Best-picking with Random Walk: Local Search for Minimum Vertex Cover in Massive Graphs

Optimizations, Constraint Optimizations, Heuristic Search

Abstract

The Minimum Vertex Cover (MinVC) problem is a well-known NP-hard problem. Although there have been many heuristics for solving this problem, most of them are focused on academic benchmarks of relatively small sizes. Considering the rapid growth of internet, wireless sensor networks, etc., there is great significance in solving this problem on real-world massive graphs. In this paper we propose a local search algorithm that exploits reduction rules, best-picking and random walk to solve the MinVC problem in such graphs. Experimental results show that our algorithm outperforms state-of-the-art local search algorithms on nearly all classes of these graphs. Moreover, our algorithm finds smaller covers on many graphs, and such a progress rarely happens in the MinVC research literature. This demonstrates the effectiveness of combining best-picking with random walk and the implementation of clever data structures.

1 Introduction

The Minimum Vertex Cover (MinVC) problem is a well-known NP-hard problem [Karp, 1972] with many real-world applications [Johnson and Trick, 1996]. Given a simple undirected graph $G = (V, E)$, an edge e is a set $\{u, v\}$ s.t. $u, v \in V$, and we say that u and v are endpoints of e. A *vertex cover* of a graph $G = (V, E)$ is a subset $V' \subseteq V$ s.t. for each $e \in E$, at least one of e's endpoints is in V'. The MinVC problem is to find a vertex cover of the minimum size.

With the development of social networks, wireless networks, etc., the MinVC problem on massive graphs has become more and more significant. So in this paper we are to find optimal or near-optimal covers in massive graphs.

It is hard to approximate MinVC within any factor smaller than 1.3606 [Dinur and Safra, 2004], so various heuristic methods have been proposed. They include: population based algorithms [Shyu *et al.*, 2004; Gilmour and Dras, 2006], stochastic local search [Richter *et al.*, 2007; Cai *et al.*, 2011; 2013; Fang *et al.*, 2014], and branch-and-bound algorithms [Pajarinen, 2007]. On the other hand, the MinVC problem is also closely related to both the Maximum Clique problem and the Maximum Independent Set problem, so algorithms for solving these latter problems are also able to solve the MinVC problem, *e.g.*, [Andrade *et al.*, 2008; Pullan, 2009; Li and Quan, 2010; Jin and Hao, 2015].

1.1 Recent Progress in Massive Graphs

Recently FastVC [Cai, 2015] makes a first step in massive graphs. For efficiency, it uses approximate heuristics instead of accurate ones. When removing a vertex, FastVC abandons the minimum loss removing heuristic in NuMVC [Cai *et al.*, 2013] which is a best-picking heuristic. Instead it adopts BMS (Best from Multiple Selections) to pick a relatively good vertex. Given a vertex set S, BMS works as follows: *randomly choose k vertices with replacement from the set S, and then return the one with the mimimum loss.*

In FastVC k is set to 50, hence, it is very probable to obtain a vertex whose loss value is among the best 10% [Cai, 2015]. However, in local search for massive graphs, S may contain millions of vertices. Therefore the best 10% could contain as many as 10^4 vertices, so we can miss some vertex with very small loss. Thus we will insist on the best-picking heuristic.

1.2 Our Motivations and Contributions

This work is to develop a local search MinVC solver called WalkCover, for massive graphs with strong structures.

Firstly our solver constructs starting vertex covers by combining reduction rules with the commonly used heuristic in [Richter *et al.*, 2007; Cai *et al.*, 2011; 2013].

Later in local search, we will utilize best-picking with random walk to remove a vertex. Given a vertex set S, our removing heuristic works like WalkSAT [Kautz *et al.*, 2009] as follows: *If there exist vertices in S whose loss is 0, randomly remove one of them. Otherwise, with probability p, remove a vertex in S with the minimum loss, breaking ties randomly; with probability $1 - p$, remove a vertex in S at random.*

We tested WalkCover and FastVC on the standard benchmark of massive graphs from the Network Data Repository[1] [Rossi and Ahmed, 2015]. Our experiments show that WalkCover outperforms FastVC in nearly all classes of instances.

Moreover WalkCover finds smaller covers for a considerable portion of the graphs. For all the 136 graphs we tested, our solver found covers containing *26* less vertices on average, and such a big improvement rarely happens in the litera-

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

ture. Previous MinVC algorithms often obtain the same quality solutions and they focus on comparing the success rate of finding a solution of such a quality [Cai, 2015].

Actually finding a k-sized cover is much more difficult than finding a $(k+1)$-sized cover [Cai *et al.*, 2013], so finding covers with *26* less vertices on average is *really difficult*. We will show the hardness by experimental data later in this paper.

2 Preliminaries

2.1 Basic Notations

If $e = \{u, v\}$ is an edge of G, we say that u and v are neighbors. We define $N(v)$ as $\{u \in V | \{u, v\} \in E\}$. The degree of a vertex v, denoted by $d(v)$, is defined as $|N(v)|$. We use $d_{avg}(G)$ to denote the average degree of graph G, suppressing G if understood from the context. An edge is covered by a vertex set S if at least one of its endpoints is in S. Otherwise it is uncovered by S. Given a graph $G = (V, E)$, the complement graph $\bar{G}$ of G is defined as $\bar{G} = (V, \bar{E})$ where $\bar{E} = \{\{u, v\} | \{u, v\} \notin E\}$.

A clique for G is a subset $Q \subseteq V$ s.t. every two vertices in Q are neighbors. The Maximum Clique (MaxClique) problem is to find a clique for G of the largest size. Then

Proposition 1 *C is a minimum cover for G iff $V \backslash C$ is a maximum clique for $\bar{G}$.*

To find a minimum cover for G, we can use a MaxClique solver, but we should first compute its complement graph $\bar{G}$.

2.2 Local Search for MinVC

Algorithm 1 shows the framework of a local search solver.

Algorithm 1: LocalSearchForMinVC(G, *cutoff*)

input : A graph $G = (V, E)$, the *cutoff* time
output: A vertex cover of G

```
1 C ← InitVC(G);
2 while elapsed time < cutoff do
3     if C covers all edges then
4         C* ← C;
5         remove a vertex from C with minimum loss;
6     ExchangeVertices(G, C);
7 return C*;
```

Algorithm 1 consists of the construction stage (Line 1) and the local search stage (Lines 2 to 6). In this paper, the vertex cover constructed in Line 1 is called the *starting vertex cover*.

In the local search stage, each time a k-sized cover is found (Line 3), the algorithm removes a vertex from C (Line 5) and continues to search for a $(k-1)$-sized cover, until some time limit is reached (Line 2). The move to a neighboring candidate solution consists of exchanging a pair of vertices (Line 6): a vertex $u \in C$ is removed from C, and a vertex $v \notin C$ is added into C. Such an exchanging procedure is also called a step by convention. When the algorithm terminates, it outputs the smallest cover that has been found.

For a vertex $v \in C$, the *loss* of v, denoted as $loss(v)$, is defined as the number of covered edges that will become uncovered by removing v from C. For a vertex $v \notin C$, the *gain* of v, denoted as $gain(v)$, is defined as the number of uncovered edges that will become covered by adding v into C.[2] Both loss and gain are *scoring properties* of vertices. In any step, a vertex v has two possible states: inside C and outside C. We use $age(v)$ to denote the number of steps that have been performed since last time v's state was changed.

2.3 Reduction Rules for MinVC

Our `InitVC(G)` will utilize the reduction rules below.

Degree-1 Rule: If $gain(v) = 1$ and u is a neighbor of v s.t. $u \notin C$, then put u into the C.

The two rules below are from [Chen *et al.*, 2001].

Degree-2 with Triangle Rule: If $gain(v) = 2$, and n_1, n_2 are both v's neighbors s.t. $n_1, n_2 \notin C$ and $\{n_1, n_2\} \in E$, then put both n_1 and n_2 into the C.

Degree-2 with Quadrilateral Rule: If $gain(u) = gain(v) = 2$, and both n_1, n_2 are neighbors shared by u, v s.t. $n_1, n_2 \notin C$ and $\{n_1, n_2\} \notin E$, then put both n_1 and n_2 into the C.

3 Constructing A Cover with Reductions

FastVC constructs starting vertex covers as follows:

Repeatedly select an uncovered edge e and add the endpoint with higher degree into C, until C becomes a cover. Redundant vertices are then removed.

However, this heuristic uses little structural information, and thus is ineffective for real-world graphs. So we apply reduction rules in the construction stage as Algorithm 2.

Algorithm 2: InitVC

input : A graph $G = (V, E)$
output: A starting vertex cover C

```
1  C ← ∅;
2  while there exist uncovered edges do
3      Repeatedly apply the Degree-1 Rule until it is not
       applicable;
4      foreach u ∈ V s.t. gain(u) = 2 do
5          apply Degree-2 with Triangle Rule if applicable;
6          otherwise apply Degree-2 with Quadrilateral
           Rule if applicable;
7      if any rule above is applicable then continue;
8      if all edges are covered then break;
9      pick a vertex v with the maximum gain (ties are
       broken randomly), then put it into C;
10 eliminate redundant vertices in C as FastVC;
11 return C;
```

In [Chen *et al.*, 2001], there are other reduction rules, but we think that they are less effective, so we do not use them.

Notice that our construction method is not preprocessing, as we never simplify the input graphs. Actually we tried preprocessing, but we did not find significant improvements.

[2] We remind readers that for a vertex $v \in C$, $gain(v)$ is undefined; for a vertex $v \notin C$, $loss(v)$ is undefined.

4 Local Search for k-sized Vertex Covers

Algorithm 3: ExchangeVertices(G, C)

input : A graph $G = (V, E)$, a candidate solution C
output: (C will be changed implicitly)

if *there exists a vertex v s.t. $loss(v) = 0$* **then**
  remove a vertex whose $loss$ is 0;
else
  With probability p: remove a vertex with min. $loss$;
  With probability $1 - p$: remove a random vertex;
$e \leftarrow$ a random uncovered edge;
add the endpoint of e with the greater $gain$ into C, breaking ties in favor of the greater age;

Our `ExchangeVertices(G,C)` is outlined in Algorithm 3 (all ties are broken randomly). Firstly it removes a vertex according to a WalkSAT-like heuristic. Then it adds a vertex into C just as what FastVC does.

5 Data Structures and Complexity Analysis

In order to lower the complexities, we exploited an efficient data structure named *score-based partition*. The main idea is to partition the vertices wrt. their scores, *i.e.*, two vertices are in the same partition if they have the same score, otherwise they are in different partitions (See Figures 1 to 3).

Given a graph $G = (V, E)$ and a candidate solution C, we implement the score-based partition on an array where each position holds a vertex. Besides, we maintain two types of pointers, each of which points to the beginning of a specific partition. Imagine the array as a book of vertices and the pointers as the indexes of this book.

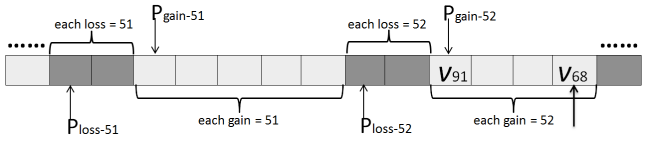


Figure 1: Adding v_{68} into C (a)

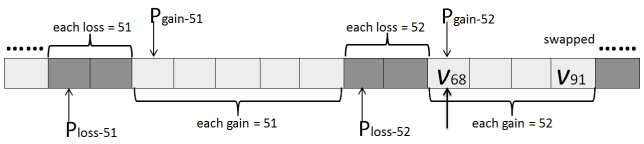


Figure 2: Adding v_{68} into C (b)

We use *loss-k* (resp. *gain-k*) partition to denote the partition that contains vertices whose loss (resp. gain) is k. Then we use Algorithm 4 to find those vertices with the minimum loss.

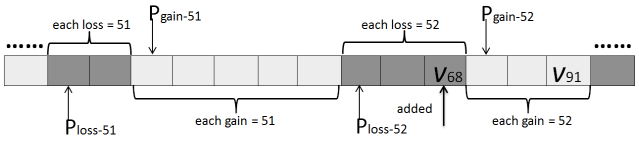


Figure 3: Adding v_{68} into C (c)

Algorithm 4: randomMinLossVertex

input : A sequence of score-based partitions
output: A random vertex $v \in C$ with minimum loss

$k \leftarrow 0$;
while *the loss-k partition is empty* **do** $k \leftarrow k + 1$;
return a random vertex in the *loss-k* partition;

5.1 Implementations

We use counting sort [Cormen *et al.*, 2009] to initialize the partitions, thus we have

Proposition 2 *Initializing the partitions is $O(|V|)$.*

After initializations, there are two cases in which a particular vertex, say v, has to be moved from one partition to another: (1) adding (resp. removing) v into (resp. from) C; (2) increasing/decreasing $gain(v)$/$loss(v)$ by 1. Thus the core operation is to move a vertex v to an adjacent partition.

Now we show how to do this with an example (See Figures 1 to 3). In this example, we are to add v_{68} into C. Initially v_{91} and v_{68} are in the *gain*-52 partition and thus their *gain* is 52 (Figure 1). Notice that after being added, v_{68}'s *loss* will become 52, *i.e.*, it should be in the *loss*-52 partition. Thus the operation is like this: (1) v_{68} is swapped with v_{91} (Figure 2); (2) $\mathsf{P}_{\text{gain-52}}$ is moved (Figure 3).

We define `placeVertexIntoC`(v) as the procedure that moves v from some *gain-k* partition to the respective *loss-k* partition. Also we define `dec_gain`(v) to be the procedure that moves v from some *gain-k* partition to the respective *gain-*(k-1) partition. Analogously we define `placeVertexOutfromC`(v), `dec_loss`(v), `inc_gain`(v), and `inc_loss`(v). Then we have

Proposition 3 *All these procedures are of $O(1)$ complexity.*

5.2 Complexity Analysis

Along with adding/removing a vertex v, we have to move v and all its neighbors to other partitions. Thus by Proposition 3, maintaining the partitions will take $O(1)$ time plus an amount of time proportional to $d(v)$. Thus we have

Theorem 4 *Suppose that each vertex has equal probability to be added or removed, then the average complexity of Algorithm 3 is $O(d_{\text{avg}})$.*

This is nice because [Cai, 2015] stated that the best-picking heuristic was of $O(|C|)$ complexity. Since most real-world graphs are sparse [Barabasi and Albert, 1999; Eubank *et al.*, 2004; Chung *et al.*, 2006], we have $d_{avg} << |C|$.

We use a hash table to store all the edges of a graph, so we determine whether two vertices are neighbors in $O(1)$ complexity *on average*. Then by Proposition 2 and 3, we have

Theorem 5 *The complexity of Algorithm 2 is* $O(|V| + |E|)$.

Notice that partitioning is a general method and can also be applied to solve huge instances for other problems.

6 Experimental Evaluation

We carry out extensive experiments to evaluate WalkCover, FastVC, and NuMVC on massive graphs. Since most real-world graphs are sparse, their complement graphs can become larger by orders of magnitude, *i.e.*, the complement graphs can be as large as thousands of gigabytes. Hence, it is not suitable to solve the MinVC problem on massive graphs using MaxClique solvers, *e.g.*, MaxCLQ [Li and Quan, 2010], so we did not include them in our experiments.

6.1 Benchmarks

We downloaded all 139 instances[3]. They were originally online[4]. But we excluded three extremely large ones, since they are out of memory for all the solvers here. Thus we only used the remaining 136 instances. Some of them were used in testing MaxClique and Coloring algorithms [Rossi and Ahmed, 2014; Rossi *et al.*, 2014]. In [Cai, 2015] there are 10 largest graphs on which NuMVC failed to return a starting vertex cover, and we collected them in Table 3.

6.2 Experiment Setup

All the solvers were compiled by g++ 4.6.3 with the '-O3' option. For FastVC[5] and NuMVC[6], we adopt the parameter settings reported in [Cai, 2015] and [Cai *et al.*, 2013] respectively. For WalkCover, we set the parameter p to 0.6. The experiments were conducted on a cluster equipped with a number of Intel(R) Xeon(R) CPUs X5650 @2.67GHz with 8GB RAM, running Red Hat Santiago OS.

Each solver is executed on each instance with a time limit of 1,000 seconds, with seeds from 1 to 100, if not mentioned explicitly. For each solver on each instance, we report the minimum size ("C_{min}") and averaged size ("C_{avg}") of the vertex covers found by the solver. To make the comparisons clearer, we also report the difference ("Δ") between the *minimum* size of the vertex covers found by FastVC and WalkCover respectively. A positive Δ means that WalkCover finds a smaller cover, while a negative Δ means that FastVC finds a smaller cover. Besides the best C_{min}'s are shown in **bold** font. The numbers of vertices of these graphs lie between 1×10^3 to 4×10^6, and readers may refer to [Cai, 2015].

6.3 Main Results

We mainly compared our solver to FastVC, since we found that FastVC performed better than other existing solvers on massive graphs [Cai, 2015][7]. Table 1 contains those instances where WalkCover and FastVC return different C_{min} or C_{avg} values. Table 2 contains the other instances. (We do not report results on graphs with less than 1,000 vertices or less than 1,000 edges where both solvers return the same C_{min} and C_{avg} values.)

In Table 1 we also show the results of `InitVC(`G`)` (named as InitVC). In this table, there are 14 instances where InitVC found *at least as good solutions as* FastVC, and we make $\star$ signs there.

Table 1: Results on graphs where WalkCover and FastVC return different C_{min} or C_{avg} values

Graph	FastVC $C_{min}(C_{avg})$	InitVC $C_{min}(C_{avg})$	WalkCover $C_{min}(C_{avg})$	Δ
ia-infect-dublin	293 (293.47)	293 (293.32)*	293 (293)	0
inf-roadNet-CA	1001254 (1001325.29)	1007205(1007374.2)	**1001104** (1001193.03)	150
inf-roadNet-PA	555203 (555248.74)	558192 (558338.4)	**555036** (555121.15)	167
rec-amazon	47606(47606.01)	47606(47610.25)*	**47605** (47605)	1
rt-retweet-crawl	81044 (81047.81)	81040(81040)*	**81040** (81040)	4
socfb-A-anon	375231 (375232.94)	375230(375230)*	**375230** (375230)	1
socfb-B-anon	303048 (303048.93)	303048(303048)*	303048 (303048)	0
socfb-Berkeley13	**17209** (17212.18)	17281(17290.92)	17210 (17213.82)	-1
socfb-CMU	4986 (4986.72)	5002(5007.22)	4986 (4986.94)	0
socfb-Duke14	7683 (7683.05)	7708(7713.21)	7683 (7683.81)	0
socfb-Indiana	**23313** (23317.19)	23428(23439.33)	23317 (23321.69)	-4
socfb-MIT	4657 (4657)	4664(4669)	4657 (4657.05)	0
socfb-OR	**36547** (36549.44)	36586(36594.77)	36548 (36550.23)	-1
socfb-Penn94	31161 (31164.95)	31302(31313.94)	**31160** (31168.80)	1
socfb-Stanford3	8517 (8517.89)	8534(8540.05)	8517 (8517.99)	0
socfb-Texas84	**28166** (28171.54)	28306(28317.33)	28168 (28177.08)	-2
socfb-UCLA	**15222** (15224.41)	15285(15295.25)	15223 (15226.54)	-1
socfb-UConn	13230 (13231.60)	13289(13299.89)	13230 (13233.81)	0
socfb-UCSB37	11261 (11262.88)	11307(11317.38)	11261 (11263.86)	0
socfb-UF	**27305** (27309.04)	27437(27454.31)	27308 (27314.44)	-3
socfb-UIllinois	**24090** (24093.97)	24210(24222.26)	24094 (24098.88)	-4
socfb-Wisconsin87	**18383** (18385.46)	18473(18483.48)	18384 (18387.73)	-1
tech-as-skitter	527161 (527204.59)	525150(525168.06)*	**525075** (525090.72)	2086
tech-RL-caida	74924 (74940.83)	74618(74625.9)*	**74609** (74615.32)	315
sc-ldoor	856754 (856757.36)	858059(858100.27)	856754 (856757.91)	0
sc-msdoor	**381558** (381559.23)	382090(382109.37)	381559 (381560.08)	-1
sc-nasasrb	51242 (51247.27)	51579(51606.6)	**51240** (51243.38)	2
sc-pkustk11	83911 (83912.97)	84124(84146.29)	83911 (83912.44)	0
sc-pkustk13	**89217** (89220.46)	89650(89673.85)	89222 (89225.92)	-5
sc-pwtk	207711 (207720.22)	208704(208760.63)	**207697** (207712.18)	14
sc-shipsec1	117305 (117338.65)	118721(118782.7)	**117251** (117304.94)	54
sc-shipsec5	147140 (147179.12)	147652(147710.09)	**146979** (147016.30)	161
scc_infect-dublin	9104 (9104)	9110(9112.83)	**9103** (9103)	1
soc-buzznet	30625 (30625)	30613(30613)*	**30613** (30613)	12
soc-delicious	85660 (85696.77)	85347(85365.14)*	**85321** (85333.84)	339
soc-digg	103243 (103244.72)	103234(103234)*	**103234** (103234)	9
soc-flickr	153272 (153272.03)	153272(153274.13)*	**153271** (153271.03)	1
soc-FourSquare	90108 (90109.09)	90108(90108)*	90108 (90108)	0
soc-gowalla	84222 (84222.36)	84223(84223.86)	84222 (84222)	0
soc-livejournal	1869044 (1869054.64)	1869191(1869203.79)	**1868924** (1868930.83)	120
soc-pokec	843419 (843432.58)	843876(843889.21)	**843347** (843352.07)	72
soc-youtube	146376 (146376.13)	146376(146376.21)*	146376 (146376)	0
web-arabic-2005	114425 (114427.28)	114429(114433.63)	**114420** (114421.26)	5
web-BerkStan	**5384** (5384)	5388(5388.09)	5385 (5385.31)	-1
web-it-2004	414671 (414675.12)	414900(414918.38)	**414644** (414648.57)	27
web-spam	2298 (2298.01)	2297(2298.08)*	**2297** (2297)	1
web-wikipedia2009	648315 (648321.83)	648396(648410.34)	**648300** (648311.45)	15

Quality Improvements

To demonstrate the quality improvements in Table 1, we present a bar graph below. We list all the graphs where $\Delta \neq 0$ on the x-axis, in the order of their appearance in Table 1.

So we find that

1. WalkCover often returns covers which are *much better*.
2. When it falls behind occasionally, it finds covers which are *only a little bit worse*.

Also, the Δ value is

1. greater than **10** for **13** graphs,
2. greater than **100** for **7** graphs,
3. and greater than **1,000** for **1** graph (`skitter`).

At last we remind readers that *it is rare to improve the quality in the MinVC research literature* [Cai *et al.*, 2013; Cai, 2015].

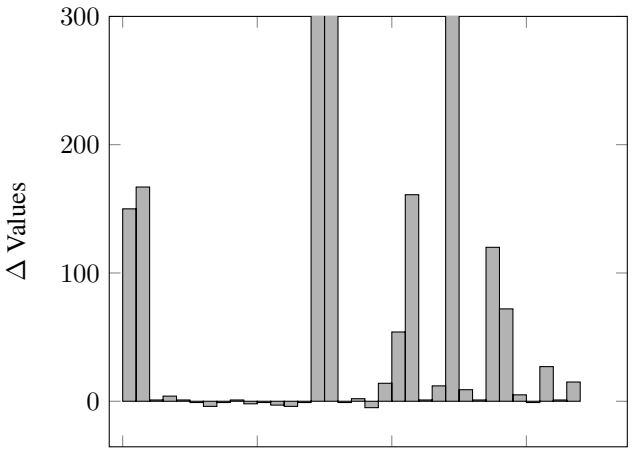
[3] http://lcs.ios.ac.cn/čaisw/Resource/realworld%20graphs.tar.gz

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

[5] http://lcs.ios.ac.cn/čaisw/Code/FastVC.zip

[6] http://lcs.ios.ac.cn/čaisw/Code/NuMVC-Code.zip, where we exploit a heap to implement the construction heuristic.

[7] This holds only when the cutoff does not exceed 1,000 seconds.



All Graph Instances on which $\Delta \neq 0$

Time Improvements and Individual Impacts

Table 2 compares the performances on those instances where two solvers return both the same C_{min} and C_{avg} values. We also present the averaged number of steps to locate a solution, and the number of steps executed in each millisecond.

1. The time columns show that WalkCover significantly outperforms FastVC.
2. The step columns illustrate that our *heuristic* is more clever than FastVC.
3. The last two columns[8] reveal that the complexity per step in WalkCover is roughly as low as that in FastVC, so our *data structure* is very efficient.
4. For most instances, WalkCover obtained its best solutions within **0** step, *i.e.*, `InitVC(`G`)` found the best solutions.

For 10 largest graphs, we show the averaged number of steps per millisecond in Table 3. So on huge instances,

1. WalkCover and FastVC had very close complexities.
2. WalkCover sometimes performed more steps per ms, because it had satisfactory starting vertex covers and thus spent much less time in updating the best solutions.

Robustness

From Tables 1 to 2, among all the 12 classes of graphs,

1. WalkCover is superior in 11 classes[9].
2. FastVC is better in only 1 class (Facebook Networks).

So WalkCover is much more robust.

The `InitVC(`G`)` Procedure

1. For about 1/3 of the instances in Table 1, InitVC found *at least as good solutions as* FastVC.
2. For nearly all instances in Table 2, InitVC found both the same C_{min} and C_{avg} values as FastVC.

[8]local search time (*not cutoff*) divided by the number of steps

[9]In Table 1 we find better solutions in `scc` networks.

So our `InitVC` procedure performed at least as well as FastVC on most instances. Besides

1. InitVC always outputs a cover within 10 seconds.
2. For most instances, it returns a cover within 1 second.

Table 2: Comparative performances on instances where both solvers find the same C_{min} and C_{avg} values

Graph	time		#step		#step/ms	
	FastVC	WalkC.	FastVC	WalkC.	FastVC	WalkC.
bio-dmela	0.010	<**0.001**	7740.1	**0**	888	**2054**
bio-yeast	<0.001	<0.001	193.3	**0**	900	**3762**
ca-AstroPh	0.029	**0.011**	18296.7	**917.0**	653	**987**
ca-citeseer	1.686	**0.128**	528968.1	**22825.5**	395	**904**
ca-coauthors-dblp	**19.581**	65.120	**4043060.9**	14413701.8	**247**	243
ca-CondMat	0.033	**0.023**	**19747.5**	22577.9	625	**1599**
ca-CSphd	<0.001	<0.001	320.3	**0**	829	**4393**
ca-dblp-2010	2.089	**0.238**	684860.3	**133199.4**	393	**914**
ca-dblp-2012	4.199	**0.280**	1077479.4	**34607.6**	338	**726**
ca-Erdos992	<0.001	<0.001	0	0	1099	**1988**
ca-GrQc	0.002	<**0.001**	2460.0	**0**	735	**2251**
ca-HepPh	0.013	**0.005**	8730.7	**0**	641	**935**
ca-hollywood-2009	26.678	**19.500**	4540583.6	**1462059.0**	**188**	154
ca-MathSciNet	5.525	**0.200**	1694815.4	**358.4**	360	**712**
ia-email-EU	**0.002**	0.003	119.3	**0**	**1052**	514
ia-email-univ	0.760	**0.000**	831881.0	**1273.9**	988	**2362**
ia-enron-large	0.065	**0.012**	38281.6	**0**	594	**1059**
ia-fb-messages	0.001	<**0.001**	2816.5	**0**	996	**2064**
ia-reality	<0.001	<0.001	0	0	**1033**	516
ia-wiki-Talk	0.121	**0.026**	57894.5	**0**	566	**695**
inf-power	0.011	<**0.001**	11332.0	**0**	875	**3360**
soc-BlogCatalog	0.309	**0.098**	152033.8	**0**	**554**	185
soc-brightkite	0.260	**0.016**	145548.5	**0**	572	**1013**
soc-douban	**0.013**	0.018	1245.3	**0**	**920**	497
soc-epinions	0.204	**0.006**	145479.2	**0**	699	**1455**
soc-flixter	2.336	**1.100**	737053.8	**0**	**526**	141
soc-lastfm	1.595	**0.615**	138125.7	**0**	**533**	183
soc-LiveMocha	29.644	**0.145**	14951362.4	**0**	**504**	259
soc-slashdot	0.268	**0.025**	132457.3	**0**	543	**784**
soc-twitter-follows	**0.036**	0.107	360.4	**0**	**929**	47
soc-youtube-snap	17.447	**0.570**	3352385.2	**0**	249	**502**
tech-as-caida2007	0.009	**0.003**	3562.7	**0**	696	**1334**
tech-internet-as	0.018	**0.014**	9474.0	**0**	650	**906**
tech-p2p-gnutella	0.009	**0.003**	7588.6	**0**	731	**1017**
tech-routers-rf	<0.001	<0.001	941.7	**0**	900	**2388**
tech-WHOIS	0.005	**0.002**	3153.3	**0**	838	**1082**
scc_fb-messages	**0.007**	0.010	0	0	**462**	66
scc_reality	**0.029**	0.079	0	0	**314**	19
scc_retweet	0.002	0.002	309.3	**0**	**883**	261
scc_retweet-crawl	0.062	**0.033**	18203.6	**0**	**479**	326
scc_rt_gmanews	<0.001	<0.001	0	0	**889**	584
scc_rt_lolgop	<0.001	<0.001	0	0	**1027**	596
scc_twitter-copen	**0.007**	0.012	486.1	**0**	**703**	111
web-edu	<0.001	<0.001	0	0	979	**2918**
web-google	0.001	<**0.001**	283.1	**0**	776	**2527**
web-indochina-2004	0.121	**0.034**	90864.1	**70410.2**	707	**1820**
web-sk-2005	**9.321**	22.386	**4357813.3**	19080249.5	454	**972**
web-uk-2005	**0.040**	0.512	0	0	**310**	128
web-webbase-2001	0.016	**0.001**	11315.9	**745.5**	**728**	524

Table 3: The number of steps per ms. on huge instances

Graph	FastVC	WalkCover	Graph	FastVC	WalkCover
ca-hollywood-2009	**641**	154	sc-ldoor	199	**265**
socfb-A-anon	**222**	129	soc-livejournal	156	**307**
socfb-B-anon	**244**	127	soc-pokec	196	**201**
inf-roadNet-CA	204	**842**	tech-as-skitter	222	**338**
inf-roadNet-PA	270	**876**	web-wikipedia2009	209	**524**

Table 4: The average Δ values wrt. different values of p

p	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
avg-Δ	25.3	27.3	26.7	26.3	26.0	26.2	26.0	25.9	25.8	25.7	25.6

6.4 Parameter Sensitivity

For each $0 \leq p \leq 1$, we did experiments and compared the results with FastVC as before. The average Δ values over 136 instances are shown in Table 4. So *for all parameter settings, we find significantly better quality covers on average*.

Besides the optimal parameter settings vary among instances.

1. For `sc-shipshec1`, `sc-shipshec5`, `web-it`, `inf-roadNet-PA`, `soc-delicious` and `tec-RL-caida`, the optimal setting is $p = 0$.
2. For `sc-pkustk13` and `soc-delicious`, the optimal setting is $p = 0.9$.
3. WalkCover with $p = 1.0$ can be viewed as an alternative which disables the function of random walk. So we can see that random walk has positive individual impacts.

Our experiments show that over the instances above, both WalkCover (with $p = 0.6$) and FastVC (with $k = 50$) are still far from the optimal covers. The cover sizes for any of them can further be decreased by at least 10.

6.5 Long-term Analysis

Since some tested graphs here are larger than the traditional benchmark graphs by orders of magnitude, it is interesting to evaluate the MinVC solvers on them using a larger cutoff.

Hardness of Finding Smaller Covers

To show the hardness, we enlarged the time limit to be 100 times as large as before, *i.e.*, **100,000s** (27.8 hours). Then among the 10 largest graphs (See Table 3), we tested NuMVC and FastVC on those 7 graphs where $\Delta \neq 0$ (See Table 1), using seeds from 1 to 10. The experimental results are shown in Table 5. Also we show the respective results of WalkCover with $p = 0.6$ in **1,000 seconds**, with seeds from 1 to 100.[10]

Comparing Tables 1 and 5, we found that the solution quality of FastVC did not improve much. Moreover, even when NuMVC and FastVC becomes 100 times as fast as before, WalkCover still takes a lead over these instances.

Table 5: Comparative results when NuMVC and FastVC use 100 times of the cutoff (WalkCover uses the original cutoff)

Graph	NuMVC×100 $C_{min}(C_{avg})$	FastVC×100 $C_{min}(C_{avg})$	WalkCover $C_{min}(C_{avg})$
socfb-A-anon	375232 (375233.4)	375230 (375230.3)	375230 (**375230**)
inf-roadNet-CA	1010097 (1010203.50)	1001272 (1001306.30)	1001104 (**1001193.03**)
inf-roadNet-PA	559963 (560015.90)	555220 (555242.00)	555036 (**555121.15**)
soc-livejournal	1871898 (1871970.90)	1869030 (1869034.70)	1868924 (**1868930.83**)
soc-pokec	844181 (844246.50)	843417 (843428.80)	843347 (**843352.07**)
tech-as-skitter	525853 (525900.90)	527185 (527195.30)	525075 (**525090.72**)
web-wikipedia2009	649165 (649234.00)	648314 (648320.40)	648300 (**648311.45**)

[10]Since the number of runs are not the same for all solvers, we **bold** the best C_{avg}.

Long-term Behaviors

For each graph in Table 5, we observed the long-term behaviors of MinVC solvers as follows. Each time the best solution was updated, we marked down the cover-size and the time stamp. Then for each cover-size we averaged the respective time stamps in different runs[11] of a solver. So we could show how the cover-sizes ($|C^*|$) decreased when time increased (t), *e.g.*, Figure 4. Our experiments show that all these graphs had similar curves, so we only present the curves for `soc-livejournal` which has the *largest* size.

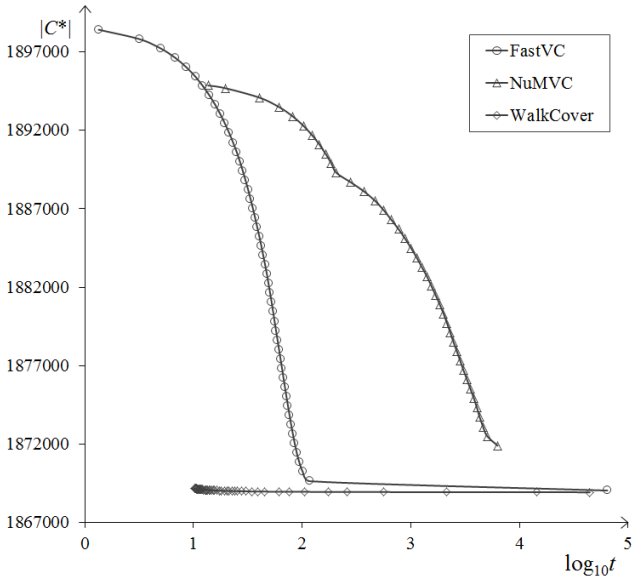


Figure 4: Long-term Behaviors on `soc-livejournal`

Observing the curves for all these graphs, we find that

1. FastVC often output starting vertex covers quickly while WalkCover often had good starting covers.
2. For any solver, finding a k-cover cost much more time than finding a $(k+1)$-cover.

7 Conclusions and Future Work

In this paper, we improve FastVC in two ways: (1) in the construction stage, we replace the edge-greedy heuristic with a max-gain with reduction heuristic; (2) in the local search stage, we change the BMS heuristic to be a WalkSAT-like removing heuristic.

Experimental results show that our solver WalkCover outperforms FastVC on nearly all classes of the instances. Moreover, WalkCover finds covers containing *26* less vertices on average, which is really a great progress in the literature of MinVC solving.

As for future works we will utilize more diversification strategies in our solver. Also, we will apply reduction rules to exchange vertices in local search.

[11]If a cover-size cannot be found in some certain run, we simply ignore that run.

References

[Andrade *et al.*, 2008] Diogo Vieira Andrade, Mauricio G. C. Resende, and Renato Fonseca F. Werneck. Fast local search for the maximum independent set problem. In *Experimental Algorithms, 7th International Workshop, WEA 2008, Provincetown, MA, USA, May 30-June 1, 2008, Proceedings*, pages 220–234, 2008.

[Barabasi and Albert, 1999] Albert-Laszlo Barabasi and Reka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[Cai *et al.*, 2011] Shaowei Cai, Kaile Su, and Abdul Sattar. Local search with edge weighting and configuration checking heuristics for minimum vertex cover. *Artif. Intell.*, 175(9-10):1672–1696, 2011.

[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. (JAIR)*, 46:687–716, 2013.

[Cai, 2015] Shaowei Cai. 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*, pages 747–753, 2015.

[Chen *et al.*, 2001] Jianer Chen, Iyad A. Kanj, and Weijia Jia. Vertex cover: Further observations and further improvements. *J. Algorithms*, 41(2):280–301, 2001.

[Chung *et al.*, 2006] F.R.K. Chung, L. Lu, Conference Board of the Mathematical Sciences, and National Science Foundation (U.S.). *Complex Graphs and Networks*. Number no. 107 in Complex graphs and networks. American Mathematical Society, 2006.

[Cormen *et al.*, 2009] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[Dinur and Safra, 2004] Irit Dinur and Shmuel Safra. On the hardness of approximating label-cover. *Inf. Process. Lett.*, 89(5):247–254, 2004.

[Eubank *et al.*, 2004] Stephen Eubank, V. S. Anil Kumar, Madhav V. Marathe, Aravind Srinivasan, and Nan Wang. Structural and algorithmic aspects of massive social networks. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004, New Orleans, Louisiana, USA, January 11-14, 2004*, pages 718–727, 2004.

[Fang *et al.*, 2014] Zhiwen Fang, Yang Chu, Kan Qiao, Xu Feng, and Ke Xu. Combining edge weight and vertex weight for minimum vertex cover problem. In *Frontiers in Algorithmics - 8th International Workshop, FAW 2014, Zhangjiajie, China, June 28-30, 2014. Proceedings*, pages 71–81, 2014.

[Gilmour and Dras, 2006] Stephen Gilmour and Mark Dras. Kernelization as heuristic structure for the vertex cover problem. In *Ant Colony Optimization and Swarm Intelligence, 5th International Workshop, ANTS 2006, Brussels, Belgium, September 4-7, 2006, Proceedings*, pages 452–459, 2006.

[Jin and Hao, 2015] Yan Jin and Jin-Kao Hao. General swap-based multiple neighborhood tabu search for the maximum independent set problem. *Eng. Appl. of AI*, 37:20–33, 2015.

[Johnson and Trick, 1996] David J. Johnson and Michael A. Trick, editors. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, Workshop, October 11-13, 1993*. American Mathematical Society, Boston, MA, USA, 1996.

[Karp, 1972] Richard M. Karp. Reducibility among combinatorial problems. In *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York.*, pages 85–103, 1972.

[Kautz *et al.*, 2009] Henry A. Kautz, Ashish Sabharwal, and Bart Selman. *Incomplete Algorithms, Handbook of Satisfiability*. IOS Press, 2009.

[Li and Quan, 2010] Chu Min Li and Zhe Quan. An efficient branch-and-bound algorithm based on maxsat for the maximum clique problem. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*, 2010.

[Pajarinen, 2007] Joni Pajarinen. Calculation of typical running time of a branch-and-bound algorithm for the vertex-cover problem. 2007.

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

[Richter *et al.*, 2007] Silvia Richter, Malte Helmert, and Charles Gretton. A stochastic local search approach to vertex cover. In *KI 2007: Advances in Artificial Intelligence, 30th Annual German Conference on AI, KI 2007, Osnabrück, Germany, September 10-13, 2007, Proceedings*, pages 412–426, 2007.

[Rossi and Ahmed, 2014] Ryan A. Rossi and Nesreen K. Ahmed. Coloring large complex networks. *Social Netw. Analys. Mining*, 4(1):228, 2014.

[Rossi and Ahmed, 2015] Ryan A Rossi and Nesreen K Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[Rossi *et al.*, 2014] Ryan A. Rossi, David F. Gleich, Assefaw Hadish Gebremedhin, and Md. Mostofa Ali Patwary. Fast maximum clique algorithms for large graphs. In *23rd International World Wide Web Conference, WWW '14, Seoul, Republic of Korea, April 7-11, 2014, Companion Volume*, pages 365–366, 2014.

[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.