Efficient Local Search for Minimum Dominating Sets in Large Graphs

Yi Fan[1,2,3], Yongxuan Lai[1(✉)], Chengqian Li[4], Nan Li[5], Zongjie Ma[3], Jun Zhou[3], Longin Jan Latecki[5], and Kaile Su[3]

[1] Shenzhen Research Institute/Software School, Xiamen University, Xiamen, China
yifan.sysu@gmail.com, laiyongxuan@gmail.com
[2] Guangxi Key Lab of Trusted Software, Guilin University of Electronic Technology, Guilin, China
[3] Griffith University, Brisbane, Australia
zongjie.ma@griffithuni.edu.au, {jun.zhou,k.su}@griffith.edu.au
[4] NetEase Guangzhou AI Lab, Guangzhou, China
lcqn2776@corp.netease.com
[5] Temple University, Philadelphia, USA
{nan.li,latecki}@temple.edu

Abstract. The Minimum Dominating Set (MinDS) problem is an NP-hard problem of great importance in both theories and applications. In this paper, we propose a new local search algorithm ScBppw (Score Checking and Best-picking with Probabilistic Walk) to solve the MinDS problem in large graphs. For diversifying the search, our algorithm exploits a tabu strategy, called Score Checking (SC), which forbids a vertex to be added into the current candidate solution if the vertex's score has not been changed since the last time it was removed out of the candidate solution. Also, to keep a good balance between intensification and diversification during the search, we propose a strategy that combines, in a novel way, best-picking with probabilistic walk at removing stages. At this stage, the algorithm selects a vertex with the minimum loss, or other vertices in the candidate solution with a probability proportional to the their degrees, depending on how repeatedly the area has been visited. Experimental results show that our solver significantly outperforms state-of-the-art MinDS solvers. Also we conducted several experiments to show the individual impacts of our novelties.

Keywords: Minimum dominating set · Score-based partition · Best-picking

1 Introduction

Given a simple undirected graph $G = (V, E)$, a dominating set is a subset $D \subseteq V$ s.t. every vertex outside D has at least one neighbor in D. The Minimum

Supported by the Natural Science Foundation of China (61672441, 61872154), the Shenzhen Basic Research Program (JCYJ20170818141325209), and the NSF grants (IIS-1814745, IIS-1302164).

Dominating Set (MinDS) problem is to find a dominating set of the minimum size. MinDS arises in many application areas, *e.g.*, biological networks [20], metro networks [1], power networks [13], computer vision [21,26], multi-document summarization [23], and wireless communication [22]. Among various algorithms, *e.g.* [6,10], local search is an effective approach and it is powerful on difficult graphs.

1.1 Local Search Techniques

Local search often suffers from the cycling problem, *i.e.*, the search may spend too much time visiting a small part of the search space, thus various tabu strategies have been proposed to deal with this problem. Recently [5] proposed the so-called configuration checking (CC) strategy, and after that numerous variants of the CC strategy have been adopted to solve a wide range of combinatorial problems *e.g.*, [11,24,25]. The idea of the CC strategy can be described as follows. If a vertex is removed out of the candidate solution, then it is forbidden to be added back until its configuration is changed, *i.e.*, some neighboring vertex is added or removed out of the candidate solution.

The forbidding strength of CC can be too weak, which may lead a CC-based local search being stuck in a cycle. To escape from such a cycle, a CC-based local search usually needs some other diversifying strategies like constraint weighting [15], which is unfortunately time-consuming and impractical for large graphs. For diversifying the search, we exploit a strong CC-like tabu strategy, called Score Checking (SC), which forbids a vertex to be added into the current candidate solution if its score has not been changed since the last time it was removed out of the candidate solution. In this strategy, when a vertex is added or removed, some vertices in the neighborhood are released, while some others are still forbidden. This is different from usual CC variants.

To keep a good balance between intensification and diversification, we propose a strategy that combines, in a novel way, best-picking with probabilistic walk at removing stages. Best-picking with random walk has proved to be effective in solving the minimum vertex cover problem in large graphs [18]. More specifically, in greedy mode it chooses small-degree vertices while in random mode it selects a vertex from a uniform distribution. However, the strategy may still focus too much on small-degree vertices, in other words, there may not be enough chances to select a big-degree vertex. Therefore we adopt a probabilistic distribution to remove vertices in random mode. To be specific, a vertex is selected to be removed with a probability proportional to its degree. Evidently, this strategy gives more chances to big-degree vertices.

Also, when we are doing local search in large instances, the complexity becomes a big issue. In each step there can be millions of possible moves, and thus it is difficult to obtain a local move which maximizes certain kinds of benefits. Since most real-world graphs are sparse [3,8,10], it is beneficial to develop data structures so that the complexity of a single search step relies on the average degree rather than the number of vertices. For instance, a local search solver LMY-GRS [11] follows this idea and finds good solutions for the maximum weight clique problem. Inspired by this, we propose an efficient data structure named score-based partition to implement best-picking. The score-based partition effectively cuts off the average complexity of exchanging vertices in the local search algorithm.

1.2 Our Contributions

In this paper, we develop a local search solver named ScBppw (Score Checking and Best-picking with Probabilistic Walk) based on the strategies above. For showing the effectiveness, we compare ScBppw with FastWMDS [24] on large graphs[1]. Experimental results show that (1) our solver ScBppw significantly outperforms FastWMDS on large graphs; (2) our proposed strategies play an important role in our solver.

2 Preliminaries

2.1 Basic Notations

We use $N(v) = \{u \in V | \{u, v\} \in E\}$ to denote the set of v's neighbours, and we use $N[v]$ to denote $N(v) \cup \{v\}$. The degree of a vertex v, denoted by $d(v)$, is defined as $|N(v)|$. We use

$$N^2(v) = (N(v) \cup \{u | \{u, w\} \in E \,\text{and}\, w \in N(v)\}) \backslash \{v\}$$

to denote the set of vertices whose distance from v is at most 2. Also we use $\bar{d}_2(G)$ to denote the average size of $N^2(v)$ over all the vertices, *i.e.*, $\bar{d}_2(G) = \frac{1}{|V|} \sum_{v \in V} |N^2(v)|$, suppressing G if understood from the context. In graph theory, we have the following proposition that is useful in implementing our solver.

Proposition 1. $\sum_{v \in V} d(v) = 2|E|$.

A vertex is said to be covered by a set D if it is in D or at least one of its neighbors is in D. Otherwise it is said to be uncovered by D. If u's removal from D makes v become uncovered, we also say that u's removal uncovers v. Likely if u's addition into D makes v become covered, we also say that u's addition covers v. For a vertex $v \in D$, the *loss* of v, denoted as $loss(v)$, is defined as the number of covered vertices that will become uncovered by removing v from D. For a vertex $v \notin D$, the *gain* of v, denoted as $gain(v)$, is defined as the number of uncovered vertices that will become covered by adding v into D. Both *gain* and *loss* are scoring properties. Obviously we have

Proposition 2. *For all* $v \in V$, $gain(v) \in [0, d(v) + 1]$, $loss(v) \in [0, d(v) + 1]$.

In MinDS solving, we have a proposition below which shows the set of vertices whose score needs to be updated.

[1] http://networkrepository.com/networks.php.

```
Algorithm 1: LocalSearchForMinDS($G$, $cutoff$)
input  : A graph $G = (V, E)$, the $cutoff$ time
output: A dominating set of $G$
1 $D \leftarrow$ InitDS($G$);
2 while $elapsed\ time < cutoff$ do
3     if $D$ covers all vertices then
4         $D^* \leftarrow D$;
5         remove a vertex from $D$ with the minimum $loss$, breaking ties randomly;
6         $lastStepImproved \leftarrow false$;
7     else
8         ExchangeVertices($G, D$);
9 return $D^*$;
```

Proposition 3. *1. When a vertex u is added or removed, for any vertex $v \notin N^2(u)$, $gain(v)/loss(v)$ remains unchanged.*
2. If $gain(v)$ or $loss(v)$ is changed, then at least one vertex $u \in N^2(v)$ has been added or removed.

In any step, a vertex v has two possible states: inside D and outside D. We use $age(v)$ to denote the number of steps that have been performed since last time v's state was changed.

2.2 An Iterative Local Search Framework

Algorithm 1 presents a local search framework for MinDS. It consists of the construction phase (Line 1) and the local search phase (Lines 2 to 8).

In our algorithm, we will adopt a simple greedy strategy to implement `InitDS`(G), which works as follows. *Given an empty set D, repeat the following operations until D becomes a dominating set: select a vertex $v \notin D$ with the maximum gain and add v into D, breaking ties randomly.* Actually we can use the data reduction rules in [2] to generate better initial solutions and more importantly simplify the input graph. Yet currently we do not do so, because our main concern in this paper is to develop an efficient local search algorithm.

In the local search phase, each time a k-sized dominating set is found (Line 3), the algorithm removes a vertex from D (Line 5) and proceeds to search for a $(k-1)$-sized dominating set, until a certain time limit is reached (Line 2). A local move consists of exchanging a pair of vertices (Line 8): a vertex $u \in D$ is removed from D, and a vertex $v \notin D$ is added into D. Such an exchanging procedure is also called a step or an iteration by convention. In our algorithm, we also employ a predicate *lastStepImproved* s.t. *lastStepImproved* $=$ *true* iff the number of uncovered vertices was decreased in the last step. This predicate will be used in the best-picking with probabilistic walk strategy we propose. Lastly, when the algorithm terminates, it outputs the smallest dominating set that has been found.

2.3 A Fast Hashing Function

In order to detect revisiting, we will use the hash function in [12] which is shown below.

Definition 1. *Given a candidate set D and a prime number p, we define the hash value of D, denoted by $hash(D)$, as $\left(\sum_{v_i \in D} 2^i\right) \mod p$, which maps a candidate set D to its hash entry $hash(D)$.*

Also, they showed that each time a vertex is added or removed, the hash value can be updated in $O(1)$ complexity.

2.4 Geometric Distribution

To analyze the complexity of our algorithm, we first introduce Geometric Distribution which depicts the probability that the first success occurs in a particular trial.

Definition 2. *A random variable X has a geometric distribution if the probability that the k^{th} trial (out of k trials) is the first success is $Pr(X = k) = (1-p)^{k-1}p$, where k is a positive integer and p denotes the probability of success in a single trial.*

Then the average number of trials needed for the first success is $\frac{1}{p}$ according to the following theorem [19].

Theorem 1. *If X is a geometric random variable with parameter p, then the expected value $E(X)$ is given by $\frac{1}{p}$.*

2.5 Sorting Vertices wrt. Degrees

Our algorithm implementation requires sorting vertices into non-decreasing order wrt. their degrees in advance, so we introduce an efficient sorting algorithm as follows. Considering that $0 \leq d(v) \leq |V|$ for any $v \in V$, this satisfies the assumption of counting sort which runs in linear time [9]. So we have

Proposition 4. *Sorting the vertices into non-decreasing order wrt. their degrees can be done in $O(|V|)$ complexity.*

2.6 Configuration Checking

In MinDS solving, there are three different CC strategies, *i.e.*, neighboring vertices based CC (NVCC) [5], two-level CC (CC^2) [25] and three-valued two-level CC (CC^{2}V3). In NVCC, the configuration of a vertex v refers to the state of the vertices in $N(v)$. On the other hand, in CC^2, the configuration of a vertex v refers to the state of the vertices in $N^2(v)$.

We use V_{NVCC} to denote the set of vertices whose configuration is changed according to the NVCC strategy, and use V_{CC^2} to denote the set of vertices whose configuration is changed according to the CC^2 strategy. [25] presented the proposition below which shows that the forbidding strength of NVCC is stronger than that of CC^2.

Proposition 5. *If $V_{NVCC} = V_{CC^2}$ in some step, then $V_{NVCC} \subsetneq V_{CC^2}$ in the next step.*

3 Score Checking

We first define the notion of reversing operations and then propose a tabu strategy which is based on score change.

Effects of a Local Move. Given a vertex v, there are two possible operations: removing v from D and adding v into D, and we call them a pair of *reversing operations*. Adding a vertex into D has two types of effects: (1) turning some vertices from being uncovered to being covered by 1 vertex; (2) turning some vertices from being covered by $c(c \geq 1)$ vertices to being covered by $(c+1)$ vertices. Likely, a removing operation has analogous effects. Given a pair of reversing operations in a sequence of local search steps, we say that their effects are *neutralized* if and only if they have completely opposite effects.

Obviously if a pair of reversing operations occur consecutively, their effects will be neutralized. This is the worst case and any tabu strategy prevents such cases from happening. However, even though two reversing operations are not consecutive, their effects may still be neutralized. So the question is in under what condition their effects will not be neutralized.

Our Tabu Strategy. In fact it is impractical to record all the effects of an operation, since it requires much space and time for checking. To make it practicable, a compromising method is to memorize some important effects like *the set of vertices which become covered (or uncovered)*. Then we avoid any two reversing operations which cover and uncover the same set of vertices.

However, maintaining these sets and checking equality relation still consumes too much time. Therefore we choose to consider the score (gain or loss) of the operations. Then we attempt to avoid any pair of reversing operations if their scores are opposite. Now we give an example to describe our motivation.

Example 1. Suppose a vertex v is removed from the candidate solution at Step i with $loss_i(v)$, and then added back at Step j with $gain_j(v)$. We observe that

1. if $loss_i(v) \neq gain_j(v)$, then the effects of these two operations will not be neutralized;
2. even though $loss_i(v) = gain_j(v)$, their effects are not necessarily neutralized, since they may cover and uncover different sets of vertices respectively;
3. if $gain(v)$ keeps unchanged after v's removal, then v's removal and later addition will uncover and cover the same set of vertices respectively.

Based on these observations, we propose a tabu strategy which is based on score change: *After a vertex is removed from the candidate solution, it cannot be added back until its score has been changed.* Throughout this paper, this strategy will be called Score Checking (SC).

Our strategy is implemented with a Boolean array named *free* s.t. $\mathit{free}(v) = 1$ means v is allowed to be added into the candidate solution, and $\mathit{free}(v) = 0$ otherwise. Then we maintain the *free* array as follows:

1. initially $\mathit{free}(v)$ is set to 1 for all $v \in V$;
2. when v is removed from D, $\mathit{free}(v)$ is set to 0;
3. when $\mathit{gain}(v)$ is changed, $\mathit{free}(v)$ is set to 1.

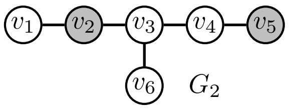
We notice that there is a concept called *promising variable* in SAT [17], which allows a variable to be flipped if its score becomes positive because of the flips of its neighboring variables. This concept is in some sense similar to the score checking strategy here.

3.1 Comparing SC to NVCC

Example 2 shows that SC has stronger forbidding strength than NVCC, while Example 3 shows the opposite.

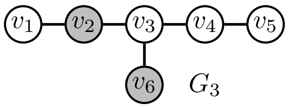
Example 2. Suppose that we have a graph G_2 as below, the current candidate solution $D = \{v_2, v_5\}$, $\mathit{free}(v_3) = 1$ and $\mathit{free}(v_4) = 0$.

Now we add v_3 into D. According to NVCC, $\mathit{free}(v_4)$ will become 1, because one of its neighbors, v_3, has changed its state. However, according to SC, $\mathit{free}(v_4)$ will still be 0, because $\mathit{gain}(v_4)$ has not changed. So the forbidding strength of SC is stronger than that of NVCC in this case.



Example 3. Suppose that we have a graph G_3 as below, the current candidate solution $D = \{v_2, v_6\}$, $\mathit{free}(v_3) = 1$ and $\mathit{free}(v_4) = \mathit{free}(v_5) = 0$.

Now we add v_3 into D. According to NVCC, $\mathit{free}(v_5)$ will still be 0, because none of its neighbors has changed its state. However, according to SC, $\mathit{free}(v_5)$ will become 1, because $\mathit{gain}(v_5)$ is decreased by 1. So the forbidding strength of NVCC is stronger than that of SC in this case.



Based on the examples above, we have

Proposition 6. *There exists a case in which the forbidding strength of SC is stronger than that of NVCC, and there also exists a case in which the forbidding strength of NVCC is stronger than that of SC.*

3.2 Comparing SC to CC^2

By Item 2 in Proposition 3, if a vertex's score is changed, then it must be that its configuration is changed according to CC^2 strategy. Therefore if a vertex is released by SC, it must also be released by CC^2. So the forbidding strength of SC is at least as strong as that of CC^2. On the other hand, Example 2 shows that the forbidding strength of SC can sometimes be stronger than that of NVCC, and thus also stronger than that of CC^2 by Proposition 5. Therefore we have a proposition below which shows that the forbidding strength of SC is strictly stronger than that of CC^2.

Proposition 7. *Let V_{SC} be the set of vertices which are free according to SC, then we have (1) $V_{SC} \subseteq V_{CC^2}$ always hold; (2) $V_{CC^2} \subseteq V_{SC}$ does not always hold.*

4 Our Algorithm for Exchanging Vertices

Here, we present the `ExchangeVertices`(G, D) procedure in Algorithm 2, which adopts our new tabu strategy. In this algorithm, we use $uncov\_v\_num_1$ and

Algorithm 2: ExchangeVertices

```
1  $uncov\_v\_num_1 \leftarrow$ the number of uncovered vertices;
2  if $lastStepImproved$ then
3      if $hash(D)$ is marked then
4          $u \leftarrow$ a vertex in $D$ with a probability proportional to its
           degree;// random mode
5      else
6          mark $hash(D)$;
7          $u \leftarrow$ a vertex in $D$ with the minimum $loss$, breaking ties
           randomly;// greedy mode
8  else
9      $u \leftarrow$ a vertex in $D$ with the minimum $loss$, breaking ties
       randomly;// greedy mode
10 remove $u$ from $D$;
11 $free(u) \leftarrow 0$;
12 $free(x) \leftarrow 1$ for all $x$ s.t. $gain(x)$ is changed;
13 $v \leftarrow$ a random uncovered vertex;
14 $w \leftarrow$ a vertex $u$ in $N[v]$ s.t. $free(u) = 1$ with the greatest $gain$, breaking ties in
   favor of the greatest $age$;
15 add $w$ into $D$;
16 $free(y) \leftarrow 1$ for all $y$ s.t. $gain(y)$ is changed;
17 $uncov\_v\_num_2 \leftarrow$ the number of uncovered vertices;
18 if $uncov\_v\_num_2 < uncov\_v\_num_1$ then
19     $lastStepImproved \leftarrow true$;
20 else
21     $lastStepImproved \leftarrow false$;
```

$uncov\_v\_num_2$ to denote the number of uncovered vertices before and after exchanging vertices respectively. Furthermore, it adopts the two-stage exchange strategy in NuMVC [4].

4.1 Removing Stage

In the removing stage (Lines 2 to 12), we propose a heuristic called *best-picking with probabilistic walk*. The intuition is as follows: if the local search has spent only a little time in the current area, we prefer greedy mode to find a good solution; otherwise we favor random mode to leave.

To be specific, we exploit the hash function in the preliminary section to approximately detect revisiting. Like [12], we set the prime number to be $10^9 + 7$ and do not resolve collisions. Since in our experiments, our solver performs less than 5×10^8 steps in any run, given the $10^9 + 7$ hash entries, the number of collisions is negligible. Notice that we mark and detect the hash table only when $lastStepImproved = true$. The intuition is that if a solution is revisited together with the same *lastStepImproved* value as before, then the current area has been visited to a significant extent. In this situation, a vertex is removed from the candidate solution with a probability proportional to its degree. We tend to remove large-degree vertices, because we rarely remove them in the greedy mode, and this will further strengthen the diversification in our algorithm.

To implement probabilistic selections, we employ an array with length $2|E|$, where $|E|$ is the number of edges, to store copies of the vertices. Before doing local search, given any vertex, say u, we put $d(u)$ copies of u into the array. According to Proposition 1, there are exactly $2|E|$ copies of the vertices in the array. When we perform the probabilistic selection, we do the following: *Randomly select an item in the array and if the result is in D, then return; otherwise repeat the random selection.*

Now we analyze the complexity of this procedure. Each time we obtain an item, the probability that it is in D, denoted by p, is at least $\frac{|D|}{2|E|}$ (we use "at least" because usually a vertex in D has more than one copy in the array). Then by Theorem 1, the averaged number of trials needed for the first success is $\frac{1}{p} \leq \frac{2|E|}{|D|}$. Moreover, we find that $\frac{2|E|}{|D|} \leq 500$ in most graphs. So the time complexity is $O(\frac{2|E|}{|D|}) = O(1)$.

4.2 Adding Stage

In the adding stage (Lines 13 to 16), our algorithm chooses a random uncovered vertex and selects a vertex in its neighborhood to add into D. By the following proposition we ensure that Line 14 in Algorithm 2 always returns a valid vertex outside D[2].

[2] In some graphs there are vertices whose degree is 0. In these cases, we prevent such vertices from being removed.

Proposition 8. *1. If v is uncovered, then $N[v] \cap D = \emptyset$;*
2. For any uncovered vertex v s.t. $d(v) \neq 0$, there exists at least one vertex $n \in N[v]$ s.t. free$(n) = 1$.

The proof follows the arguments in [4].

5 Speeding up of Best-Picking

In this section we present the details of our data structure that achieves the average complexity of `ExchangeVertices`(G, D) (Algorithm 2) of $O(\bar{d}_2)$.

Score-Based Partition. The 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 Fig. 1). Given a graph $G = (V, E)$ and a candidate solution D, we implement the score-based partition on an array where each position holds a vertex. Besides, we maintain two types of pointers, *i.e.*, P_{gain} and P_{loss}, each of which points to the beginning of a specific partition.

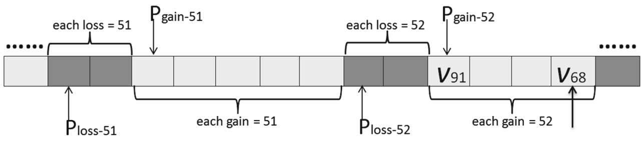


Fig. 1. Initial state before adding v_{68} into D

Algorithms and Implementations. We use *loss-k* (resp. *gain-k*) partition to denote the partition that contains vertices whose loss (resp. gain) is k. All the *loss-k* partitions are shown as dark regions, and all the *gain-k* partitions are shown as light ones. Then we use Algorithm 3 to find those vertices with the minimum loss.

Algorithm 3: randomMinLossVertex

```
input  : A sequence of score-based partitions
output: A random vertex $v \in D$ with the minimum loss
1  $k \leftarrow 0$;
2  while the loss-k partition is empty do  $k \leftarrow k + 1$
   return a random vertex in the loss-k partition;
```

Since $loss(v) \leq d(v) + 1$ for any $v \in V$, we have

Proposition 9. *If Algorithm 3 returns a vertex v, then the complexity of its execution is $O(d(v))$.*

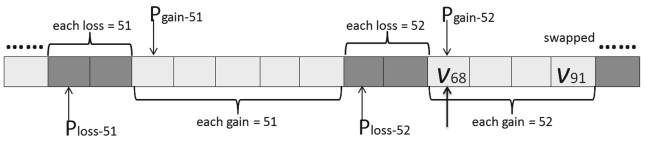


Fig. 2. Swapping vertex v_{68} and v_{91}

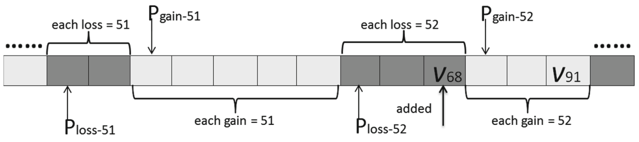


Fig. 3. Moving pointer $P_{gain-52}$

At the beginning when D is empty, there are no dark regions in our data structure, so initializing the partitions is equivalent to sorting the vertices into a non-decreasing order wrt. their *gain*. Notice that at this time, $gain(v) = d(v)+1$ for all $v \in V$. By Proposition 4, we have

Proposition 10. *The complexity of initializing the partitions is $O(|V|)$.*

In both construction and local search phases, our algorithm will repeatedly add vertices into D or remove vertices from D, until some cutoff arrives. When a vertex v is added or removed, we have to maintain the *gain* or *loss* wrt. the vertices in $N^2(v)$. 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) D; (2) increasing/decreasing $gain(v)/loss(v)$. 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 Figs. 1, 2 and 3). In this example, we are to add v_{68} into D. Before v_{91} and v_{68} are in the *gain*-52 partition and thus their *gain* is 52 (Fig. 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} (Fig. 2); (2) $\mathsf{P}_{\text{gain-52}}$ is moved (Fig. 3).

We define `placeVertexIntoD`(v) as the procedure that moves v from some *gain-k* partition to the respective *loss-k* partition. Also we define `dec_gain_move`(v) to be the procedure that moves v from some *gain-k* partition to the respective *gain*-(k-1) partition. Likely we define `dec_loss_move`(v), `inc_gain_move`(v), `inc_loss_move`(v) and `placeVertexOutFromD`(v). Then we have

Proposition 11. *All the procedures above are of $O(1)$ complexity.*

Complexity Analysis. When a vertex v is added or removed, we simply need to maintain the *gain* or *loss* wrt. the vertices in $N^2(v)$. Hence we have

Theorem 2. *Suppose that each vertex has equal probability to be added or removed, then the average complexity of* `ExchangeVertices`*(G, D) (Algorithm 2) is* $O(\bar{d}_2)$.

Table 1. Results on large graphs where FastWMDS and ScBppw found different D_{min} or D_{avg}

Graph	FastMWDS $D_{min}(D_{avg})$	ScBppw $D_{min}(D_{avg})$	δ_{min} (δ_{avg})
aff-orkut-user2groups	796325 (796405.9)	**791051** (**791057.0**)	5274 (5348.9)
bn-human-BNU_1	**1189958** (**1189968.3**)	1190244 (1190256.1)	−286 (−287.8)
ca-coauthors-dblp	**35626** (**35634.3**)	37060 (37090.3)	−1434 (−1456.0)
ca-hollywood-2009	50752 (50928.2)	**50334** (**50364.5**)	418 (563.7)
ca-MathSciNet	**65572** (**65572.0**)	65594 (65598.0)	−22 (−26.0)
channel-500	403347 (405584.3)	**392298** (**392409.6**)	11049 (13174.7)
dbpedia-link*	1537046 (1537073.6)	**1536656** (**1536657.3**)	390 (416.3)
delaunay_n22	744846 (746514.5)	**689101** (**689191.1**)	55745 (57323.4)
delaunay_n23	1514747 (1516652.2)	**1378214** (**1378400.0**)	136533 (138252.2)
delaunay_n24*	3044234 (3045694.7)	**2756143** (**2756488.0**)	288091 (289206.7)
friendster	656866 (656992.3)	**656464** (**656466.4**)	402 (525.9)
hugebubbles-00020*	7464363 (7464956.0)	**6809676** (**6810434.5**)	654687 (654521.5)
hugetrace-00010*	3849625 (3850125.7)	**3392608** (**3393012.3**)	457017 (457113.4)
hugetrace-00020*	5111615 (5112821.7)	**4508857** (**4509283.6**)	602758 (603538.1)
inf-europe_osm*	18854839 (18855830.6)	**17007576** (**17009440.0**)	1847263 (1846390.6)
inf-germany_osm*	4236871 (4238019.7)	**3846526** (**3846836.6**)	390345 (391183.1)
inf-roadNet-CA	628942 (632080.1)	**595745** (**595800.1**)	33197 (36280.0)
inf-roadNet-PA	345733 (346350.5)	**332071** (**332138.1**)	13662 (14212.4)
inf-road-usa*	8400680 (8401874.2)	**7852863** (**7853454.2**)	547817 (548420.0)
rec-epinions	**9598** (**9598.0**)	9599 (9599.0)	−1 (−1.0)
rgg_n_2_23_s0	739203 (741106.5)	**687540** (**687945.5**)	51663 (53161.0)
rgg_n_2_24_s0	4102680 (4103927.2)	**4006264** (**4006471.4**)	96416 (97455.8)
rt-retweet-crawl	75740 (**75740.0**)	75740 (75740.1)	0 (−0.1)
sc-ldoor	**62473** (**62484.7**)	64912 (66171.9)	−2439 (−3687.2)
sc-pwtk	**4194** (**4197.8**)	5479 (5504.5)	−1285 (−1306.7)
sc-rel9	**119531** (**120439.9**)	124304 (129925.1)	−4773 (−9485.2)
soc-delicious	**55722** (**55722.0**)	55726 (55727.8)	−4 (−5.8)
soc-digg	66155 (**66155.0**)	66155 (66156.6)	0 (−1.6)
soc-flickr	**98062** (**98062.3**)	98063 (98064.9)	−1 (−2.6)
soc-flickr-und	295773 (295790.9)	**295702** (**295705.1**)	71 (85.8)
soc-FourSquare	60982 (60985.7)	**60979** (**60979.0**)	3 (6.7)
soc-groups	1072250 (1072306.7)	**1071123** (**1071124.0**)	1127 (1182.7)
soc-ljournal-2008	1015711 (1015933.3)	**1005983** (**1005988.2**)	9728 (9945.1)

(contniued)

Table 1. *(contniued)*

Graph	FastMWDS $D_{min}(D_{avg})$	ScBppw $D_{min}(D_{avg})$	δ_{min} (δ_{avg})
soc-orkut-dir	98716 (99715.8)	**94012** (**94037.5**)	4704 (5678.3)
soc-pokec	213149 (213241.0)	**207383** (**207389.7**)	5766 (5851.3)
soc-youtube-snap	213122 (213131.0)	213122 (**213123.3**)	0 (7.7)
socfb-A-anon	201691 (201699.5)	**201690** (**201690.5**)	1 (9.0)
socfb-B-anon	187030 (187030.2)	187030 (**187030.1**)	0 (0.1)
tech-as-skitter	182427 (182736.4)	**181852** (**181869.8**)	575 (866.6)
tech-ip	160 (161.2)	**156** (**157.1**)	4 (4.1)
tech-RL-caida	**40095** (**40095.8**)	40142 (40152.9)	−47 (−57.1)
web-it-2004	**32997** (**32997.0**)	32998 (32999.3)	−1 (−2.3)
web-wikipedia2009	348003 (348024.5)	**346676** (**346682.4**)	1327 (1342.1)
web-wikipedia-growth	117626 (117663.0)	**116817** (**116818.5**)	809 (844.5)
web-wikipedia_link_it	618963 (619083.8)	**617660** (**617661.3**)	1303 (1422.5)
wikipedia_link_en*	23995928 (23995933.6)	**23995924** (**23995924.0**)	4 (9.6)

6 Experimental Evaluations

We compared the overall performances of our solver ScBppw to FastWMDS on the benchmark instances in [24] and [16] which contain more than 5×10^5 vertices. Notice that large graphs are the main challenge of current solvers, and their size will keep growing due to the amount of data available. There are some other MinDS solvers like SAMDS [14], CC^{2}FS [25] and RLS-DS [7], but they do not materially change our conclusions below, because our preliminary experiments show that FastWMDS significantly outperforms them. Also we conducted experiments to evaluate the individual impacts.

6.1 Experimental Setup

FastWMDS was compiled by g++ 5.4.0 with O3 option while ScBppw was compiled by g++ 4.7.3 with O3 option. The experiments were conducted on a cluster equipped with Intel Xeon E5-2670 v3 2.3 GHz with 64 GB RAM, running CentOS6. Each solver was executed on each instance with seeds from 1 to 10. The cutoff was set to 1,000s. For each algorithm on each instance, we report the minimum size ("D_{min}") and averaged size ("D_{avg}") of the dominating sets found by the algorithm over the 10 runs. To make the comparisons clearer, we also report the difference ("δ_{min}") between the minimum size returned by ScBppw and that returned by other solvers. Similarly δ_{avg} represents the difference between the averaged sizes. A positive difference indicates that ScBppw performs better, while a negative value indicates the opposite. For the sake of space, we do not report results on those graphs where all solvers found the same D_{min} and D_{avg}. The best D_{min} and D_{avg} values are shown in **bold** font. For the sake of space, in each table we will write *bn-human-BNU_1* in short for *bn-human-BNU_1_0025865_session_1-bg*, *channel-500* in short for *channel-500x100x100-b050*, and *soc-groups* in short for *soc-livejournal-user-groups.*

6.2 Main Results

Table 1 compares the overall performances of FastWMDS and ScBppw. From this table, we observe that:

1. ScBppw outperforms FastWMDS on most instances in terms of D_{min} and D_{avg}.
2. Further observations show that there are 11 graphs which contain more than 10^7 vertices. Among these graphs, ScBppw outperforms FastWMDS on 9 graphs (marked with * in the table) in terms of both D_{min} and D_{avg}. On the other 2 graphs, *i.e.*, `soc-sinaweibo` and `socfb-uci-uni`, both solvers found the same D_{min} and D_{avg}. So ScBppw is more scalable than FastWMDS.

We also did similar experiments on those graphs in [24] and [16] whose vertex number lies between 2×10^4 and 5×10^5. Experimental results show that ScBppw is comparable and complementary with FastWMDS. More specifically, they have different performances, and the number of graphs on which ScBppw performs better is roughly equal to the number of graphs on which FastWMDS performs better. In addition, both solvers outperforms the other MinDS solvers.

Notice that FastWMDS exploits a list of reduction rules to simplify the input graphs before doing local search, while our solver only adopts a simple greedy heuristic. Therefore we can conclude that pure local search is also competitive.

6.3 Individual Impacts

In what follows, we first modify our algorithm and develop several variants. Then we redo the experiments above and compare them in terms of average solution quality.

Effectiveness of Score Checking. We replace the SC strategy in ScBppw with the NVCC, CC^2 and CC^{2}V3 strategies, and develop three variants named NVCCBppw, CC^{2}Bppw and CC^{2}V3Bppw respectively.

When comparing NVCCBppw with ScBppw, we find that

1. ScBppw performs better on 19 instances;
2. NVCCBppw performs better on 14 instances.

This means that the SC strategy is more suitable than the NVCC strategy in our solver. The detailed results are shown in Table 2.

Furthermore, when comparing CC^{2}Bppw, CC^{2}V3Bppw with ScBppw, we find that

1. CC^{2}Bppw outperforms the other solvers on 0 instances;
2. CC^{2}V3Bppw outperforms the others on 6 instances;

Table 2. Results on large graphs where NVCCBppw and ScBppw found different D_{min} or D_{avg}

Graph	NVCCBppw $D_{min}(D_{avg})$	ScBppw $D_{min}(D_{avg})$	δ_{min} (δ_{avg})
aff-orkut-user2groups	791055 (791063.7)	**791051** (**791057.0**)	4 (6.7)
ca-coauthors-dblp	37069 (37095.3)	**37060** (**37090.3**)	9 (5.0)
ca-hollywood-2009	50418 (50445.9)	**50334** (**50364.5**)	84 (81.4)
delaunay_n22	**689100** (**689190.7**)	689101 (689191.1)	−1 (−0.4)
delaunay_n23	**1378207** (**1378398.3**)	1378214 (1378400.0)	−7 (−1.7)
delaunay_n24	**2756090** (**2756470.0**)	2756143 (2756488.0)	−53 (−18.0)
hugebubbles-00020	6809994 (6810453.1)	**6809676** (**6810434.5**)	318 (18.6)
hugetrace-00010	3392673 (3393040.0)	**3392608** (**3393012.3**)	65 (27.7)
hugetrace-00020	4508901 (4509384.1)	**4508857** (**4509283.6**)	44 (100.5)
inf-europe_osm	**17006505** (**17009166.9**)	17007576 (17009440.0)	−1071 (−273.1)
inf-germany_osm	**3846466** (**3846831.3**)	3846526 (3846836.6)	−60 (−5.3)
inf-roadNet-CA	595868 (595938.4)	**595745** (**595800.1**)	123 (138.3)
inf-roadNet-PA	332093 (332190.3)	**332071** (**332138.1**)	22 (52.2)
inf-road-usa	7855630 (7856923.6)	**7852863** (**7853454.2**)	2767 (3469.4)
rgg_n_2_23_s0	**687538** (**687944.9**)	687540 (687945.5)	−2 (−0.6)
rgg_n_2_24_s0	4006264 (**4006469.2**)	4006264 (4006471.4)	0 (−2.2)
sc-ldoor	**64485** (**65997.7**)	64912 (66171.9)	−427 (−174.2)
sc-rel9	**124210** (**129763.8**)	124304 (129925.1)	−94 (−161.3)
soc-delicious	55726 (**55727.4**)	55726 (55727.8)	0 (−0.4)
soc-digg	66155 (**66156.5**)	66155 (66156.6)	0 (−0.1)
soc-flickr	98064 (98065.7)	**98063** (**98064.9**)	1 (0.8)
soc-FourSquare	60979 (60979.4)	60979 (**60979.0**)	0 (0.4)
soc-orkut-dir	**93999** (**94032.2**)	94012 (94037.5)	−13 (−5.3)
soc-pokec	207559 (207597.9)	**207383** (**207389.7**)	176 (208.2)
soc-youtube-snap	213122 (213123.5)	213122 (**213123.3**)	0 (0.2)
socfb-A-anon	201690 (201690.6)	201690 (**201690.5**)	0 (0.1)
socfb-B-anon	187030 (187030.3)	187030 (**187030.1**)	0 (0.2)
tech-as-skitter	181874 (181890.1)	**181852** (**181869.8**)	22 (20.3)
tech-ip	**155** (**156.7**)	156 (157.1)	−1 (−0.4)
tech-RL-caida	40143 (**40152.5**)	**40142** (40152.9)	1 (−0.4)
web-wikipedia2009	**346675** (346687.9)	346676 (**346682.4**)	−1 (5.5)
web-wikipedia-growth	116817 (116818.8)	116817 (**116818.5**)	0 (0.3)

3. ScBppw outperforms the others on 32 instances.

This indicates that the SC strategy is more effective than the CC^2 and CC^2V3 strategies. For the sake of space, we omit the detailed results here. Overall, the SC strategy plays an essential role in our solver.

Now we analyzes the performances. (1) Apart from the tabu strategy, we find that there are two powerful diversification strategies in FastWMDS: a frequency based scoring function and a best-from-multiple-selection heuristic with some random walks. In contrast, besides the tabu strategy, ScBppw only exploits two weak diversification strategies: (i) best-picking with probabilistic walk; (ii) randomly selecting an uncovered vertex. Since (i) and (ii) are weak diversification strategies, an effective tabu strategy in our solver should be stronger than the CC^{2}V3 strategy in FastWMDS. Indeed, NVCC and SC greatly outperform CC^2 and CC^{2}V3 in our solver. (2) When a vertex v is added or removed, SC can release some vertices whose distance from v is 2. However, NVCC tends to release more vertices whose distance from v is 1. Thus SC is more likely to search in a bigger area while NVCC is more cautious in current areas. Therefore their performances are quite different.

Effectiveness of Best-Picking with Probabilistic Walk. We modify this strategy and develop two variants, ScBprw (Score Checking and Best-picking with Random Walk) and ScBppw-nlsp (ScBppw with no $lastStepImproved$) as follows. To develop ScBprw, we replace the probability selection in ScBppw with random walk, which means every vertex in D has equal probability to be removed. On the other hand, to develop ScBppw-nlsp, we remove the *lastStepImproved* predicate from ScBppw so that our algorithm will mark and check the hash table regardless of the *lastStepImproved* value. More specifically, we remove lines 1, 2, 8, 9, and 17 to 21, so as to develop ScBppw-nlsp. Then we compare these two variants with ScBppw. Experimental results show that

1. ScBprw outperforms the other solvers on 7 instances;
2. ScBppw-nlsp outperforms the others on 7 instances;
3. ScBppw outperforms the others on 21 instances.

Moreover, we remove the probabilistic selection component and develop a variant called ScDS, which will always do best-picking in the removing stage. Experimental results show ScBppw significantly outperforms ScDS as well.

Overall, we conclude that the probabilistic selection as well as the *lastStepImproved* predicate play an important role in ScBppw. For the sake of space, we omit the detailed results here.

7 Conclusions

In this paper, we have developed a local search MinDS solver named ScBppw. Experimental results show that our solver outperforms state-of-the-art on large graphs. In particular, it makes a substantial improvement on those graphs that contain more than 10^7 vertices. The main contributions include: (1) a tabu strategy based on score change; (2) a best-picking with probabilistic walk strategy; (3) a data structure to accelerate best-picking.

For future works, we will develop efficient heuristics to solve the MinDS problem on massive graphs which are too large to be stored in the main memory and has to be accessed through disk IOs.

References

1. Abseher, M., Dusberger, F., Musliu, N., Woltran, S.: Improving the efficiency of dynamic programming on tree decompositions via machine learning. In: IJCAI 2015, pp. 275–282 (2015)
2. Alber, J., Fellows, M.R., Niedermeier, R.: Efficient data reduction for Dominating Set: a linear problem kernel for the planar case. In: Penttonen, M., Schmidt, E.M. (eds.) SWAT 2002. LNCS, vol. 2368, pp. 150–159. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45471-3_16
3. Barabasi, A.L., Albert, R.: Emergence of scaling in random networks. Science **286**(5439), 509–512 (1999)
4. Cai, S., Su, K., Luo, C., Sattar, A.: NuMVC: an efficient local search algorithm for minimum vertex cover. J. Artif. Intell. Res. (JAIR) **46**, 687–716 (2013)
5. Cai, S., Su, K., Sattar, A.: Local search with edge weighting and configuration checking heuristics for minimum vertex cover. Artif. Intell. **175**(9–10), 1672–1696 (2011)
6. Campan, A., Truta, T.M., Beckerich, M.: Fast dominating set algorithms for social networks. In: Proceedings of the 26th Modern AI and Cognitive Science Conference 2015, pp. 55–62 (2015)
7. Chalupa, D.: An order-based algorithm for minimum dominating set with application in graph mining. Inf. Sci. **426**, 101–116 (2018)
8. Chung, F.R., Lu, L.: Complex graphs and networks, vol. 107 (2006)
9. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009)
10. Eubank, S., Kumar, V.S.A., Marathe, M.V., Srinivasan, A., Wang, N.: Structural and algorithmic aspects of massive social networks. In: Proceedings of SODA 2004, pp. 718–727 (2004)
11. Fan, Y., Li, C., Ma, Z., Wen, L., Sattar, A., Su, K.: Local search for maximum vertex weight clique on large sparse graphs with efficient data structures. In: Kang, B.H., Bai, Q. (eds.) AI 2016. LNCS (LNAI), vol. 9992, pp. 255–267. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50127-7_21
12. Fan, Y., Li, N., Li, C., Ma, Z., Latecki, L.J., Su, K.: Restart and random walk in local search for maximum vertex weight cliques with evaluations in clustering aggregation. In: IJCAI 2017, Melbourne, Australia, 19–25 August 2017, pp. 622–630 (2017)
13. Haynes, T.W., Hedetniemi, S.M., Hedetniemi, S.T., Henning, M.A.: Domination in graphs applied to electric power networks. SIAM J. Discrete Math. **15**(4), 519–529 (2002)
14. Hedar, A.R., Ismail, R.: Simulated annealing with stochastic local search for minimum dominating set problem. Int. J. Mach. Learn. Cybern. **3**(2), 97–109 (2012)
15. Hoos, H.H., Stützle, T.: Stochastic local search. In: Handbook of Approximation Algorithms and Metaheuristics (2007)
16. Jiang, H., Li, C., Manyà, F.: An exact algorithm for the maximum weight clique problem in large graphs. In: AAAI, California, USA, pp. 830–838 (2017)
17. Li, C.M., Huang, W.Q.: Diversification and determinism in local search for satisfiability. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 158–172. Springer, Heidelberg (2005). https://doi.org/10.1007/11499107_12
18. Ma, Z., Fan, Y., Su, K., Li, C., Sattar, A.: Local search with noisy strategy for minimum vertex cover in massive graphs. In: Booth, R., Zhang, M.-L. (eds.) PRICAI 2016. LNCS (LNAI), vol. 9810, pp. 283–294. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-42911-3_24

19. Murray, R.S.: Theory and Problems of Probability and Statistics. McGraws-Hill Book and Co., Singapore (1975)
20. Nacher, J.C., Akutsu, T.: Minimum dominating set-based methods for analyzing biological networks. Methods **102**, 57–63 (2016)
21. Potluri, A., Bhagvati, C.: Novel morphological algorithms for dominating sets on graphs with applications to image analysis. In: Barneva, R.P., Brimkov, V.E., Aggarwal, J.K. (eds.) IWCIA 2012. LNCS, vol. 7655, pp. 249–262. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34732-0_19
22. Samuel, H., Zhuang, W., Preiss, B.: DTN based dominating set routing for manet in heterogeneous wireless networking. Mob. Netw. Appl. **14**(2), 154–164 (2009)
23. Shen, C., Li, T.: Multi-document summarization via the minimum dominating set. In: COLING 2010, Beijing, China, pp. 984–992 (2010)
24. Wang, Y., Cai, S., Chen, J., Yin, M.: A fast local search algorithm for minimum weight dominating set problem on massive graphs. In: IJCAI 2018, pp. 1514–1522 (2018)
25. Wang, Y., Cai, S., Yin, M.: Local search for minimum weight dominating set with two-level configuration checking and frequency based scoring function. J. Artif. Intell. Res. **58**, 267–295 (2017)
26. Yao, B., Fei-Fei, L.: Action recognition with exemplar based 2.5D graph matching. In: Fitzgibbon, A., Lazebnik, S., Perona, P., Sato, Y., Schmid, C. (eds.) ECCV 2012. LNCS, vol. 7575, pp. 173–186. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33765-9_13