Two Weighting Local Search for Minimum Vertex Cover

Shaowei Cai[1*] and **Jinkun Lin**[2] and **Kaile Su**[3]
[1]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
[2]Key Laboratory of High Confidence Software Technologies, Peking University, Beijing, China
[3]Institute for Integrated and Intelligent Systems, Griffith University, Brisbane, Australia
shaoweicai.cs@gmail.com; jkunlin@gmail.com

Abstract

Minimum Vertex Cover (MinVC) is a well known NP-hard combinatorial optimization problem, and local search has been shown to be one of the most effective approaches to this problem. State-of-the-art MinVC local search algorithms employ edge weighting techniques and prefer to select vertices with higher weighted score. These algorithms are not robust and especially have poor performance on instances with structures which defeat greedy heuristics. In this paper, we propose a vertex weighting scheme to address this shortcoming, and combine it within the current best MinVC local search algorithm NuMVC, leading to a new algorithm called TwMVC. Our experiments show that TwMVC outperforms NuMVC on the standard benchmarks namely DIMACS and BHOSLIB. To the best of our knowledge, TwMVC is the first MinVC algorithm that attains the best known solution for all instances in both benchmarks. Further, TwMVC shows superiority on a benchmark of real-world networks.

Introduction

A *vertex cover* of a graph is a set of vertices such that each edge of the graph is incident to at least one vertex of the set. The Minimum Vertex Cover (MinVC) problem is to find the minimum sized vertex cover in a graph. MinVC is a prominent NP-hard combinatorial optimization problem with many applications, such as network security, VLSI design and industrial machine assignment. It is also closely related to Maximum Clique (MaxClq) and Maximum Independent Set (MaxIS) problems. Algorithms for MinVC can be directly used to solve the MaxClq problem, which has many applications from computer vision to social networks. Due to their hardness and importance to many real-world applications, even a small progress in solving these three problems can have a significant impact in practice.

MinVC as well as MaxClq and MaxIS are all NP-hard and the associated decision problems are NP-complete (Garey and Johnson 1979). Furthermore, it is NP-hard to approximate MinVC within any factor smaller than 1.3606 (Dinur and Safra 2005); and state-of-the-art approximation algorithms can only achieve an approximation ratio of $2 - o(1)$ (Karakostas 2005). Besides, both MaxClq and MaxIS are not approximable within $|V|^{1-\epsilon}$ for any $\epsilon > 0$, unless NP=P (Zuckerman 2006).

However, the hardness of these problems are based on worst case complexity analysis. In practice, we can develop algorithms to tackle these problems efficiently in most cases. Practical algorithms for these three problems mainly fall into two types: exact ones mainly including branch-and-bound algorithms and heuristic ones mainly including local search algorithms. Exact algorithms guarantee the optimality of the solutions they find, but may fail to give a solution within reasonable time for large instances. Local search algorithms cannot guarantee the optimality of their solutions, but they can find optimal or near-optimal solutions for large and hard instances within reasonable time.

A huge amount of effort has been devoted to local search algorithms for these three problems, e.g., (Battiti and Protasi 2001; Pullan 2006; Richter, Helmert, and Gretton 2007; Andrade, Resende, and Werneck 2008; Cai, Su, and Sattar 2011; Benlic and Hao 2013; Cai et al. 2013), to name a few. Most of the algorithms are designed to solve MinVC or MaxClq. Recently, local search algorithms for MinVC have been shown to be more efficient than those for MaxClq and MaxIS, based on experiments with the standard benchmarks namely DIMACS and BHOSLIB. Especially, the very recent algorithm NuMVC (Cai et al. 2013) dominates other MinVC local search algorithms on both DIMACS and BHOSLIB benchmarks, and significantly outperforms state-of-the-art MaxClq local search algorithms such as PLS[1] (Pullan 2006), with an exception on the `brock` family.

State-of-the-art MinVC local search algorithms, including NuMVC, employ edge weighting techniques and prefer to select vertices with higher weighted score. A disadvantage of such edge-weighting greedy search algorithms is that, they are not robust and have poor performance on certain types of instances (such as those with structures which defeat greedy heuristics).

In this paper, we propose a vertex weighting scheme

*Corresponding author

[1]PLS is a milestone of MaxClq local search, and still competes very well with the most recent MaxClq local search algorithms (Benlic and Hao 2013).

to address this shortcoming. The main idea is to penalize vertices when they are not in the current candidate solution C, and force the algorithm to select those vertices with great penalties to add into C; at the same time, we also decrease the penalty when a vertex is chosen thanks to its penalty, in order to prevent a vertex being selected over and over again because of its penalty (i.e., vertex weight). We combine the vertex weighting scheme within the NuMVC algorithm, leading to a new algorithm called TwMVC (Two weighting local search for Minimum Vertex Cover).

We carry out experiments to compare TwMVC with NuMVC on standard benchmarks including DIMACS and BHOSLIB benchmarks, as well as a real-world benchmark of web link networks. Experimental results show that TwMVC outperforms NuMVC on most DIMACS instances and almost all BHOSLIB instances. As far as we know, TwMVC is the first MinVC algorithm that attains the best known results for all DAIMCS and BHOSLIB instances. Note that it is known that for a single heuristic, it is very difficult to perform well on all these instances (Pullan, Mascia, and Brunato 2011). Furthermore, TwMVC performs better than NuMVC on real-world web link networks.

The following section introduces some necessary background knowledge, and Section 3 reviews NuMVC. We propose the vertex weighting scheme in Section 4 and describe the TwMVC algorithm in Section 5. Experimental evaluations of TwMVC are presented in Section 6. Finally, we give some concluding remarks.

Preliminaries

Basic Definitions and Notation

An undirected graph $G = (V, E)$ consists of a vertex set V and an edge set E where each edge is a 2-element subset of V. A simple graph is an unweighted, undirected graph containing no graph loops or multiple edges. For an edge $e = \{u, v\}$, we say that vertices u and v are the $endpoints$ of edge e. Two vertices are neighbors if and only if they both belong to some edge.

Given a simple graph $G = (V, E)$, a $candidate\ solution$ for MinVC is a subset of vertices. An edge $e \in E$ is $covered$ by a candidate solution X if at least one endpoint of e belongs to X. Usually, MinVC local search algorithms maintain a current candidate solution during the search. For convenience, in the rest of this paper, we use C to denote the current candidate solution. The $state$ of a vertex v is denoted by $s_v \in \{1, 0\}$, such that $s_v = 1$ means $v \in C$, and $s_v = 0$ means $v \notin C$. The age of a vertex is the number of steps since its state was last changed.

In edge weighting local search algorithms for MinVC, each edge $e \in E$ is associated with a non-negative integer number $w(e)$ as its edge weight. Given a candidate solution X, the cost of X, denoted by $cost(X)$, is defined as the total weight of edges uncovered by X. MinVC local search algorithms search for candidate solutions with lower cost. For a vertex v, $score(v) = cost(C) - cost(C')$ where $C' = C \backslash \{v\}$ if $v \in C$, and $C' = C \cup \{v\}$ otherwise, measuring the benefit of changing the state of vertex v.

A Local Search Framework for MinVC

Most local search algorithms for MinVC, including COVER (Richter, Helmert, and Gretton 2007), EWLS (Cai, Su, and Chen 2010), EWCC (Cai, Su, and Sattar 2011) and NuMVC, solve the MinVC problem by iteratively solving its decision version — given a positive integer number k, searching for a k-sized vertex cover. A basic framework for such algorithms is shown in Algorithm 1, as described below.

Algorithm 1: A Local Search Framework of MinVC

```
construct C until it is a vertex cover;
while not reach terminate condition do
    if C covers all edges then
        C* := C;
        remove a vertex from C;
    perform an exchanging step;
return C*;
```

In the beginning, a vertex cover is constructed to initialize the current candidate solution C, usually using a greedy construction procedure. During the search, whenever the algorithm finds a k-sized vertex cover, one vertex is removed from C and the algorithm goes on to search for a $(k-1)$-sized vertex cover. The step to a neighboring candidate solution consists of exchanging a pair of vertices: a vertex $u \in C$ is removed from C, and a vertex $v \notin C$ is put into C. Such a step is called an exchanging step. When the algorithm terminates, it outputs the smallest vertex cover it has found.

Review of NuMVC

This section reviews the NuMVC algorithm. NuMVC adopts the framework in Algorithm 1, and here we only introduce its exchanging step.

We first introduce the configuration checking (CC) strategy, which is an important strategy in NuMVC. Initially introduced in (Cai, Su, and Sattar 2011), the CC strategy aims to avoid cycling during local search. A vertex is configuration changed if and only if after its last removal from C, at least one of its neighboring vertices has changed its state. According to the CC strategy, only configuration changed vertices are allowed to be added into C. It has been proved that for any uncovered edge, at least one of its endpoints is configuration changed (Cai et al. 2013).

NuMVC performs an exchanging step in two stages. First, it picks a vertex $u \in C$ with the greatest $score$ to remove, breaking ties in favor of the oldest one. Then, it chooses a random uncovered edge e and picks one of e's endpoints to add into C as follows: if only one endpoint of e is configuration changed, then that vertex is selected; if both endpoints are configuration changed, then the one with greater $score$ is selected, breaking ties in favor of the older one. At the end of each step, NuMVC increases weights of all uncovered edges by one, and if the averaged weight achieves a threshold, all edge weights are decreased by multiplying a positive real number that is smaller than 1.

Vertex Weighting for MinVC

In this section, we propose a vertex weighting scheme and illustrate its usage in TwMVC. We also explain the differences of the proposed vertex weighting scheme with existing weighting schemes.

The Vertex Weighting Scheme

From an abstract perspective, the vertex weighting scheme in TwMVC includes two folds of meanings: first, it diversifies the search by forcing the algorithm to select those vertices that stay too often outside C to add into C; on the other hand, a vertex should not benefit too many times from its vertex weight. Although we expect this vertex weighting scheme effective for most instances, we conjecture it is particularly effective for instances where some vertices would not be selected for very long time using greedy heuristics. As we will see, there are experimental evidences for this conjecture.

The vertex weighting scheme works as follows. Each vertex v is associated with a non-negative integer number $w_v(v)$ as its weight, which is initialized to 0. At the end of each exchanging step, for each vertex $v \notin C$, $w_v(v)$ is increased by one, to measure the frequency that it is outside C. Vertex weights are also decreased by one periodically. The vertex weights are used and updated if necessary when selecting a vertex to add into C. We note that TwMVC always picks a vertex from an uncovered edge to add into C. For the selected edge, if one endpoint's vertex weight is significantly greater than the other's, then it is chosen into C (when two vertices have similar weights, we ignore their weight difference), and also its vertex weight is decreased using the formula $w_v(v) := w_v(v) \cdot \beta$, where $\beta \in [0, 1]$.

While it is quite intuitive to increase vertex weights for vertices when they are not in C, we here explain the idea in the weight decreasing mechanism. For the sake of diversification, it is reasonable to force the algorithm to pick the vertex with significantly greater weight to add into C. However, if the weight of a vertex gets too high, then the vertex will be selected into C over and over again in a period of time, usually shortly after it is removed from C. This is because for such a high-weight vertex v, it needs some time for its neighbors to increase their weights to match with it. Before that, whenever TwMVC picks an uncovered edge containing v, v will be selected to add into C thanks to its high weight, regardless of its $score$. Thus the algorithm is likely misled during this period of time. In fact, a significant vertex weight can be regarded as a privilege of the vertex, which assures it to be selected into C. In order to avoid a vertex with significant weight to be added over and over again, this privilege should not be used too many times. A solution is thus to decrease the vertex weight of a vertex after it is selected into C via this privilege.

Differences with Previous Weighting Schemes

As a popular form of diversification, weighting techniques have been widely used in local search algorithms for constraint search problems. However, the vertex weighting scheme in this work is different from existing weighting schemes in three important aspects.

First, our vertex weighting scheme decreases weight for only one vertex, while existing weighting schemes each time decrease weights for many elements. Some weighting methods decrease weights for all elements, such as the SAPS scheme for SAT (Hutter, Tompkins, and Hoos 2002) and the edge weighting scheme in NuMVC (Cai et al. 2013); others decrease weights for elements whose weights are greater than one, such as the PAWS scheme for SAT (Thornton et al. 2004) and the weighting scheme in DLS-MC for MaxClq (Pullan and Hoos 2006). *In spirit, previous weighting schemes decrease weights in order to focus on recent weighting decisions, while our vertex weighting scheme decreases weight for a vertex to avoid the vertex using its weight as a privilege too many times*.

Second, our vertex weighting scheme decreases weight when a vertex uses its weight as a privilege to be selected, while previous weighting schemes do so depending on other conditions. For instance, PAWS (Thornton et al. 2004) and DLS-MC (Pullan and Hoos 2006) decrease weights periodically, SAPS decreases weights based on a probability (Hutter, Tompkins, and Hoos 2002), while SWT (Cai and Su 2013) and the weighting scheme in NuMVC do so when the averaged weight reaches a threshold.

Finally, we would like to note that the way we utilize the vertex weights (i.e., focusing on whether two vertices have significant vertex weight difference) is novel.

A recent MinVC algorithm VEWLS, which is modified from NuMVC, also uses edge and vertex weights (Fang et al. 2014). However, the vertex weights are only simply used in the restart phase. Results in (Fang et al. 2014) shows that VEWLS has slight improvement over NuMVC on about 70% of the instances but a little worse on others.

The TwMVC Algorithm

In this section, we develop a local search algorithm called TwMVC, which is obtained from NuMVC by applying the vertex weighting scheme in the preceding section.

We outline the TwMVC algorithm in Algorithm 2. The initialization is trivial: all vertex weights are initialized to 0, all edge weights are initialized to 1, and $scores$ of vertices are computed accordingly; then the current candidate solution C is constructed by adding a vertex with the greatest $score$ iteratively until it becomes a vertex cover.

At each step, TwMVC first picks a vertex $u \in C$ with the greatest $score$ to remove[2], breaking ties in favor of the oldest vertex. Then, it selects a random uncovered edge e, and chooses one of e's endpoints to add into C according to the $chooseAddVertex$ function. At the end of each step, edge weights of uncovered edges are increased by one, and all edge weights greater than one are decreased by one each γ (a parameter) steps. Also, vertex weights of vertices outside C are increased by one, and all vertex weights greater than one are decreased by one each 100 steps.

The $chooseAddVertex$ function (Algorithm 3) in TwMVC chooses a vertex to add into C from an uncovered

[2]Note that in C, a vertex with the greatest $score$ is the one has the minimum absolute value of $score$, as all these scores are negative.

edge . We denote the input edge as e, and its two endpoints as v_{i_1} and v_{i_2}. If only one endpoint of e is configuration changed, then the function returns that configuration changed vertex. Otherwise, the function selects a vertex from v_{i_1} and v_{i_2} depending on whether the two vertices have significant weight difference, or formally, whether $|w_v(v_{i_1}) - w_v(v_{i_2})| > \delta$, where δ is an integer parameter. If this is the case, then the function returns the vertex with greater weight, and also decreases its weight by multiplying a factor $\beta \in (0, 1)$. If not, the function returns the vertex with greater $score$, breaking ties by preferring the older one.

Algorithm 2: TwMVC

```
Input: graph G = (V, E), the cutoff time
Output: vertex cover of G
begin
    step := 0;
    initialize edge weights and vertex weights;
    construct C greedily until it is a vertex cover;
    while elapsed time < cutoff do
        if C covers all edges then
            C* := C;
            remove a vertex with the greatest score from C;
            continue;
        choose a vertex u ∈ C with the greatest score,
        breaking ties in favor of the oldest one;
        C := C\{u};
        choose an uncovered edge e randomly;
        v := chooseAddVertex(e);
        C := C ∪ {v};
        w(e)+=1 for each uncovered edge e;
        if step%γ=0 then for each w(e) > 1, w(e)-=1 ;
        w_v(v)+=1 for each v ∉ C;
        if step% 100=0 then for each w_v(v) >1, w_v(v)-=1;
        step+=1;
    return C*;
end
```

Algorithm 3: chooseAddVertex(e)

```
Input: an uncovered edge e = {v_{i_1}, v_{i_2}}
Output: a vertex
if only one endpoint of e is configuration changed then
    return v* ∈ e such that v* is configuration changed;
if |w_v(v_{i_1}) − w_v(v_{i_2})| > δ then
    v* := v ∈ e with greater vertex weight;
    w_v(v*) := w_v(v*) · β;
    return v*;
else
    return v* ∈ e with greater score, breaking ties in favor
    of the older one;
```

Empirical Results

In this section, we first compare TwMVC with NuMVC on standard benchmarks in the literature, i.e., the DIMACS and BHOSLIB benchmarks. Then, we compare TwMVC and NuMVC on a real-world benchmark of web link networks. Since NuMVC clearly outperforms other MinVC local search algorithms (Cai et al. 2013), we do not compare TwMVC with other MinVC local search algorithms. Finally, we also compare TwMVC with state-of-the-art exact MaxClq solvers.

The Benchmarks

The DIMACS benchmark is taken from the Second DIMACS Implementation Challenge for the Maximum Clique problem (1992-1993)[3]. Thirty seven graphs were selected by the organizers for a summary to indicate the effectiveness of algorithms, comprising the Second DIMACS Challenge Test Problems. These instances were generated from real world problems and random graphs in various models. The DIMACS benchmark remains the most popular benchmark and has been widely used for evaluating MinVC and MaxClq algorithms. Note that as the DIMACS graphs were originally designed for the MaxClq problem, MinVC algorithms are tested on their complementary graphs.

The BHOSLIB[4] (Benchmarks with Hidden Optimum Solutions) instances were generated in the phase transition area according to model RB (Xu et al. 2007). This benchmark is famous for its hardness and has been widely used to evaluate recent local search solvers to MinVC, MaxClq and MaxIS.

The web link networks[5] were generated from web pages online. Here, vertices are web-pages and edges are hyperlinks between pages. Some instances are not simple graphs and thus not included in our experiment. These real-world instances have much larger sizes than those in DIMACS and BHOSLIB benchmarks, and has recently been used in testing MaxClq and Coloring algorithms (Rossi et al. 2013; Rossi and Ahmed 2014; Rossi et al. 2014).

Experimental Preliminaries

TwMVC is programmed in C++, on the basis of source codes of NuMVC[6]. Both algorithms are complied by g++ (version 4.4.5) with the '-O2' option. For NuMVC, we adopt the parameter setting reported in (Cai et al. 2013). The parameters in TwMVC are tuned manually based on experiments, and are set as follows.

Setting β and δ: We tune β by testing values in [0.6,0.9] with step of 0.05 (values outside this interval are essentially worse); δ is tuned similarly. We find that TwMVC is not sensitive to β and δ. For example, when β varies in [0.7,0.9] TwMVC has similar performance on most instances. This makes it easy to find a good setting for these two parameters. For all instances: $\beta = 0.8$ and $\delta = 100000$.

Setting γ: Recall that TwMVC increases edge weights every step but decreases edge weights each γ steps. Thus, it is intuitive that γ should be linear to $|E|$. That is, for instances with more edges, the decreasing delay should be longer so that we have sufficient edge weights to guide the

[3] ftp://dimacs.rutgers.edu/pub/challenges

[4] http://www.nlsde.buaa.edu.cn/~kexu/benchmarks/graph-benchmarks.htm

[5] http://www.graphrepository.com/networks.php

[6] Available from http://lcs.ios.ac.cn/~caisw/MVC.html

search. Experiments suggest that $|E|/k$ is a good form for γ. Finally, γ is set to $|E|/7$ for most instances, except for C instances (where $\gamma = |E|/15$), MANN instances (where $\gamma = |E|/3.3$) and web instances (where $\gamma = |E|/10$).

Although the above setting can already yield good performance on brock instances, we observe a much better setting for brock instances is: $\beta = 1$, $delta = 10000$ and $\gamma = 1500$, which is adopted in our experiment.

Each algorithm is executed 50 independent runs with different random seeds on each instance. The time limit is set to be half an hour for each run. All experiments are carried out on a machine under Linux, using an Intel(R) Core(TM) 2.8 GHz CPU and 4 GB RAM. To execute the DIMACS machine benchmarks[7], this machine requires 0.22 CPU seconds for r300.5, 1.32 CPU seconds for r400.5 and 4.67 CPU seconds for r500.5.

DIMACS Benchmark Results

Experimental results on the DIMACS benchmark are presented in Table 1. Most DIMACS instances are so easy that they are solved by both solvers with 100% success rate within 2 seconds, and thus are not reported.

For each instance we report optimal (or minimum known) vertex cover size ("VC*"); success rate ("suc rate"), i.e., the number of successful runs divided by the number of total runs, where a run is successful if a solution of size VC* is found; and the penalized averaged run time in CPU seconds over all runs ("time"), where run time of a successful run is the time to find the VC^* solution, and that of a failed run is considered to be an hour. If there are no successful runs, the "time" column is marked with "n/a". The results in bold indicate the best performance for an instance.

Table 1: Comparison of NuMVC and TwMVC on the DIMACS benchmark. The VC* column marked with an asterisk means that the minimum known vertex cover size has been proved optimal.

Graph		NuMVC		TwMVC	
Instance	VC^*	suc rate	time	suc rate	time
brock400_2	371*	90%	954	**100%**	**18**
brock400_4	367*	100%	6.17	100%	**0.64**
brock800_2	776*	0%	n/a	**6%**	**3451**
brock800_4	774*	0%	n/a	**50%**	**2156**
C2000.9	1920	0%	n/a	**6%**	**3483**
C4000.5	3982	100%	282	100%	**238**
keller6	3302	100%	4.32	100%	4.41
MANN_a45	690*	100%	115	100%	**85**
MANN_a81	2221	16%	3263	**26%**	**2614**
p_hat1500-1	1488*	100%	**4.36**	100%	6.28

TwMVC outperforms NuMVC on these DIMACS instances, except for keller6 and p_hat1500-1 where the two solvers have similar performance. The performance of TwMVC is significantly better than that of NuMVC on brock instances and the two putatively hardest instances C2000.9 and MANN_a81 (Richter, Helmert, and Gretton 2007; Grosso, Locatelli, and Pullan 2008; Cai, Su, and Sattar 2011; Cai et al. 2013). Remarkably, TwMVC finds a 1920-sized vertex cover (or equivalently a 80-sized clique in the complementary graph) in 6% runs for C2000.9. Note that only a few algorithms can locate a 1920-sized vertex cover (or a clique of 80 vertices for the complementary graph), and even for those that do, they only succeed in one trial out of 100 and usually need more time (Grosso, Locatelli, and Pullan 2008; Pullan, Mascia, and Brunato 2011; Wu, Hao, and Glover 2012; Cai et al. 2013).

There are two families of instances that are very difficult for greedy heuristics, including brock instances where optimal solutions consist of low-degree vertices, and MANN instances which have a large proportion of plateaus in search space. Thanks to the vertex weighting scheme, TwMVC shows significant improvement over NuMVC on brock and MANN instances. Finally, we note that TwMVC succeeds in finding the best known solution for all DIMACS instances, indicating its robustness. Comparatively, NuMVC fails to find the best known solution for 3 instances.

Table 2: Comparison of NuMVC and TwMVC on the BHOSLIB benchmark. All these BHOSLIB instances have a hidden optimal vertex cover, whose size is shown in the VC* column.

Graph		NuMVC		TwMVC	
Instance	VC^*	suc rate	time	suc rate	time
frb53-24-1	1219	68%	1843	**74%**	**1508**
frb53-24-2	1219	100%	284	100%	**243**
frb53-24-3	1219	100%	75	100%	**66**
frb53-24-4	1219	100%	422	100%	**334**
frb53-24-5	1219	100%	75	100%	**68**
frb56-25-1	1344	94%	780	94%	**755**
frb56-25-2	1344	84%	1145	**90%**	**992**
frb56-25-3	1344	100%	175	100%	**138**
frb56-25-4	1344	100%	73	100%	**53**
frb56-25-5	1344	100%	45	100%	**37**
frb59-26-1	1475	76%	1671	**78%**	**1448**
frb59-26-2	1475	22%	3026	**40%**	**2424**
frb59-26-3	1475	82%	1177	**90%**	**881**
frb59-26-4	1475	**60%**	**1879**	46%	2203
frb59-26-5	1475	100%	97	100%	**91**

BHOSLIB Benchmark Results

Experimental results on the BHOSLIB benchmark are shown in Table 2. We adopt the same report methodology used for the DIMACS benchmark. For concentrating on hard instances, we only present the three groups of large instances with more than 1200 vertices, which are much more difficult than those smaller sized ones.

The results illustrate that TwMVC outperforms NuMVC for these hard BHOSLIB instances in terms of both success rate and run time. For instances solved by both algorithms with 100% success rate, TwMVC finds an optimal solution faster than NuMVC. For other instances, TwMVC has better success rate than NuMVC except for frb59-26-4. Indeed, as reported in (Cai et al. 2013), NuMVC significantly improves the success rate on hard BHOSLIB instances, and it is challenging to improve such good success rates. Nevertheless, TwMVC does push this further by showing better success rates consistently.

[7] ftp://dimacs.rutgers.edu/pub/dsj/clique/

Web Link Benchmark Results

The web link instances are much larger than standard benchmark instances, and their optimal solutions are still unknown. For these large instances, we first report the number of vertices ("|V|") and the number of edges ("|E|"). Then, we report the minimum size ("VC_{min}") and averaged size ("VC_{avg}") of vertex covers found by each algorithm, as the sizes of vertex covers found by the two algorithms vary considerably on these instances.

Table 3: Comparison of NuMVC and TwMVC on web link benchmark.

Instance	|V|	|E|	NuMVC		TwMVC	
			VC_{min}	VC_{avg}	VC_{min}	VC_{avg}
web-BerkStan	12.3K	19.5K	5384	5384	5384	5384
web-arabic05	163.5K	1.7M	114464	114475.36	**114435**	**114438.3**
web-edu	3K	6.4K	1451	1451	1451	1451
web-google	1.2K	2.7K	498	498	498	498
web-indochina04	11.3K	47.6K	7300	7300	7300	7300
web-it04	509K	7.1M	414700	414714.52	**414699**	414714.76
web-polblogs	643	2.2K	244	244	244	244
web-sk-2005	121.4K	334.4K	58198	58206	**58173**	**58173**
web-spam	4.7K	37.3K	2297	2297	2297	2297
web-uk-2005	129K	11.7M	127774	127774	127774	127774
web-webbase01	16K	25.5K	2651	2651.96	2651	**2651.86**

The results on the web link networks are summarized in Table 3. Overall, TwMVC finds better solutions than NuMVC on these web link networks. Specially, TwMVC finds smaller vertex covers that NuMVC cannot reach for 3 instances, and for `web-webbase01` where they both can find vertex covers of 2651 vertices, TwMVC does so with a better success rate.

For the remaining 7 instances, the two algorithms find solutions of the same quality consistently. Among these instances, 4 of them (`web-edu, web-google, web-polblogs and web-spam`) are relatively small and easy, and the run time to reach the solution is usually less than one second. For the other 3 instances, the averaged run time of TwMVC to reach such a solution in Table 3 is 62s, **1s** and **71s** for `Berkstan`, `indochina` and `web-uk-2005`, while that of NuMVC is **12s**, 16s and 82s.

Comparison with Exact MaxClq Algorithms

Recently, there has been great progress on MaxClq exact algorithms, mainly thanks to MaxSAT reasoning techniques (Li and Quan 2010b) and exploitation of graph structures (Li and Quan 2010a; Li, Fang, and Xu 2013).

For most DIMACS instances, TwMVC finds an optimal solution very quickly (usually within one second), much faster than exact algorithms. Also, these exact algorithms are not evaluated on some open DIMACS instances including `C2000.9`, `MANN_a81` and `keller6`, which are too difficult for exact algorithms (Li and Quan 2010a) and they cannot find a solution as good as TwMVC finds.

We compare TwMVC with two state-of-the-art exact solvers namely MaxCLQ (Li and Quan 2010b) and IncMC (Li, Fang, and Xu 2013) on DIMACS benchmark. The runtime of the exact solvers are taken from (Li, Fang, and Xu 2013) (the open instances and small instances are not reported there), where the time limit is 2000 seconds and the computing platform is an Intel Xeon CPU E7-8837@2.67GHz under Linux with 4GB RAM. The results (Table 4) show that TwMVC outperforms them on most instances. As for BHOSLIB instances, IncMC is currently the best exact solver on these instances (Li, Fang, and Xu 2013). Nevertheless, exact solvers still lag far behind local search solvers on BHOSLIB instances. For example, we are unaware of any exact MaxClq solver that can solve `frb50-23-3` or any `frb59` instance.

Table 4: Comparison of TwMVC and exact solvers on DIMACS instances

Graph Instance	TwMVC suc rate	TwMVC time	MaxCLQdyn time	IncMC time
`brock400_2`	100%	**18**	130	210
`brock400_4`	100%	**0.64**	154	259
`brock800_2`	6%	>2000	>2000	>2000
`brock800_4`	28%	>2000	>2000	>2000
`C250.9`	100%	<**0.1**	361	333
`C500.9`	100%	**0.1**	>2000	>2000
`C1000.9`	100%	**2**	>2000	>2000
`C2000.5`	100%	**2**	>2000	>2000
`C4000.5`	100%	**238**	>2000	>2000
`DSJC1000.5`	100%	**0.5**	382	261
`keller5`	100%	<**0.1**	>2000	177
`MANN_a27`	100%	<**0.1**	0.16	0.43
`MANN_a45`	100%	85	**25**	114
`p_hat700-1`	100%	<**0.1**	>2000	>2000
`p_hat700-2`	100%	<**0.1**	6.2	1.3
`p_hat700-3`	100%	<**0.1**	1383	357
`p_hat1500-1`	100%	6.28	12.3	**5.14**
`p_hat1500-2`	100%	<**0.1**	>2000	>2000
`p_hat1500-3`	100%	<**0.1**	>2000	>2000

Conclusions and Future Work

This paper proposed a vertex weighting scheme for MinVC and combined it within the current best MinVC local search algorithm NuMVC. While previous local search algorithms for MinVC prefer edge weighting techniques, this work suggests vertex weighting techniques can improve the robustness and efficiency of those algorithms. The resulting algorithm TwMVC outperforms NuMVC on standard benchmarks DIMACS and BHOSLIB, as well as a real-world benchmark from web link analysis.

TwMVC has a few instance-dependent parameters. A direction of future work is to eliminate the parameters in TwMVC. We would also like to develop more efficient MinVC and MaxClq algorithms for huge real-world networks.

Acknowledgement

This work is supported by China National 973 Program 2014CB340301, National Natural Science Foundation of China 61370072 and 61472369, and ARC Grant FT0991785. We would like to thank the anonymous referees for their helpful comments.

References

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

Battiti, R., and Protasi, M. 2001. Reactive local search for the maximum clique problem. *Algorithmica* 29(4):610–637.

Benlic, U., and Hao, J.-K. 2013. Breakout local search for maximum clique problems. *Computers & OR* 40(1):192–206.

Cai, S., and Su, K. 2013. Local search for Boolean Satisfiability with configuration checking and subscore. *Artif. Intell.* 204:75–98.

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

Cai, S.; Su, K.; and Chen, Q. 2010. EWLS: A new local search for minimum vertex cover. In *Proc. of AAAI-10*, 45–50.

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

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

Fang, Z.; Chu, Y.; Qiao, K.; Feng, X.; and Xu, K. 2014. Combining edge weight and vertex weight for minimum vertex cover problem. In *Proc. of FAW-14*, 71–81.

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

Grosso, A.; Locatelli, M.; and Pullan, W. J. 2008. Simple ingredients leading to very efficient heuristics for the maximum clique problem. *J. Heuristics* 14(6):587–612.

Hutter, F.; Tompkins, D. A. D.; and Hoos, H. H. 2002. Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In *Proc. of CP-02*, 233–248.

Karakostas, G. 2005. A better approximation ratio for the vertex cover problem. In *Proc. of ICALP-05*, 1043–1050.

Li, C. M., and Quan, Z. 2010a. Combining graph structure exploitation and propositional reasoning for the maximum clique problem. In *Proc. of ICTAI-10*, 344–351.

Li, C. M., and Quan, Z. 2010b. An efficient branch-and-bound algorithm based on MaxSAT for the maximum clique problem. In *Proc. of AAAI-10*, 128–133.

Li, C.-M.; Fang, Z.; and Xu, K. 2013. Combining maxsat reasoning and incremental upper bound for the maximum clique problem. In *Proc. of ICTAI-13*, 939–946.

Pullan, W., and Hoos, H. H. 2006. Dynamic local search for the maximum clique problem. *J. Artif. Intell. Res. (JAIR)* 25:159–185.

Pullan, W.; Mascia, F.; and Brunato, M. 2011. Cooperating local search for the maximum clique problem. *J. Heuristics* 17(2):181–199.

Pullan, W. 2006. Phased local search for the maximum clique problem. *J. Comb. Optim.* 12(3):303–323.

Richter, S.; Helmert, M.; and Gretton, C. 2007. A stochastic local search approach to vertex cover. In *Proc. of KI-07*, 412–426.

Rossi, R. A., and Ahmed, N. K. 2014. Coloring large complex networks. *Social Network Analysis and Mining (SNAM)* 1–52.

Rossi, R. A.; Gleich, D. F.; Gebremedhin, A. H.; and Patwary, M. A. 2013. A fast parallel maximum clique algorithm for large sparse graphs and temporal strong components. *arXiv preprint arXiv:1302.6256* 1–9.

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

Thornton, J.; Pham, D. N.; Bain, S.; and Jr., V. F. 2004. Additive versus multiplicative clause weighting for SAT. In *Proc. of AAAI-04*, 191–196.

Wu, Q.; Hao, J.-K.; and Glover, F. 2012. Multi-neighborhood tabu search for the maximum weight clique problem. *Annals of OR* 196(1):611–634.

Xu, K.; Boussemart, F.; Hemery, F.; and Lecoutre, C. 2007. Random constraint satisfaction: Easy generation of hard (satisfiable) instances. *Artif. Intell.* 171(8-9):514–534.

Zuckerman, D. 2006. Linear degree extractors and the inapproximability of max clique and chromatic number. In *Proc. of STOC-06*, 681–690.