover of networks. *IEEE Transactions on Cybernetics*, PP(99):1–12, 2017.

[Tuba and Jovanovic, 2009] Milan Tuba and Raka Jovanovic. An analysis of different variations of ant colony optimization to the minimum weight vertex cover problem. *Wseas Transactions on Information Science and Applications*, 6(6):3742–56, 2009.

[Voß and Fink, 2012] Stefan Voß and Andreas Fink. A hybridized tabu search approach for the minimum weight vertex cover problem. *J. Heuristics*, 18(6):869–876, 2012.

[Wang *et al.*, 2016] Yiyuan Wang, Shaowei Cai, and Minghao Yin. Two efficient local search algorithms for maximum weight clique problem. In *Proc. of AAAI 2016*, pages 805–811, 2016.

[Zhou *et al.*, 2016] Taoqing Zhou, Zhipeng Lü, Yang Wang, Junwen Ding, and Bo Peng. Multi-start iterated tabu search for the minimum weight vertex cover problem. *J. Comb. Optim.*, 32(2):368–384, 2016.