Local Search for Maximum Vertex Weight Clique on Large Sparse Graphs with Efficient Data Structures

Yi Fan[1(✉)], Chengqian Li[2], Zongjie Ma[1], Lian Wen[1], Abdul Sattar[1], and Kaile Su[1]

[1] Institute for Integrated and Intelligent Systems, Griffith University, Brisbane, Australia
{yi.fan4,zongjie.ma}@griffithuni.edu.au,
{l.wen,a.sattar,k.su}@griffith.edu.au
[2] Department of Computer Science, Sun Yat-sen University, Guangzhou, China

Abstract. The Maximum Vertex Weight Clique (MVWC) problem is a generalization of the Maximum Clique problem, which exists in many real-world applications. However, it is NP-hard and also very difficult to approximate. In this paper we developed a local search MVWC solver to deal with large sparse instances. We first introduce random walk into the multi-neighborhood greedy search, and then implement the algorithm with efficient data structures. Experimental results showed that our solver significantly outperformed state-of-the-art local search MVWC solvers. It attained all the best-known solutions, and found new best-known solutions on some instances.

Keywords: Local search · Maximum vertex weight clique · Large sparse graphs · Data structures

1 Introduction

The Maximum Clique (MC) problem is a well-known NP-hard problem [17]. Given a simple undirected graph $G = (V, E)$, a *clique* C is a subset of V s.t. each vertex pair in C is mutually adjacent. The Maximum Clique problem is to find a clique of the maximum size. An important generalization of the MC problem is the Maximum Vertex Weight Clique (MVWC) problem in which each vertex is associated with a positive integer weight, and the goal is to find a clique with the greatest total vertex weight. This generalization is important in many real-world applications like [1,3,4,8,9,18,19,23]. In this paper we are concerned in finding a clique whose total vertex weight is as great as possible.

Both MC and MVWC are NP-hard, and the state-of-the-art approximation algorithm can only achieve an approximation ratio of $O(n(\log\log n)^2/(\log n)^3)$ [15]. Thus various heuristic methods have been developed to find a "good" clique within reasonable time. Up to now, there are two types of algorithms for the

B.H. Kang and Q. Bai (Eds.): AI 2016, LNAI 9992, pp. 255–267, 2016.
DOI: 10.1007/978-3-319-50127-7_21

MVWC problems: complete algorithms and incomplete ones. Complete algorithms for MVWC include [2,14,21,30]. On the other hand, incomplete algorithms for MVWC include [7,10,22,27,28].

The aim of this work is to develop a local search MVWC solver named LMY-GRS[1] to deal with large crafted graphs. Firstly we incorporate random walk into the multi-neighborhood greedy search in [28]. Then we propose novel data structures to achieve greater efficiency.

We used the large crafted benchmark in [27][2] to test our solver. And we make two state-of-the-art solvers, MN/TS [28] and LSCC [27], as the competitors[3]. Experimental results show that LMY-GRS *attained all the best-known solutions* in this large crafted benchmark. Moreover, for a large proportion of the graphs LMY-GRS achieves better average quality. Among them there are four graphs where LMY-GRS reports new best-known solutions.

2 Preliminaries

2.1 Basic Notations

Given a graph $G = (V, E)$ where $V = \{v_1, \ldots, v_n\}$, an edge is a 2-element subset of V. Given an edge $e = \{u, v\}$, we say that u and v are adjacent to each other. Also we say that u and v are neighbors, and we use $N(v) = \{u | \{u, v\} \in E\}$ to denote the set of v's neighbors. The degree of a vertex v, denoted by $d(v)$, is defined as $|N(v)|$. We use $d_{max}(G)$ to denote the maximum degree of graph G, suppressing G if understood from the context. A clique C of G is a subset of V where vertices are pair-wise adjacent. An empty set is said to be an empty clique. A set that contains a single vertex is called a single-vertex clique.

Given a weighting function $w : V \to Z^+$, the weight of a clique C, denoted by $w(C)$, is defined to be $\sum_{v \in C} w(v)$. We use $age(v)$ to denote the number of steps since last time v changed its state (inside or outside the candidate clique).

2.2 The Large Crafted Benchmark

MC and MVWC solvers are often tested on the DIMACS [16] and BOSHLIB [29] benchmarks. Recently large real-world benchmarks have become very popular. Many of these graphs have millions of vertices and dozens of millions of edges. They were used in testing Graph Coloring and Minimum Vertex Cover algorithms [11,24,26], as well as the MC [25] and MVWC [27] algorithms.

In this paper we focus on the large benchmark. They were originally unweighted, and to obtain the corresponding MVWC instances, we use the same method as in [22,27,28]. For the i-th vertex v_i, $w(v_i) = (i \mod 200) + 1$.

[1] https://github.com/Fan-Yi/Local-Search-for-Maximum-Vertex-Weight-Clique-on-Large-Sparse-Graphs-with-Efficient-Data-Structures.

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

[3] In [27], both solvers are incorporated with a heuristic named BMS to solve large instances. For simplicity, we write them as MN/TS and LSCC for short.

2.3 Multi-neighborhood Greedy Search

Usually the local search moves from one clique to another until the cutoff arrives, then it returns the best clique that has been found. There are three operators: `add`, `swap` and `drop`, which guide the local search. To ensure that these operators preserve the clique property, two sets are defined as below.

1. $AddS = \{v | v \notin C, v \in N(u) \text{ for all } u \in C\}$.
2. $SwapS = \{(u, v) | u \in C, v \notin C, \{u, v\} \notin E, v \in N(w) \text{ for all } w \in C \backslash \{u\}\}$.

So the `add` operator can only take an element from $AddS$ as its operand. Similarly the `swap` (resp. `drop`) operator can only take an element from $SwapS$ (resp. C).

Proposition 1. *If* $(u_1, v) \in SwapS$ *and* $(u_2, v) \in SwapS$, *then* $u_1 = u_2$[4].

That is, if a vertex v can go into C through a `swap` operation, then the vertex to be removed is unique. Then we have

Lemma 1. *For any* $P \subseteq SwapS$, $|P| = |\{v | (u, v) \in P\}|$.

So P can be projected to V by considering the second element in the swap-pairs.

We use Δ_{add}, Δ_{swap} and Δ_{drop} to denote the increase of $w(C)$ for the operations `add`, `swap` and `drop` respectively. Obviously, we have (1) for a vertex $v \in AddS$, $\Delta_{add}(v) = w(v)$; (2) for a vertex $u \in C$, $\Delta_{drop}(u) = -w(u)$; (3) for a vertex pair $(u, v) \in SwapS$, $\Delta_{swap}(u, v) = w(v) - w(u)$. Basically both MN/TS and LSCC obtain the best local move like Algorithm 1.

Algorithm 1. bestLocalMove

```
1 v ← a vertex in AddS with the biggest Δ_add;
2 (u, u') ← a pair in the SwapS with the biggest Δ_swap;
3 x ← a vertex in C with the biggest Δ_drop;
4 if AddS ≠ ∅ then
5     C ← (Δ_add > Δ_swap)?(C ∪ {v}) : (C\{u}) ∪ {u'};
6 else
7     C ← (Δ_drop > Δ_swap)?(C\{x}) : (C\{u}) ∪ {u'};
```

[27] stated that *SwapS is usually very large when we solve large sparse graphs.* Yet we will show that this statement seems not to be the case.

2.4 The Strong Configuration Checking Strategy

[27] proposed a strategy named strong configuration checking (SCC): (1) in the beginning of the search, $confChange(v)$ is set to 1 for each vertex v; (2) when v is added, $confChange(n)$ is set to 1 for all $n \in N(v)$; (3) when v is dropped, $confChange(v)$ is set to 0; (4) When $u \in C$ and $v \notin C$ are swapped, $confChange(u)$ is set to 0.

[4] For any vertices u and v, we use $u = v$ to denote that u and v are the same vertex.

2.5 Best from Multiple Selections (BMS)

BMS is equivalent to *deterministic tournament selection* in genetic algorithms [20]. Given a set S and a positive integer k, it works as follows: *randomly select k elements from S with replacements and then return the best one.*

3 Local Move Yielded by Greedy and Random Selections

Based on Lemma 1, we have a theorem below which shows the bound of $|SwapS|$.

Theorem 1

1. *If $C = \{w\}$ for some w, then $|SwapS| = |V| - |N(w)| - 1$;*
2. *If $|C| > 1$, then $|SwapS| \leq 2d_{\max}$.*

Proof. The first item is trivial, so we only prove the second one. Since $|C| > 1$, there must exist two different vertices u and w in C. Now we partition $SwapS$ to be: $P_1 = \{(x, y) \in SwapS | x = u\}$ and $P_2 = \{(x, y) \in SwapS | x \neq u\}$.

For any pair $(x, y) \in P_1$, since $x = u$, y must be a neighbor of w. Therefore, $\{y|(x, y) \in P_1\} \subseteq N(w)$. So $|\{y|(x, y) \in P_1\}| \leq |N(w)|$. By the Lemma 1, we have $|P_1| = |\{y|(x, y) \in P_1\}| \leq d(w)$, and thus $|P_1| \leq d_{max}$.

For any pair $(x, y) \in P_2$, since $x \neq u$, y must be a neighbor of u. Therefore, $\{y|(x, y) \in P_2\} \subseteq N(u)$. So $|\{y|(x, y) \in P_2\}| \leq |N(u)|$. By the Lemma 1, we have $|P_2| = |\{y|(x, y) \in P_2\}| \leq d(u)$, and thus $|P_2| \leq d_{max}$.

Therefore, $|SwapS| = |P_1| + |P_2| \leq 2d_{max}$.

Since most large real-world graphs are sparse [5,12,13], we have $d_{max} \ll |V|$. Moreover, observing all the graph instances in the experiments, we find that $d_{max} < 3,000$. So for most of the time, $SwapS$ is a small set.

$SwapS$ becomes huge only when C contains a *single* vertex, namely u. In this situation, any vertex outside C but not adjacent to u, namely v, can go into C through a `swap` operation, and $\Delta_{swap}(u, v) = w(v) - w(u)$. Thus picking the best pair is somewhat equivalent to picking a vertex outside with the greatest weight. If we do so, we will obtain a single-vertex clique that contains a vertex with the greatest or near-greatest weight.

Usually there is only a tiny proportion of vertices whose weight is the greatest or close to the greatest. So when $|C| = 1$, if we pick the best swap-pair, we will obtain a clique from a tiny proportion of the single-vertex ones. Therefore, when $|C| = 1$ happens many times, the local search may revisit some areas.

We realize that when $|C| = 1$, [27] uses BMS, SCC and age to help diversify the local search. Yet it is unclear whether greedy search here is necessary, and we believe that random walk is feasible. In our solver, we will abandon BMS and use random walk instead, and the experimental performances are satisfactory.

4 The LMY-GRS Algorithm

LMY-GRS works with three operators `add`, `swap` and `drop`. With the current clique denoted by C, two sets S_{add} and S_{swap} are defined as below which are slightly different from $AddS$ and $SwapS$.

$$S_{add} = \begin{cases} \{v | v \notin C, v \in N(u) \text{ for all } u \in C\} & \text{if } |C| > 0; \\ \emptyset & \text{if } |C| = 0. \end{cases}$$

$$S_{swap} = \begin{cases} \{(u,v) | u \in C, v \notin C, \{u,v\} \notin E, v \in N(w) \text{ for all } w \in C \backslash \{u\}\} & \text{if } |C| > 1; \\ \emptyset & \text{if } |C| \leq 1. \end{cases}$$

Obviously we have

Proposition 2. *$|S_{add}| \leq d_{\max}$, and $|S_{swap}| \leq 2d_{\max}$.*

In our algorithm, the vertices of the operations are explicit from the context and thus omitted.

Basically LMY-GRS firstly initializes a clique via `RandInitClique(`G`)`, and then uses local search to find better cliques. The initialization procedure is just the same as that in MN/TS and LSCC, which is shown in Algorithm 3.

Algorithm 2. LMY-GRS

input : A graph $G = (V, E, w)$ and the *cutoff*
output: A clique that was found with the greatest weight

```
1  step ← 0; initialize the confChange array; C ← RandInitClique(G);
2  while elapsed time < cutoff do
3      if C = ∅ then add a random vertex into C ;
4      v ← a vertex in S_add s.t. confChange(v) = 1 with the biggest Δ_add,
         breaking ties in favor of the oldest one; otherwise v ← NULL;
5      (u, u') ← a pair in S_swap s.t. confChange(u') = 1 with the biggest Δ_swap,
         breaking ties in favor of the oldest u'; otherwise (u, u') ← (NULL, NULL);
6      if v ≠ NULL then
7          if (u, u') = (NULL, NULL) or Δ_add > Δ_swap then C ← C ∪ {v} ;
8          else C ← C\{u} ∪ {u'} ;
9      else
10         x ← a vertex in C with the biggest Δ_drop, breaking ties in favor of the
             oldest one;
11         if (u, u') = (NULL, NULL) or Δ_drop > Δ_swap then C ← C\{x} ;
12         else C ← C\{u} ∪ {u'} ;
13     step ← step + 1; if w(C) > w(C*) then C* ← C;
14     if step mod L = 0 then
15         drop all vertices in C; C ← RandInitClique(G);
16     update confChange array according to SCC rules;
17 return C*;
```

Algorithm 3. RandInitClique(G)

```
C ← ∅;
add a random vertex in C;
while S_add ≠ ∅ do add a random vertex from S_add to C; step ← step + 1 ;
if w(C) > w(C*) then C* ← C;
return C;
```

Like MN/TS and LSCC, we also exploit the multi-restart strategy. More specifically, every L steps we restart the local search. The difference from previous restarting methods is that we simply drop all vertices from C and regenerate a random maximal clique.

In LMY-GRS, C may sometimes become empty. In this situation, we add a random vertex in C and proceed to search for a better clique.

5 Data Structures

5.1 Connect Clique Degrees and Clique Neighbors

Definition 1. *Given a clique C and a vertex $v \notin C$, we define the connect clique degree of v to be $\kappa(v, C) = |\{u \in C | u \text{ and } v \text{ are neighbors.}\}|$.*

So the connect clique degree of v is the number of vertices in C that are adjacent to v. Then we can maintain $\kappa(v, C)$ when a vertex is added into or dropped from C, based on the following proposition.

Proposition 3

1. $\kappa(v, C \backslash \{v\}) = |C| - 1$ *for any* $v \in C$;
2. $\kappa(v, C \backslash \{w\}) = \kappa(v, C) - 1$ *for all* $v \in (N(w) \backslash C)$;
3. $\kappa(v, C \cup \{w\}) = \kappa(v, C) + 1$ *for all* $v \in (N(w) \backslash C)$.

In [6], there is a notion named *the number of missing connections*. In fact the number of v's missing connections is $|C| - \kappa(v, C)$.

Definition 2. *Given a clique C, we define the clique neighbor set to be*

$$\mathcal{N}(C) = \begin{cases} \{v \notin C | v \in N(u) \text{ for some } u \in C\} & \text{if } |C| > 0; \\ \emptyset & \text{if } |C| = 0. \end{cases}$$

So clique neighbors are those vertices outside C which are adjacent to at least one vertex inside C.

When a vertex namely u is added into or dropped from C, $\mathcal{N}(C)$ is updated based on the proposition below.

Proposition 4

1. *For any vertex* u, $u \notin \mathcal{N}(C \cup \{u\})$;
2. *for any* $u \in C$, $|C| > 1$ *iff* $u \in \mathcal{N}(C \backslash \{u\})$;
3. *for any* $v \notin C$, $\kappa(v, C) > 0$ *iff* $v \in \mathcal{N}(C)$.

Lastly, we use the following proposition to maintain S_{add} and S_{swap}.

Proposition 5

1. *For any* v, $v \in S_{add}$ *iff* $v \in \{w \in \mathcal{N}(C) | \kappa(w, C) = |C|\}$;
2. *there exists* $u \in C$ *s.t.* $(u, v) \in S_{swap}$, *iff* $v \in \{w \in \mathcal{N}(C) | \kappa(w, C) = |C| - 1\}$.

This tells us that we can maintain S_{add} and S_{swap} simply by traversing $\mathcal{N}(C)$, so we do not have to traverse V. Previous local search solvers for MC or MVWC exclusively traverse all the vertices in V to maintain S_{add} and S_{swap}, so our implementation can sometimes be much more efficient. Notice that $|\mathcal{N}(C)| \ll |V|$ usually holds in huge sparse graphs.

5.2 A Hash Table for Determining Neighbor Relations

In graph algorithms there is a common procedure: *Given a graph* G *and two vertices* u *and* v, *determining whether* u *and* v *are neighbors.* In MN/TS, LSCC and LMY-GRS, this procedure is called *very frequently*, so we have to implement it efficiently. However, it is unsuitable to store the large sparse graphs by adjacency matrices. Therefore, we propose the following data structure which is both memory and time efficient.

We employ a one-to-one function $f : (Z^+, Z^+) \to Z^+$ and use a hash table to implement the procedure above. Given a graph $G = (V, E)$, for any $\{v_i, v_j\} \in E$ where $i < j$, we store $f(i, j)$ in a hash table $\mathcal{T}_h$. Then each time when we need to determine whether v_k and v_l $(k < l)$ are neighbors, we simply check whether $f(k, l)$ is in $\mathcal{T}_h$. If so they are neighbors; otherwise, they are not.

In LMY-GRS, we adopt Cantor's pairing function as the function f above, *i.e.*, $f(x, y) = (x + y)(x + y + 1) \div 2 + y$. Then we can determine whether two vertices are neighbors in $O(1)$ complexity on average.

With the data structures above, our solver is able to perform steps faster than LSCC by orders of magnitude on huge sparse graphs.

6 Experimental Evaluation

In this section, we carry out extensive experiments to evaluate LMY-GRS on large crafted graphs, compared against the state-of-the-art local search MVWC algorithms MN/TS and LSCC.

6.1 Experiment Setup

All the solvers were compiled by g++ 4.6.3 with the '-O3' option. For the search depth L, MN/TS, LSCC and LMY-GRS set $L = 4,000$. Both MN/TS and LSCC employ the BMS heuristic, and the parameter k was set to 100, as in [27]. MN/TS employs a tabu heuristic and the tabu tenure TL was set to 7 as in [28]. The experiments were conducted on a cluster equipped with Intel(R) Xeon(R) CPUs X5650 @2.67 GHz with 8 GB RAM, running Red Hat Santiago OS.

Each solver was executed on each instance with a time limit of 1,000 s, with seeds from 1 to 100. For each algorithm on each instance, we report the maximum weight ("w_{max}") and averaged weight ("w_{avg}") of the cliques found by the algorithm. To make the comparisons clearer, we also report the difference ("δ_{max}") between the maximum weight of the cliques found by LMY-GRS and that found by LSCC. Similarly δ_{avg} represents the difference between the averaged weights. A positive δ_{avg} (resp. δ_{max}) indicates that LMY-GRS performed better, while a negative value indicates that LMY-GRS performed worse.

6.2 Main Results

We show the main experimental results in Tables 1 and 2. For the sake of space, we do not report the results on graphs with less than 1,000 vertices.

Quality Improvements. Table 1 shows the performances on the instances where LSCC and LMY-GRS returned different w_{max} or w_{avg} values.
From the results in Table 1, we observe that:

1. LMY-GRS attained best-known solutions on all the graphs;
2. In a large proportion of the graphs, LMY-GRS returned solutions which had better average quality;
3. LMY-GRS found new best-known solutions in 4 graphs.

In fact these 4 graphs are the largest ones in the benchmark, and each of them has at least 10^6 vertices. Since LMY-GRS and LSCC present different solution qualities over these instances, it is inconvenient to evaluate the individual impacts of the heuristics and the data structures over these instances.

Time Improvements and Individual Impacts. Table 2 compares the performances on those instances where LSCC and LMY-GRS returned both the same w_{max} and w_{avg} values. We also present the averaged number of steps to locate a solution, and the number of steps executed in each millisecond.

Over these instances, we find that

1. The time columns show that LMY-GRS and LSCC performed closely well;
2. The step columns show that the heuristic in LMY-GRS is as good as the one in LSCC, since they needed roughly the same number of steps;
3. The last two columns show that LMY-GRS sometimes performed steps faster than LSCC, but sometimes slower.

To show the power of our data structures, we present the averaged number of steps per millisecond in some largest instances in Table 3. In this table we found that LMY-GRS is able to perform steps faster than LSCC by orders of magnitudes in some instances.

Based on the discussions above, we conclude that using random walk is roughly as good as using BMS, and our data structures are very powerful on huge sparse graphs.

Table 1. Results where LSCC and LMY-GRS returned different w_{max} or w_{avg} values

Graph	$\|V\|$	$\|E\|$	MN/TS	LSCC	LMY-GRS	$\delta_{max}(\delta_{avg})$
			w_{max} (w_{avg})	w_{max} (w_{avg})	w_{max} (w_{avg})	
ca-coauthors-dblp	540486	15245729	37884 (27411.17)	37884 (34211.68)	37884 (**37884.00**)	0 (3672.32)
ca-dblp-2010	226413	716460	7575 (7256.55)	7575 (7470.61)	7575 (**7575.00**)	0 (104.39)
ca-dblp-2012	317080	1049866	14108 (11623.72)	14108 (13739.21)	14108 (**14108.00**)	0 (368.79)
ca-hollywood-2009	1069126	56306653	222720 (136456.22)	222720 (206446)	222720 (**219297.42**)	0 (12851.4)
ca-MathSciNet	332689	820644	2792 (2484.43)	2792 (2518.30)	2792 (**2792.00**)	0 (273.7)
inf-roadNet-CA	1957027	2760388	691 (598.72)	668 (622.60)	**752** (**752.00**)	84 (129.4)
inf-roadNet-PA	1087562	1541514	637 (597.45)	599 (598.86)	**669** (**669.00**)	70 (70.14)
sc-ldoor	952203	20770807	4074 (3874.34)	4081 (3966.68)	4081 (**4081.00**)	0 (114.32)
sc-msdoor	415863	9378650	4088 (3966.31)	4088 (4028.97)	4088 (**4088.00**)	0 (59.03)
sc-nasasrb	54870	1311227	4548 (4429.28)	4548 (4546.56)	4548 (**4548.00**)	0 (1.44)
sc-pkustk11	87804	2565054	5298 (4794.50)	5298 (5063.51)	5298 (**5298.00**)	0 (234.49)
sc-pkustk13	94893	3260967	6306 (5751.92)	6306 (5906.58)	6306 (**6287.10**)	0 (380.52)
sc-pwtk	217891	5653221	4596 (4384.80)	4620 (4606.56)	4620 (**4620.00**)	0 (13.44)
sc-shipsec1	140385	1707759	3540 (3084.78)	3540 (3294.51)	3540 (**3540.00**)	0 (245.49)
sc-shipsec5	179104	2200076	4524 (4320.50)	4524 (4407.96)	4524 (**4524.00**)	0 (116.04)
socfb-A-anon	3097165	23667394	2872 (2150.87)	2872 (2172.79)	2872 (**2872.00**)	0 (699.21)
socfb-B-anon	2937612	20959854	2537 (1911.92)	2662 (2008.74)	2662 (**2583.94**)	0 (575.2)
socfb-OR	63392	816886	3523 (3520.88)	3523 (3516.27)	3523 (**3523.00**)	0 (6.73)
soc-brightkite	56739	212945	3672 (3652.86)	3672 (3654.87)	3672 (**3661.98**)	0 (7.11)
soc-delicious	536108	1365961	1547 (1532.90)	1547 (1535.79)	1547 (**1546.93**)	0 (11.14)
soc-digg	770799	5907132	5287 (4742.21)	5303 (4712.43)	5303 (**4839.18**)	0 (126.75)
soc-flickr	513969	3190452	7083 (7032.71)	7083 (7069.15)	7083 (**7083.00**)	0 (13.85)
soc-flixster	2523386	7918801	3805 (3389.09)	3805 (3403.42)	3805 (**3805.00**)	0 (401.58)
soc-FourSquare	639014	3214986	3064 (**3057.10**)	3064 (3035.96)	3064 (2980.31)	0 (-55.65)
soc-gowalla	196591	950327	2335 (2249.70)	2335 (**2256.22**)	2335 (2253.51)	0 (-2.71)
soc-lastfm	1191805	4519330	1773 (1770.28)	1773 (1771.50)	1773 (**1773.00**)	0 (1.5)
soc-livejournal	4033137	27933062	7238 (2312.07)	15855 (2661.46)	**21368** (**17783.19**)	5513 (15121.7)
soc-pokec	1632803	22301964	3191 (2132.10)	3191 (2008.17)	3191 (**3191.00**)	0 (1182.83)
soc-youtube-snap	1134890	2987624	1787 (1787.00)	1787 (1775.29)	1787 (**1787.00**)	0 (11.71)
tech-as-skitter	1694616	11094209	5703 (4894.34)	5703 (5076.34)	5703 (**5671.28**)	0 (594.94)
web-it-2004	509338	7178413	45477 (40088.88)	45477 (45380.95)	45477 (**45477.00**)	0 (96.05)
web-sk-2005	121422	334419	11925 (10583.83)	11925 (11892.72)	11925 (**11925.00**)	0 (32.28)
web-wikipedia2009	1864433	4507315	3823 (1752.62)	3823 (2100.06)	**3891** (**3833.28**)	68 (1733.22)

Table 2. Performances on instances where they returned the same w_{max} and w_{avg} values

Graph	Time		#Step		#Step/ms	
	LSCC	LMY-GRS	LSCC	LMY-GRS	LSCC	LMY-GRS
bio-dmela	0.006	**0.005**	**140.9**	564.2	27.995	**113.956**
bio-yeast	0.002	**0.000**	**390.9**	489.16	94.419	**745.513**
ca-AstroPh	20.896	**11.057**	425754.9	**425117**	20.744	**40.648**
ca-citeseer	202.994	**0.647**	318659.9	**147513**	1.604	**231.164**
ca-CondMat	1.928	**0.168**	**32474.4**	36398	17.702	**228.675**
ca-CSphd	0.003	**0.000**	**523.5**	1085.49	57.501	**1069.836**
ca-Erdos992	0.003	**0.000**	**172.6**	183.19	**1099**	217.888
ca-GrQc	0.062	**0.009**	4882.4	**4496.11**	80.002	**297.030**
ca-HepPh	**0.086**	0.143	**2683.7**	5721.1	**33.919**	33.179
ia-email-EU	0.227	**0.079**	2129.3	**1053.53**	9.306	**15.749**
ia-email-univ	0.000	0.000	270.5	**232.85**	199.638	**348.546**
ia-enron-large	**6.280**	7.197	52895.0	**44609.6**	**8.575**	6.443
ia-fb-messages	0.000	0.000	21.8	**23.88**	**130.787**	111.793
ia-reality	0.048	**0.012**	**1325.7**	1550.71	26.814	**117.159**
ia-wiki-Talk	0.621	**0.433**	**1855.3**	2029.29	2.841	**3.666**
inf-power	**0.027**	0.051	**932.4**	1236.47	33.649	**1412.931**
rec-amazon	3.096	**0.000**	**4858.8**	66364.1	1.618	**1642.741**
socfb-Berkeley13	**49.960**	52.664	668700.0	**551266**	**13.601**	10.657
socfb-CMU	**3.312**	11.442	**127743.3**	145167	**39.716**	13.538
socfb-Duke14	**6.649**	19.876	**11332.0**	210134	**28.680**	11.280
socfb-Indiana	102.016	**90.841**	1073259.6	**1007020**	10.611	**12.112**
socfb-MIT	**3.246**	13.241	**134955.9**	172320	**42.597**	13.587
socfb-Penn94	**104.695**	135.512	782995.3	**397520**	7.638	**8.163**
socfb-Stanford3	**15.511**	36.116	**371036.8**	397520	**24.467**	11.514
socfb-Texas84	**79.057**	149.417	**673376.8**	868678	**8.510**	6.200
socfb-UCLA	**34.333**	45.327	**511804.3**	568163	**15.415**	13.202
socfb-UConn	**18.490**	19.268	**323312.8**	333033	**17.653**	17.254
socfb-UCSB37	**21.804**	30.156	426798.7	**410292**	**20.024**	14.249
socfb-UF	**70.877**	73.057	603137.5	**513143**	**8.575**	7.671
socfb-UIllinois	158.450	**140.481**	**1539421.1**	1633500	10.039	**12.798**
socfb-Wisconsin87	42.416	**34.065**	522928.6	**479857**	12.365	**15.266**
soc-BlogCatalog	**17.210**	122.788	45587.6	**31182.9**	**2.778**	0.267
soc-buzznet	92.703	**79.329**	211468.5	**17649.3**	**2.340**	0.233
soc-douban	1.770	**0.061**	**2789.6**	3718.46	1.801	**78.276**
soc-epinions	31.984	**17.557**	**356916.5**	488393	11.578	**28.758**
soc-LiveMocha	**7.809**	15.138	**21018.2**	31450.8	**2.719**	2.322
soc-slashdot	12.845	**1.557**	51360.6	**10392.7**	4.248	**6.993**
soc-twitter-follows	8.608	**0.511**	**3994.0**	6218.97	0.518	**12.417**
soc-youtube	**23.655**	33.585	11783.2	**17391**	**0.507**	0.487
tech-as-caida2007	0.094	**0.055**	951.9	**593.18**	**11.078**	3.257
tech-internet-as	**0.525**	0.803	**3871.9**	4119.28	**7.870**	3.503
tech-p2p-gnutella	0.968	**0.012**	3354.6	**3241.77**	3.582	**317.924**
tech-RL-caida	20.520	**0.741**	29238.2	**23989.1**	1.487	**27.937**
tech-routers-rf	**0.113**	0.116	**14551.5**	15513.4	127.910	**145.814**
tech-WHOIS	**0.426**	1.108	**16010.9**	21880.5	**38.349**	19.012
web-arabic-2005	30.543	**0.408**	57788.0	**51285**	1.897	**86.461**
web-Berkstan	0.077	**0.001**	1555.3	**1502.56**	22.143	**1439.152**
web-edu	2.885	**0.012**	191629.5	**2741.87**	67.589	**211.956**
web-google	0.117	**0.006**	25789.6	**15207**	220.953	**1644.486**
web-indochina-2004	0.041	**0.011**	**874.2**	1409.67	23.025	**124.223**
web-spam	**9.608**	26.431	417141.9	**392304**	**44.426**	14.681
web-uk-2005	33.803	**0.696**	**84260.7**	93855	2.567	**133.358**
web-webbase-2001	148.073	**2.234**	2393708.0	**57063.4**	16.748	**27.596**

Table 3. The number of steps per millisecond on huge sparse instances

Graph	LSCC	LMY-GRS	Graph	LSCC	LMY-GRS
ca-hollywood-2009	0.246	**0.365**	sc-ldoor	0.310	**356.573**
socfb-A-anon	0.070	**5.261**	soc-livejournal	0.057	**25.203**
socfb-B-anon	0.059	**5.751**	soc-pokec	0.125	**20.734**
inf-roadNet-CA	0.070	**1175.015**	tech-as-skitter	0.125	**0.785**
inf-roadNet-PA	0.126	**1085.205**	web-wikipedia2009	0.099	**11.713**

7 Conclusions and Future Work

In this paper, we developed a local search MVWC solver named LMY-GRS, which significantly outperforms state-of-the-art solvers on large sparse graphs. It attains best-known solutions on all the graphs in the experiments, and it finds new best-known solutions on some of them.

Our contributions are of three folds. Firstly we rigorously showed that the swap-set is usually small even when we are solving large sparse graphs. Secondly we incorporated random walk into the multi-neighborhood greedy search and showed that it is satisfactory. Thirdly we proposed efficient data structures that work well with huge sparse graphs.

In the future we will improve SCC to further avoid cycles. Also we will introduce more diversification strategies into LMY-GRS.

Acknowledgment. We thank all anonymous reviewers for their valuable comments. This work is supported by ARC Grant FT0991785, NSF Grant No.61463044, NSFC Grant No.61572234 and Grant No.[2014]7421 from the Joint Fund of the NSF of Guizhou province of China.

We gratefully acknowledge the support of the Griffith University eResearch Services Team and the use of the High Performance Computing Cluster "Gowonda" to complete this research.

References

1. Amgalan, B., Lee, H.: Wmaxc: a weighted maximum clique method for identifying condition-specific sub-network. PLoS ONE **9**(8), e104993 (2014)
2. Babel, L.: A fast algorithm for the maximum weight clique problem. Computing **52**(1), 31–38 (1994). http://dx.doi.org/10.1007/BF02243394
3. Balasundaram, B., Butenko, S.: Graph domination, coloring and cliques in telecommunications. In: Resende, M.G.C., Pardalos, P.M. (eds.) Handbook of Optimization in Telecommunications, pp. 865–890. Springer, Heidelberg (2006)
4. Ballard, D.H., Brown, C.M.: Computer Vision, 1st edn. Prentice Hall Professional Technical Reference, New York (1982)
5. Barabasi, A.L., Albert, R.: Emergence of scaling in random networks. Science **286**(5439), 509–512 (1999). http://www.sciencemag.org/cgi/content/abstract/286/5439/509

6. Battiti, R., Protasi, M.: Reactive local search for the maximum clique problem. Algorithmica **29**(4), 610–637 (2001)
7. Bomze, I.M., Pelillo, M., Stix, V.: Approximating the maximum weight clique using replicator dynamics. IEEE Trans. Neural Netw. Learn. Syst. **11**(6), 1228–1241 (2000). http://dx.doi.org/10.1109/72.883403
8. Brendel, W., Amer, M.R., Todorovic, S.: Multiobject tracking as maximum weight independent set. In: 24th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2011, Colorado Springs, CO, USA, 20–25 June 2011, pp. 1273–1280 (2011). http://dx.doi.org/10.1109/CVPR.2011.5995395
9. Brendel, W., Todorovic, S.: Segmentation as maximum-weight independent set. In: Advances in Neural Information Processing Systems 23: 24th Annual Conference on Neural Information Processing Systems 2010. Proceedings of a meeting held 6–9 December 2010, Vancouver, British Columbia, Canada, pp. 307–315 (2010). http://papers.nips.cc/paper/3909-segmentation-as-maximum-weight-independent-set
10. Busygin, S.: A new trust region technique for the maximum weight clique problem. Discrete Appl. Math. **154**(15), 2080–2096 (2006). http://dx.doi.org/10.1016/j.dam.2005.04.010
11. Cai, S.: 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, 25–31 July 2015, pp. 747–753 (2015). http://ijcai.org/papers15/Abstracts/IJCAI15-111.html
12. Chung, F., Lu, L.: Complex Graphs and Networks, vol. 107. American Mathematical Society (2006). https://books.google.com.au/books?id=BqqDsEKlAE4C
13. Eubank, S., Kumar, V.S.A., Marathe, M.V., Srinivasan, A., Wang, N.: 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, 11–14 January 2004, pp. 718–727 (2004). http://dl.acm.org/citation.cfm?id=982792.982902
14. Fang, Z., Li, C., Qiao, K., Feng, X., Xu, K.: Solving maximum weight clique using maximum satisfiability reasoning. In: 21st European Conference on Artificial Intelligence, ECAI 2014, 18–22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014), pp. 303–308 (2014). http://dx.doi.org/10.3233/978-1-61499-419-0-303
15. Feige, U.: Approximating maximum clique by removing subgraphs. SIAM J. Discret. Math. **18**(2), 219–225 (2005). http://dx.doi.org/10.1137/S089548010240415X
16. Johnson, D.J., Trick, M.A. (eds.): Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, Workshop, October 11–13, 1993. American Mathematical Society, Boston (1996)
17. Karp, R.M.: Reducibility among combinatorial problems. In: Proceedings of a Symposium on the Complexity of Computer Computations, 20–22 March 1972. At the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, pp. 85–103 (1972). http://www.cs.berkeley.edu/luca/cs172/karp.pdf
18. Li, N., Latecki, L.J.: Clustering aggregation as maximum-weight independent set. In: Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held 3–6 December 2012, Lake Tahoe, Nevada, United States, pp. 791–799 (2012). http://papers.nips.cc/paper/4731-clustering-aggregation-as-maximum-weight-independent-set

19. Ma, T., Latecki, L.J.: Maximum weight cliques with mutex constraints for video object segmentation. In: 2012 IEEE Conference on Computer Vision and Pattern Recognition, Providence, RI, USA, 16–21 June 2012, pp. 670–677 (2012). http://dx.doi.org/10.1109/CVPR.2012.6247735
20. Miller, B.L., Goldberg, D.E.: Genetic algorithms, tournament selection, and the effects of noise. Complex Syst. **9**(3), 193–212 (1995)
21. Östergård, P.R.J.: A new algorithm for the maximum-weight clique problem. Nord. J. Comput. **8**(4), 424–436 (2001). http://www.cs.helsinki.fi/njc/References/ostergard2001:424.html
22. Pullan, W.J.: Approximating the maximum vertex/edge weighted clique using local search. J. Heuristics **14**(2), 117–134 (2008). http://dx.doi.org/10.1007/s10732-007-9026-2
23. Ravetti, M.G., Moscato, P.: Identification of a 5-protein biomarker molecular signature for predicting alzheimer's disease. PLoS ONE **3**(9), e3111 (2008)
24. Rossi, R.A., Ahmed, N.K.: Coloring large complex networks. Soc. Netw. Analys. Min. **4**(1), 228 (2014). http://dx.doi.org/10.1007/s13278-014-0228-y
25. Rossi, R.A., Ahmed, N.K.: The network data repository with interactive graph analytics and visualization. In: Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (2015)
26. Rossi, R.A., Gleich, D.F., Gebremedhin, A.H., Patwary, M.M.A.: Fast maximum clique algorithms for large graphs. In: 23rd International World Wide Web Conference, WWW 2014, Seoul, Republic of Korea, 7–11 April 2014, Companion Volume, pp. 365–366 (2014). http://doi.acm.org/10.1145/2567948.2577283
27. Wang, Y., Cai, S., Yin, M.: Two efficient local search algorithms for maximum weight clique problem. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, 12–17 February 2016, Phoenix, Arizona, USA, pp. 805–811 (2016). http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/11915
28. Wu, Q., Hao, J., Glover, F.: Multi-neighborhood tabu search for the maximum weight clique problem. Ann. OR **196**(1), 611–634 (2012). http://dx.doi.org/10.1007/s10479-012-1124-3
29. Xu, K., Boussemart, F., Hemery, F., Lecoutre, C.: A simple model to generate hard satisfiable instances. In: Proceedings of the 19th International Joint Conference on Artificial Intelligence, IJCAI 2005, pp. 337–342. Morgan Kaufmann Publishers Inc., San Francisco (2005). http://dl.acm.org/citation.cfm?id=1642293.1642347
30. Yamaguchi, K., Masuda, S.: A new exact algorithm for the maximum weight clique problem. In: ITC-CSCC: 2008, pp. 317–320 (2008)