Improving Local Search for Minimum Weighted Connected Dominating Set Problem by Inner-Layer Local Search

Bohan Li
State Key Laboratory of Computer Science Institute of Software,
Chinese Academy of Sciences, Beijing, China
School of Computer Science and Technology,
University of Chinese Academy of Sciences, Beijing, China

Kai Wang
School of Computer Science and Information Technology,
Northeast Normal University, Changchun, China

Yiyuan Wang
School of Computer Science and Information Technology,
Northeast Normal University, Changchun, China
Key Laboratory of Applied Statistics of MOE,
Northeast Normal University, Chnagchun, China

Shaowei Cai[1]
State Key Laboratory of Computer Science Institute of Software,
Chinese Academy of Sciences, Beijing, China
School of Computer Science and Technology,
University of Chinese Academy of Sciences, Beijing, China

Abstract

The minimum weighted connected dominating set (MWCDS) problem is an important variant of connected dominating set problems with wide applications, especially in heterogenous networks and gene regulatory networks. In the paper, we develop a nested local search algorithm called NestedLS for solving MWCDS on classic benchmarks and massive graphs. In this local search framework, we propose two novel ideas to make it effective by utilizing previous search information. First, we design the restart based smoothing mechanism as a diversification method to escape from local optimal. Second, we propose a novel inner-layer local search method to enlarge the candidate removal set, which can be modelled as an optimized version of spanning tree problem. Moreover, inner-layer local search method is a general method for maintaining the connectivity constraint when dealing with massive graphs. Experimental results show that NestedLS outperforms state-of-the-art meta-heuristic algorithms on most instances.

2012 ACM Subject Classification Theory of computation → Randomized local search; Applied computing → Operations research

Keywords and phrases Operations Research, NP-hard Problem, Local Search, Weighted Connected Dominating Set Problem

Digital Object Identifier 10.4230/LIPIcs.CP.2021.39

Supplementary Material *Software (Source Code)*: `https://github.com/DouglasLee001/NestedLS`

Funding This work is partially supported by National Key R&D Program of China (2019AAA0105200), Beijing Academy of Artificial Intelligence (BAAI), NSFC Grant 61806050, and Jilin Science and Technology Association QT202005.

[1] corresponding author

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).
Editor: Laurent D. Michel; Article No. 39; pp. 39:1–39:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Given an undirected connected graph $G = (V, E)$, a set $D \subseteq V$ forming a connected subgraph in G is called a connected dominating set (CDS) if each vertex in V either belongs to D or is adjacent to at least one vertex in D. The minimum connected dominating set (MCDS) problem is to find a CDS with the minimum size. MCDS is a well-known combinatorial optimization problem with important applications [1, 14].

MCDS assumes that vertices are equally important. However, this assumption fails to hold in many real world scenarios where each vertex is associated with various types of weights. A specific application is to model heterogenous networks [22] where each vertex generates different cost (e.g., energy consumption and communication delay). The paradigm of handling such vertex weighted graph refers to an important generalization of MCDS, i.e., minimum weighted connected dominating set (MWCDS) problem, aiming to find a CDS with the minimum total weight. The MWCDS is used to form a low-cost network backbone for communication applications where the cost usually represents the power consumption rate or corresponding security coefficient of backbones [27, 28]. Moreover, MWCDS has other applications in biological networks [16] and generating pictorial storylines [23]

1.1 Related Work

MWCDS is a classic NP-hard problem, meaning that there are no polynomial-time algorithms for the MWCDS problem, unless NP=P. Although MCDS is widely studied and many specialized algorithms have already been proposed to solve MCDS on graphs with different sizes, these MCDS algorithms [9, 18, 11, 13] cannot be directly used to deal with the MWCDS problem because they fail to consider the weight information and structure characteristics.

Because of its NP-hardness, much of the research effort in the past decade has focused on obtaining a good MWCDS solution within a reasonable time. In the literature, two types of algorithms are mainly distinguished for MWCDS, i.e., approximation algorithms and meta-heuristic algorithms. The approximation algorithms can find approximate solutions with provable guaranteed approximation ratio, but they usually have poor performance in practice, especially in massive graphs. Representative approximation algorithms for MWCDS mainly used centralized methods [2, 29] or distributed methods [5, 21]. According to the literature, the current best meta-heuristic algorithm for MWCDS is ACO-RVNS [3] based on ant colony optimization and reduced variable neighborhood search.

1.2 Our Contributions

Previous MWCDS algorithms performed well for classic benchmarks, but they had poor performance on massive graphs. In this paper, to further improve the performance of MWCDS on both classic and massive graphs, we propose a nested local search framework called NestedLS, including three phases, i.e., vertices swapping phase, tree reconstruction phase and solution restart phase. Based on the framework, we design two novel ideas by utilizing previous search information.

First, we propose the restart based smoothing mechanism (*ReSmooth*), which can be viewed as a diversification method. In order to escape from a local optima, *ReSmooth* restarts the algorithm by reconstructing a new solution during the solution restart phase. During the reconstruction process, two kinds of previous search information (w.r.t, non-dominated information and best solution information) are inherited to guide the algorithm to the promising search space, resulting in a new inheriting scoring function, denoted as

$score_{inher}$. Moreover, after a few restart operations, the initial solution may converge. To address this, we propose a smoothing mechanism based on the repeating rate of solution to further diversify the search spaces.

The second and more important idea is the inner-layer local search method ($InnerSearch$). Although an efficient tree-based connectivity maintenance method (TBC) proposed by Li [13] used the spanning tree to maintain the candidate removal set, it cannot utilize search information when constructing the spanning tree. In order to enlarge the candidate removal set, the $InnerSearch$ is applied to reconstruct the spanning tree by modelling it as a weighted max-leaf spanning tree problem (WMST). Meanwhile, based on three novel intuitions of WMST, the corresponding vertex selection rule is proposed to guide the $InnerSearch$ to construct the spanning tree and further improve it by a local search procedure.

These proposed ideas can be generally applied to other heuristic algorithms. Specifically, $InnerSearch$ is a general method for maintaining the connectivity constraint when dealing with massive graphs, and *ReSmooth* provides a novel diversification scheme for restart-based heuristic algorithms.

Extensive experiments are carried out to evaluate NestedLS on classic benchmarks and massive graphs. Experimental results indicate that NestedLS outperforms other state-of-the-art MWCDS heuristic algorithms on most instances, and confirm the effectiveness of two novel ideas.

2 Preliminaries

Let $G = (V, E, w)$ be a weighted graph where V is the set of vertices, E is the set of edges and each vertex $v \in V$ is associated with a positive weight $w(v)$. For a vertex v, its neighborhood is $N_G(v) = \{u \in V | \{u, v\} \in E\}$, and its closed neighborhood is $N_G[v] = N_G(v) \cup \{v\}$. The degree of a vertex v, denoted as $d_G(v)$, is defined as $|N_G(v)|$, and Δ_G is the maximum number of $d_G(v)$ for $\forall v \in V$. Given a vertex set $S \subseteq V$, $N_G(S) = \bigcup_{v \in S} N_G(v) \setminus S$ and $N_G[S] = \bigcup_{v \in S} N_G[v]$ stands for the neighborhood and closed neighborhood of S, respectively. $G[S] = (V_S, E_S)$ is a subgraph in G induced by S such that $V_S = S$ and E_S consists of all the edges in E whose endpoints are in S. A weighted graph G is connected when it has at least one vertex and there is a path between every pair of vertices.

▶ **Definition 1.** *Given a weighted connected graph G, a vertex in G is an articulation vertex iff removing it, together with the edges connected to it, disconnects the graph. The articulation vertex set of G is denoted as $art(G)$.*

Given a vertex set $D \subseteq V$, a vertex $v \in V$ is *dominated* by D if $v \in N_G[D]$, and is non-dominated otherwise. We use $D \subseteq V$ to denote a candidate solution and the weight of D is $w(D) = \sum_{v \in D} w(v)$. $unDom_G(D) = V \setminus N_G[D]$ denotes a subset of vertices in G non-dominated by D. If $G[D]$ is connected and D dominates all vertices in V, D is a connected dominating set (CDS). The minimum weighted connected dominating set problem (MWCDS) is to find a CDS with the minimum total weight.

2.1 Review of Scoring Function for MWCDS

The frequency based scoring function $score_f$ is recently proposed by Wang et al. [26]. Each vertex $v \in V$ has a property: frequency, denoted as $freq[v]$. It works as follows: 1) at first, $freq[v]$=1 for $\forall v \in V$; 2) at the end of each iteration of local search, $freq[v]$=$freq[v] + 1$ for each non-dominated vertex. If $u \in D$, $score_f(u)$=$-\sum_{v \in C_1(u)} freq[v]/w(u)$, and otherwise $score_f(u)$=$\sum_{v \in C_2(u)} freq[v]/w(u)$, where $C_1(u)$ is the set of dominated vertices that would become non-dominated by removing u from D and $C_2(u)$ is the set of non-dominated vertices that would become dominated by adding u to D. Moreover, considering that *age* [2] is usually used to break ties for diversification, the selection rule is described as follows.

Selection Rule: Select the added or removed vertex with the greatest $score_f$, breaking ties by preferring the one with the greatest *age*.

2.2 Review of TBC

For combinatorial optimization problem with connectivity constraint, a key factor to the performance is the connectivity maintenance methods, especially for massive graphs. To tackle it, a tree-based connectivity maintenance called TBC method was proposed [13], inspired by spanning trees. Given a candidate solution D, a spanning tree T of $G[D]$ and its corresponding leaf set $LS(T)$ are maintained during the search process. Each vertex $v \in LS(T)$ is allowed to be removed from D, while all other vertices are forbidden to be removed. Details for TBC can refer to [13].

3 The NestedLS Algorithm

In this section, we propose an algorithm for solving MWCDS called NestedLS.

The pseudo code of NestedLS is presented in Algorithm 1. On a top level, NestedLS works as follows. After the initialization, a loop (lines 3–18) is executed until a given time limit is reached, and the best solution is finally returned (line 19). Each iteration of the loop consists of three phases, namely vertices swapping phase, tree reconstruction phase and solution restart phase. At the first phase (lines 4–12), the candidate solution is updated by swapping vertices. In the second phase (lines 13–14), the spanning tree is periodically updated for diversification. During the third phase (lines 15–18), the candidate solution and corresponding spanning tree are rebuilt if the algorithm falls into the local optima.

Before detailed description, we first introduce some notations and definitions. In NestedLS, $NoImproveStep$ denotes the number of consecutive iterations without improvement. $MaxNoImprove$ and $TreeNoImprove$ denote the parameters for reconstructing the solution and the spanning tree respectively. D, D^* and D_{last} denote the current candidate solution, the best solution and the previous solution after last construction, respectively. During the search process, two candidate selection subsets are maintained as follows.

(1) The candidate subset for addition is defined as $candAdd(D) = N_G(D) \cap N_G(unDom_G(D))$, where $N_G(D)$ contains vertices maintaining connectivity and $N_G(unDom_G(D))$ is adjacent to the non-dominated vertex set. To avoid visiting previous candidate solutions, we use the CC^2 strategy [26] to further restrain $candAdd(D)$.

(2) The candidate subset for removal is denoted as $candRem(D)$. If $|D| < \kappa$, $candRem(D) = D \setminus art(G[D])$ where $art(G[D])$ is calculated by Tarjan's algorithm [10]. Otherwise, TBC is adopted and $candRem(D) = LS(T)$. To overcome the cycling problem, the tabu method [8] is applied to exclude those just added vertices from $candRem(D)$ for the next tt iterations. In our work, $tt = 5 + rand(10)$ and $\kappa = 100$.

Now we describe the NestedLS algorithm in detail.

[2] The *age* of a vertex v is the number of steps that have occurred since v last changed its state.

Algorithm 1 The NestedLS algorithm.

```
   Input: A weighted graph G = (V, E, w), the cutoff time
   Output: The best obtained solution D*
1  D := D* := D_last := ReSmooth(∅, V);
2  T := InnerSearch(G[D]) and NoImproveStep := 1;
3  while timeElapse < cutoff do
4      for i := 1 to neighborSize do
5          choose vertex u ∈ candRem(D) using selection rule and D := D \ {u};
6      while |unDom_G(D)| ≠ 0 and w(D) < w(D*) do
7          choose vertex v ∈ candAdd(D) using selection rule and D := D ∪ {v};
8      if D is a feasible solution then
9          D* := D and NoImproveStep := 1;
10     else
11         freq[v] := freq[v] + 1, for ∀v ∈ unDom_G(D);
12         NoImproveStep := NoImproveStep + 1;
13     if NoImproveStep%TreeNoImprove == 0 then
14         T := InnerSearch(G[D]);
15     if NoImproveStep > MaxNoImprove then
16         NoImproveStep := 1;
17         D := D_last := ReSmooth(D_last, D*);
18         T := InnerSearch(G[D]);
19 return D*;
```

In the beginning, D, D^* and D_{last} are initialized by the *ReSmooth* procedure (line 1) which will be discussed in Section 4. The corresponding spanning tree T is built by a novel inner-layer local search, which will be introduced in Section 5, and $NoImproveStep$ is set to 1 (line 2).

In the vertices swapping phase, $neighborSize$ vertices are first chosen from $candRem(D)$ using the selection rule. Then, vertices $v \in candAdd(D)$ are added via the selection rule, until there are no non-dominated vertices or $w(D) \geq w(D^*)$. During this process, the total weight of current candidate solution stays below the best value.

Thus, after swapping vertices, if a feasible solution is obtained, indicating that a better solution is found, then D^* and $NoImproveStep$ are updated (line 9). Otherwise, the corresponding $freq$ values and $NoImproveStep$ are increased by one (lines 10–12).

In the tree reconstruction phase, if the condition is satisfied (line 13), then T will be reconstructed accordingly (line 14).

In the solution restart phase, when $NoImproveStep$ exceeds $MaxNoImprove$, meaning that the algorithm falls into the local optima, $NoImproveStep$ is reset and the candidate solution D and D_{last} are reconstructed (lines 16–17). Then, the spanning tree T is rebuilt accordingly (line 18).

4 Restart Based Smoothing Mechanism

In the solution restart phase of NestedLS, an important component is called restart based smoothing mechanism (*ReSmooth*), which restarts the algorithm by constructing a new solution when falling into the local optima.

4.1 Inheriting Scoring Function

In the solution restart phase, starting from an empty candidate set, vertices are iteratively added to the candidate solution by some strategy, until its weight exceeds the best solution or all vertices are dominated. During this phase, if a pure random procedure is applied to generate an initial solution, the initial solution will fail to inherit previous search information. This may make the algorithm deviate from the promising search space and thus degrade the convergence rate of local search.

To hand this issue, two kinds of search information need to be considered.

The first information is the accumulated non-dominated information, represented by $score_f$. The second essential information is "the high-quality solution", from which the vertices should be selected with higher priority than others. To make full use of the two kinds of search information above, we define a novel scoring function called inheriting scoring function, denoted as $score_{inher}$ as follows.

$$score_{inher}(v) = \begin{cases} score_f(v) \times \beta, v \notin D^* \cup D_{last\_best} \\ score_f(v), \qquad v \in D^* \cup D_{last\_best} \end{cases}$$

In the above equation, "the high-quality solution" refers to D^* and D_{last_best} which denotes the solution dominating most vertices since last solution restart phase. If all vertices are dominated by D_{last_best}, then D_{last_best} is equal to D^*. Parameter β denotes the penalty coefficient. Based on this scoring function, we propose the novel selection rule.

Inheriting-Based Selection Rule: Choose the vertex with the greatest $score_{inher}$ value, breaking ties randomly.

4.2 Smoothing Mechanism

We observe that the initial candidate solution may converge after several solution restart phases. The main reason is that $freq$ values of some vertices accumulate to a large amount, leading the algorithm to follow the previous search trajectory and then explore some recently visited search spaces.

To avoid such phenomenon, $freq$ should be smoothed when the initial solutions converge. Thus, NestedLS employs a weight smoothing scheme which resembles SWT [4] in some respect. First, we introduce the Jaccard index [12] to illustrate the repeating rate of solutions.

▶ **Definition 2.** *The repeating rate between the initial solution of last restart D_{last} and D is defined by the Jaccard index: $J(D, D_{last}) = |D \cap D_{last}|/|D \cup D_{last}|$.*

When $J(D, D_{last})$ exceeds a threshold $MaxRepeat$, indicating that the initial solutions converge, the $freq$ values of all vertices are smoothed as follows.

$$freq[v] = \rho \cdot freq[v] + (1 - \rho) \cdot \overline{freq}, \quad \forall v \in V$$

where $\overline{freq}$ is the average value of $freq$ and ρ is the smoothing parameter. After smoothing all $freq$ values, score values will be updated accordingly. Experiments on classic benchmark show that the average repeating rate without smoothing is on average 0.69 after calling the *ReSmooth* 100 times, which confirms that without the smoothing method, the initial candidate solution may converge.

Algorithm 2 *ReSmooth*(D_{last},D^*).

Input: The solution after last construction D_{last}, D^*
Output: A restart candidate solution D
1 choose a random node $v \in V$ and $D := \{v\}$;
2 **while** $|unDom_G(D)| \neq 0$ *and* $w(D) < w(D^*)$ **do**
3   choose $v \in candAdd(D)$ based on **Inheriting-based Selection Rule** and $D := D \cup \{v\}$;
4 **if** $J(D, D_{last}) > MaxRepeat$ **then**
5     $freq[v] = \rho \cdot freq[v] + (1 - \rho) \cdot \overline{freq}$, for $\forall v \in V$;
6 **return** D;

4.3 The *ReSmooth* Algorithm

The *ReSmooth* is described in Algorithm 2. A random vertex is first added into the empty candidate solution (line 1). Then, vertices are chosen to the candidate solution D based on the inheriting-based selection rule, until all vertices are dominated, or the weight of D exceeds that of the best solution ever (lines 2–3). The $freq$ values are smoothed if the repeating rate exceeds the threshold $MaxRepeat$ (lines 4–5). Finally, the restart candidate solution D is returned (line 6).

5 Inner-layer local search

In the tree reconstruction and solution restart phases when handling massive graphs, in order to enlarge *candRem*, an important component called inner-layer local search $InnerSearch$ is proposed to rebuild a corresponding spanning tree. Also, it can be modelled as a weighted max-leaf spanning tree problem, which is an interesting version of classic spanning tree problem [7].

For current solution D, $G[D]$=(V_D, E_D) and T denote its subgraph and corresponding spanning tree. $LS(T)$ is the leaf set of T, which serves as $candRem(D)$, while $TS(T) = D \setminus LS(T)$ denotes the trunk set where vertices are forbidden to be removed during the vertices swapping phase.

5.1 Motivation for Inner-layer Local Search

Before constructing a new spanning tree, we first formally define the weighted max-leaf spanning tree problem (WMST).

▶ **Definition 3.** *Given a graph $G = (V, E, w)$, the weighted max-leaf spanning tree problem is to find a spanning tree of G with the maximum total weight of leaf set, that is, the minimum total weight of trunk set.*

For any spanning tree T of solution D, its trunk set $TS(T)$ is connected and connects to all leaf vertices in $LS(T)$. Thus, WMST can be converted to find a MWCDS of $G[D]$, serving as the trunk set $TS(T)$. We propose an $InnerSearch$ method to construct a CDS as $TS(T)$, and then further improve its quality by the local search procedure. To define the scoring function for obtaining $TS(T)$, we propose three intuitions whose importance is displayed in descending order.

(1) The first intuition is that there should be more candidate removal vertices to enlarge the search space. Moreover, vertices with large weight value should be more likely to be removed to lower $w(D)$. During the vertices swapping phase, $CandRem(D) = LS(T)$ when solving massive graphs. In order to implement the above intuition, there should be more leaf vertices, and vertices with large weight values should be maintained in $LS(T)$. Specifically, we employ a simplified version of $score_f$ as the main scoring function of solving WMST, denoted as $score^{'}_f$ with respect to $TS(T)$. Given a graph $G[D]$, if $u \in TS(T)$, $score^{'}_f(u) = -|C_1(u)|/w(u)$ and otherwise $score^{'}_f(u) = |C_2(u)|/w(u)$, where $C_1(u)$ and $C_2(u)$ have already been defined in Section 2.2.

(2) Our second intuition is that vertices which intensively degrade the quality of D if deleted, should be forbidden to be removed, and thus they should be excluded from leaf set. To achieve it, the direct way is that the $score_f$ of leaf vertices should be higher, while the $score_f$ of trunk vertices should be lower. This means that vertices with lower $score_f$ are preferred to be left in $TS(T)$.

(3) The third intuition is that the leaf set should differ from previous ones, so that the algorithm can have more different removing options. To achieve this, vertices with higher exchanging frequency of operations (i.e., to be moved during the vertices swapping), denoted as $score_e$, are preferred to be left in the trunk set. Since those vertices are frequently set as leaf vertices since last construction, leaving them in the trunk set can make the leaf set differ from the previous one.

It is important to notice that during the $InnerSearch$ procedure, the $score^{'}_f$ values will be dynamically updated, while the corresponding $score_f$ values keep unchanged because the corresponding $score_f$ is based on D that remains unchanged in this procedure. For $v \in D$, $score_f(v)$ is always no larger than 0. Based on these three intuitions, we propose the novel selection rule for constructing $TS(T)$ as follows.

WMST Selection Rule: Select an added (or removed) vertex with the greatest $score^{'}_f$, breaking ties by picking one with the highest (or lowest) $|score_f|$ value. Further ties are broken by choosing one with the highest (or lowest) $score_e$.

5.2 The $InnerSearch$ Algorithm

The pseudo code of *InnerSearch* is shown in Algorithm 3. The algorithm first constructs a CDS of $G[D]$ called D', serving as the trunk set of $G[D]$, by greedily adding vertices until it becomes a feasible solution (lines 1–3), similar to the *ReSmooth* procedure, and then the spanning tree T' of D' is built by breadth first search (line 5). The loop iterates until it fails to find a better solution within $MaxNoImproveInner$ steps (line 6). During each loop, local search is applied by iteratively swapping vertices based on the WMST selection rule to improve D' (lines 7–10). At the end of each loop, the corresponding spanning tree T' needs to be updated (line 11). After the loop, the spanning tree T of D is constructed by adding the remaining vertices in $D \setminus D'$ to T' by using the adding rule of TBC method [13] (line 15). At last, the new spanning tree T is returned (line 16).

Note that to lower the complexity, the best solution during $InnerSearch$ is not recorded, and an approximated best solution D' is obtained by setting $MaxNoImproveInner$ to a small value. In $InnerSearch$, the complexity of each iteration (lines 6–14) is $O(neighborSize * \Delta_{G[D]})$, while the complexity of remaining parts is $O(|V_D| * \Delta_{G[D]} + |E_D|)$. Since D only accounts for 13.07% of vertices of the original graph on average, $InnerSearch$ can be seen as a lightweight local search procedure, compared to Algorithm 1.

Algorithm 3 $InnerSearch(G[D])$.

Input: a subgraph $G[D]$ induced by candidate solution D
Output: a spanning tree T of $G[D]$

choose a random vertex $v \in G[D]$ and $D' := \{v\}$;
while $|unDom_{G[D]}(D')| \neq 0$ **do**
  choose vertex $v \in N_{G[D]}(D')$ using **WMST selection rule** and $D' := D' \cup \{v\}$;
$MinWeight := w(D')$ and $InnerStep := 1$;
construct a spanning tree T' of D';
while $InnerStep < MaxNoImproveInner$ **do**
  for $i := 1$ *to* $neighborSize$ **do**
    choose vertex $u \in LS(T')$ using **WMST selection rule** and $D' := D' \setminus \{u\}$;
  while $|unDom_{G[D]}(D')| \neq 0$ **do**
    choose vertex $v \in candAdd(D')$ using **WMST selection rule** and $D' := D' \cup \{v\}$;
  update the spanning tree T' based on D';
  if $w(D') < MinWeight$ **then**
    $MinWeight := w(D')$ and $InnerStep := 1$;
  else $InnerStep := InnerStep + 1$;
construct T where $TS(T) = T'$ and $LS(T) = D \setminus D'$;
return T;

6 Experimental Results

6.1 Experiment Preliminaries

Extensive experiments are carried out to evaluate the performance of NestedLS, compared with four state-of-the-art heuristic algorithms, including HGA [6], PBIG [6], ACO-RVNS [3] and ACO-e, which was modified by the author of ACO-RVNS, specialized for massive graphs. Since the source or binary codes of HGA and PBIG were not available, we reimplemented and then compared to them. The source code of ACO-RVNS and ACO-e were kindly provided by authors. The data structure of all competitors was modified for massive graphs. Specifically, the adjacency list are applied to store the graph information. NestedLS and its competitors were implemented in C++ and compiled by g++ with '-O3'. All experiments were run on a server with Intel Xeon CPU E7-8850 v2 2.30GHz with 2048GB RAM under Ubuntu 16.04.5. All algorithms were executed 10 times with random seeds from 1 to 10 on each instance independently. The cutoff time was set to 1000 seconds for the classic benchmarks, and 5000 seconds for massive graphs. We report the best size (*min*) and average size (*avg*) of the solution found by each algorithm. The bold values indicate the best solution among all the algorithms.

The parameters of NestedLS are tuned by irace [15]. We select 40 graphs randomly from all benchmarks, and irace was applied for 5000 s with a budget of 10000 applications. The chosen values of parameters are presented in Table 1. Moreover, the parameters of all competitors are also tuned by irace, and our re-implementation versions can obtain similar performance as the original papers, which confirms their effectiveness and efficiency.

We evaluate NestedLS on 5 benchmarks, including 2 classic benchmarks in the literature and 3 massive benchmarks.

Table 1 Parameter tuning.

Parameter	Domain	Chosen value
$neighborSize$	{1,3,5}	3
$MaxNoImprove$	{10000,50000,100000}	100000
$MaxNoImproveInner$	{1000,5000}	1000
$TreeNoImprove$	{5000,10000}	10000
$MaxRepeat$	{0.1,0.3,0.5}	0.3
ρ	{0.3,0.7}	0.7
β	{0.5,0.7,0.9}	0.7

Table 2 Experiment results on the first classic benchmark. The averaged value of min ($\overline{min}$) and the number of connected instances with the same size ($\#inst$) are reported for each family.

Instance Family	$\#inst$	NestedLS $\overline{min}$	PBIG $\overline{min}$	ACO-e $\overline{min}$	ACO-RVNS $\overline{min}$	HGA $\overline{min}$	Instance Family	$\#inst$	NestedLS $\overline{min}$	PBIG $\overline{min}$	ACO-e $\overline{min}$	ACO-RVNS $\overline{min}$	HGA $\overline{min}$
TYPEI							V800E10000	1	**2059**	2080	2111	2076	2442
V250E750	7	**2833**	2850.3	2836.4	**2833**	3068.1	V1000E5000	1	**6538**	6762	6652	6668	7281
V250E1000	9	**2038**	2056.8	2039.1	**2038**	2227.8	V1000E10000	1	**2989**	3013	3052	3029	3531
V250E2000	10	**965.9**	974	968.7	**965.9**	1090.3	V1000E15000	1	**2164**	2178	2189	2189	2434
V250E3000	10	**650.4**	653.1	653	**650.4**	744.3	V1000E20000	1	**1612**	1639	1645	1616	1800
V250E5000	10	**390.2**	392.3	391.5	390.9	433.9	**TYPEII**						
V300E750	2	**4272.5**	4242.4	4283.5	**4272.5**	4449.5	V250E750	6	877.5	896.5	877.5	**876.8**	924.7
V300E1000	9	**3067.9**	3111	3076.2	3068.2	3315.4	V250E1000	9	**953.7**	956.2	958	953.9	1014.6
V300E2000	10	**1439.4**	1457.5	1444.7	**1439.4**	1639.4	V250E2000	10	**1159.9**	1161.7	1163.6	**1159.9**	1272.9
V300E3000	10	**936.1**	942.2	939.5	936.3	1066.4	V250E5000	10	**1469.8**	1471.9	1471.8	**1469.8**	1601.5
V300E5000	10	**555.1**	561.1	557.6	556.9	634.9	V300E750	1	**974**	981	979	**974**	999
V500E2000	1	**4179**	4239	4183	4182	4579	V300E1000	9	**1037.7**	1054.6	1040.6	**1037.7**	1092.6
V500E5000	1	**1565**	1571	1580	**1565**	1748	V300E2000	10	**1276.3**	1287.6	1279.4	1276.4	1395.6
V500E10000	1	**852**	**852**	868	**852**	922	V300E5000	10	**1612.9**	1618.9	1613	**1612.9**	1882.5
V800E5000	1	**4178**	4321	4223	4205	4740							

The first classic benchmark originally from [19] is classified into Type I (96 instances) and Type II (65 instances). There are a few unconnected graphs in the benchmark, and we choose to ignore them. The second classic benchmark (20 instances) is originally generated in [6]. To save space, we do not report the results on graphs with less than 250 vertices where NestedLS always performs best. In total, we selected 181 classic instances.

A total of 118 massive real-world graphs are selected from the Network Data Repository (NDR) [17] and Stanford Large Network Dataset Collection (SNAP)[3], as well as large instances from the 10th DIMACS implementation challenge (DIMACS10)[4]. Due to space limitations, we only report results on graphs from the SNAP and DIMACS10 benchmarks with at least 30,000 vertices and graphs from the NDR benchmark with more than 100,000 vertices and more than 1,000,000 edges. Hence, we picked 22, 31 and 65 graphs in SNAP, DIMACS10, and NDR, respectively. To obtain the corresponding weighted instances, we used the same method as in previous works [24, 25]: for the ith vertex v_i, $w(v_i)=(i \bmod 200)+1$.

[3] `http://snap.stanford.edu/data`
[4] `https://www.cc.gatech.edu/dimacs10/`

Table 3 Experiment results on the second classic benchmark.

Instance	NestedLS *min*(*avg*)	PBIG *min*	ACO-e *min*(*avg*)	ACO-RVNS *min*(*avg*)	HGA *min*
V250E500	**4464(4464)**	4585	**4464**(4479.6)	4464(4469.2)	4716.1
V250E1000	**2203.5(2203.5)**	2228	2227.4(2227.5)	2211.8(2213.1)	2389
V250E1500	**1365.7(1365.7)**	1384	**1365.7(1365.7)**	**1365.7(1365.7)**	1548.1
V250E2000	**1020.3(1020.3)**	1044.9	**1020.3**(1026.7)	**1020.3(1020.3)**	1104.9
V250E2500	**822.1(822.1)**	**822.1**	**822.1(822.1)**	**822.1(822.1)**	960.3
V500E1000	**8636.7(8637.1)**	8837.3	8646.69(8679.2)	8637.2(8648)	9444.3
V500E2000	**4256(4256)**	4352	4296.1(4340)	4277.9(4294.4)	4693.7
V500E3000	**2867.2(2867.3)**	2915.8	2895.1(2927.8)	2875.7(2875.7)	3256.9
V500E4000	**2145.7(2145.7)**	2164.3	2157.1(2176.6)	2157.1(2170.2)	2434.5
V500E5000	**1531.6(1531.6)**	1538	1531.6(1541.8)	**1531.6(1531.6)**	1766.5
V750E1500	**13894.9(13903.7)**	14298.5	14042.2(14101.7)	13984.6(14031.4)	15491.2
V750E3000	**6106.7(6110.9)**	6250.9	6209.7(6252.7)	6154.6(6173.3)	6979.4
V750E4500	**4244.4(4244.4)**	4383.5	4330.6(4398.8)	4308.7(4328.1)	4674.7
V750E6000	**3151.7(3152.9)**	3188.9	3167.7(3180.1)	3163.1(3163.1)	3505.6
V750E7500	**2401.8(2402.5)**	2435	2451.2(2469)	2434.1(2434.7)	2744.8
V1000E2000	**17745.5(17768.1)**	18235.3	17838.3(17922.3)	17845(17889.3)	19786.5
V1000E4000	**8222.8(8222.8)**	8453.3	8328.6(8360.6)	8319.7(8335.8)	9532
V1000E6000	**5247.9(5250.4)**	5341.9	5332(5372.1)	5301.2(5319.2)	5938.7
V1000E8000	**3906.2(3910.7)**	3983.5	3955.5(4012.7)	3931.1(3956.5)	4465.1
V1000E10000	**3106.6(3108.7)**	3154.8	3187.2(3201.3)	3119.2(3150.9)	3683.4

6.2 Results on Classic Benchmarks

Results on classic benchmarks are reported in Tables 2 and 3. NestedLS is better than all competitors, except for V250E750, indicating its robustness. The average run time of NestedLS on some instances where it can generate the same solution quality (i.e., same minimal and average values) as PBIG, ACO-e and ACO-RVNS is 16.3 s, 27.3 s and 11.6 s, respectively, while that of competitors is 5.2 s, 87 s and 15 s.

6.3 Results on Massive Graphs

Note that ACO-RVNS and HGA fail to find a solution on most massive instances, mainly due to their high complexity heuristics (i.e., RVNS and Minimize functions). Thus, we mainly report the results of NestedLS, PBIG and ACO-e on Tables 4 and 5. NestedLS significantly outperforms all competitors on most instances, with only 8 exceptions. Moreover, NestedLS can solve all the 118 instances within the time limit, while PBIG, ACO-e, ACO-RVNS and HGA can only solve 103, 47, 19 and 13 instances, respectively. Among all the instances solvable by NestedLS and a corresponding competitor, the best solution obtained by NestedLS is on average 4.18%, 1.37%, 1.08% and 1.39% better than that found by PBIG, ACO-e, ACO-RVNS and HGA, respectively. Since the weight value can amount to 10^8 on some massive graphs, they are significant improvements.

Table 4 Experiment results on SNAP and DIMACS10 benchmarks. If an algorithm fails to find a solution within the cutoff time, it is indicated by "N/A".

Instance	NestedLS *min(avg)*	PBIG *min(avg)*	ACO-e *min(avg)*
Amazon0302	**3607951(3628251.9)**	3898308(3903172)	3702466(3702466)
Amazon0312	**4201432(4215293.4)**	4397464(4402398.5)	N/A
Amazon0505	**4383032(4393754.3)**	4576088(4579237)	N/A
Amazon0601	**3780727(3792534.5)**	3987798(3989201.8)	N/A
Cit-HepPh	**247185(247337.8)**	255517(255731)	253099(253493.8)
Cit-HepTh	**256033(256647.4)**	263235(263496.3)	260885(261364)
cit-Patents	**59497255(59657281.8)**	64161977(64167456)	N/A
Email-EuAll	**228890(228936.4)**	228935(228954)	228951(228975.5)
p2p-Gnutella04	**210746(210813.7)**	211570(211610.5)	211153(211304.3)
p2p-Gnutella25	451125(**451178.5)**	451333(451426.5)	**451056**(451208.4)
p2p-Gnutella30	711958(**712177.5)**	712094(712192.5)	**711915**(712066.5)
p2p-Gnutella31	1262834(1263059.3)	**1262095(1262181.3)**	1262676(1262869.1)
Slashdot0811	**1460128(1460346.5)**	1461470(1461546.8)	1461427(1461427)
Slashdot0902	**1580606(1580816.2)**	1583119(1583276.5)	1582395(1582395)
soc-Epinions1	**1663107(1663222.1)**	1663776(1663911.3)	1664085(1664085)
web-BerkStan	2936734(2939268.8)	2987292(2989303.8)	**2934995(2934995)**
web-Google	**7864164(7868993.6)**	7985154(7985584)	N/A
web-NotreDame	**2495507(2496448.9)**	2519655(2520284)	2497288(2497288)
web-Stanford	**980121(980582.7)**	1007522(1008040.5)	987255(987255)
Wiki-Vote	**107222**(107227.9)	**107222(107223.5)**	107234(107249.3)
WikiTalk	**3478539(3478544.5)**	3478560(3478579)	N/A
333SP	**95311299(96023116.9)**	104222969(104222969)	N/A
as-22july06	**193529(193542.3)**	193557(193560.8)	193562(193581.1)
audikw1	**544788(546375.8)**	645510(646619.3)	561656(561656)
belgium.osm	**117292752(117713316.1)**	N/A	N/A
cage15	**18904266(18939673.7)**	22288856(22296289.5)	N/A
caidaRouterLevel	**4324957(4327477.8)**	4376005(4377893.3)	N/A
citationCiteseer	**4434164(4439087.7)**	4525350(4527033.5)	4466529(4466529)
cnr-2000	**2443499(2444996)**	2457027(2457309.8)	2449713(2449713)
coAuthorsCiteseer	**3701461(3702751.2)**	3717505(3718382.5)	N/A
coAuthorsDBLP	**4738187(4739697.2)**	4765060(4765668.3)	4749845(4749845)
cond-mat-2005	**482173(482484.3)**	487804(488072)	486812(487278.3)
coPapersCiteseer	**2840116(2848253.2)**	2928376(2929221)	N/A
coPapersDBLP	**3779813(3790462.5)**	3883208(3885354)	N/A
ecology1	**37169194(37512291.1)**	40995892(41068701.3)	N/A
eu-2005	**3186216(3187215.3)**	3212190(3212849.5)	3193332(3193332)
G_n_pin_pout	**706058(707810.3)**	793613(797417.8)	745885(745885)
in-2004	**8493855(8495948.8)**	8540255(8542346.3)	N/A
kron...logn16	**369629(369629)**	370490(370553.5)	370386(370495.5)
ldoor	**2130615(2131629.9)**	2607563(2615331.8)	N/A
luxembourg.osm	**9954051(9955957.2)**	10123554(10232006.8)	N/A
pref...Attachment	**544964(545494.2)**	582867(583472.8)	564066(564066)
rgg_n_2_17_s0	**1143351(1145682.7)**	1411146(1422660.5)	1194638(1194638)
rgg_n_2_19_s0	**3619945(3623934.4)**	4882585(4902738)	N/A
rgg_n_2_20_s0	**6597646(6612833.5)**	9305396(9391407.5)	N/A
rgg_n_2_21_s0	**12315149(12359474.8)**	17639374(17775821.8)	N/A
rgg_n_2_22_s0	**27505305(27607164.4)**	33784024(34175561.5)	N/A
rgg_n_2_23_s0	**50168656(63767170.7)**	N/A	N/A
smallworld	**1218021(1221583.3)**	1311258(1312652.3)	1281937(1281937)
uk-2002	114212809(117625999.3)	**113849945(113854708.5)**	N/A
wave	**975601(978701.5)**	1082203(1083760)	999481(999481)

Table 5 Experiment results on NDR benchmark. If an algorithm fails to find a solution within the cutoff time, it is indicated by "N/A".

Instance	NestedLS *min(avg)*	PBIG *min(avg)*	ACO-e *min(avg)*
bn-human...1-bg	**231444(232173.4)**	248647(248983.5)	234886(234886)
bn-human...2-bg	**193908(194530.6)**	206436(206702)	197614(197614)
ca-coauthors-dblp	**3780154(3790227.3)**	3883208(3885354)	N/A
ca-dblp-2012	**4898659(4900170.1)**	4931550(4932246.8)	4912016(4912016)
ca-hollywood-2009	**4196974(4208205.8)**	4484581(4485353.5)	N/A
channel...b050	**33862787(33944666)**	37854483(37974114)	N/A
dbpedia-link	**153458727(153739853.3)**	154088350(154089188)	N/A
delaunay_n22	**95435272(95559053.8)**	104855386(105650155.3)	N/A
delaunay_n23	**188365284(188544203.8)**	N/A	N/A
delaunay_n24	**379521881(408693281.8)**	N/A	N/A
friendster	**63527982(63557140.7)**	64653832(64656091.5)	N/A
hugebubbles-00020	**970833598(1202159141.6)**	N/A	N/A
hugetrace-00010	**554859414(566474478.5)**	N/A	N/A
hugetrace-00020	**731497084(792040454.8)**	N/A	N/A
inf-europe_osm	**5092357075(5094787915.4)**	N/A	N/A
inf-germany_osm	**941456751(942923953.3)**	N/A	N/A
inf-road-usa	**2375849346(2377787146.5)**	N/A	N/A
inf-roadNet-CA	**92151724(92854875.3)**	98142433(98369828.5)	N/A
inf-roadNet-PA	**50457051(50693249.2)**	54112058(54182813.5)	N/A
rec-dating	**1137467(1137484.5)**	1138910(1139023.8)	1138531(1138531)
rec-epinions	**826618(826642.9)**	831768(832227)	829707(829707)
rec-libimseti-dir	**1209219(1209288.3)**	1213842(1214225)	1212387(1212387)
rgg_n_2_23_s0	**50329249(50441454.7)**	N/A	N/A
rgg_n_2_24_s0	**518533632(713025008.5)**	N/A	N/A
rt-retweet-crawl	8119952(8120894.8)	**8112459(8112605.3)**	N/A
sc-ldoor	**2148767(2153395.2)**	2608260(2628863.8)	N/A
sc-msdoor	**853236(854334.5)**	1006625(1011167)	N/A
sc-pwtk	**441580(442377.9)**	602802(605173.5)	451326(451326)
sc-rel9	**12466895(12494110.1)**	13371415(13373856.3)	N/A
sc-shipsec1	**587596(589711.3)**	673591(675410.8)	607640(607640)
sc-shipsec5	**737132(741136.8)**	840811(849266)	762199(762199)
soc-buzznet	**8275(8275)**	8373(8381.5)	8337(8386.8)
soc-delicious	5684064(5685039.5)	5689449(5689994.5)	**5683212(5683212)**
soc-digg	**6884347(6888681.2)**	6906274(6906978.3)	N/A
soc-dogster	**2343228(2344555.2)**	2373262(2373356)	2352360(2352360)
soc-flickr-und	**29310795(29333534)**	29701624(29702432)	N/A
soc-flixster	9190111(9190239.9)	**9189919(9190039.3)**	N/A
soc-FourSquare	**6055451(6058625.3)**	6063201(6064206)	6062299(6062299)
soc-lastfm	**6747994(6748217.7)**	6748666(6748975.3)	N/A
soc-livejournal	**80381637(80396298.8)**	83450152(83455918.8)	N/A
soc-...-user-groups	**109130362(109143584)**	109708034(109708334)	N/A
soc-LiveMocha	**106551(106560.2)**	108182(108286.5)	107712(107846.8)
soc-ljournal-2008	**103641550(103684264.4)**	105872796(105877923.8)	N/A
soc-orkut	**8377576(8436848.5)**	9246155(9249442.5)	N/A
soc-orkut-dir	**7371792(7388939.2)**	8257778(8260096.8)	N/A
soc-pokec	**18650680(18678287.5)**	19938844(19941250)	N/A
soc-sinaweibo	**5894908130(5894908130)**	N/A	N/A
soc-twitter-higgs	**1160854(1161304)**	1184020(1185051.3)	1170510(1170510)
soc-youtube	**9898687(9900778.7)**	9936591(9937572.8)	N/A
soc-youtube-snap	**23382235(23384447.6)**	23408462(23585903.3)	N/A
socfb-A-anon	**19919414(19952815.8)**	20350881(20351694.5)	N/A
socfb-B-anon	**18669945(18697816.9)**	18997889(18999053)	N/A
socfb-uci-uni	**5866001161(5866001161)**	N/A	N/A
tech-as-skitter	**17726432(17747980.9)**	18668301(18669829)	N/A
tech-ip	**2283(2283.5)**	2986(3010)	2484(2484)
twitter_mpi	**56327895(56337886.8)**	56435803(56436632)	N/A
web-arabic-2005	**2017151(2017601.7)**	2021106(2022620)	2021129(2021129)
web-baidu-baike	**25951517(25969911.1)**	26457056(26457712.3)	N/A
web-it-2004	**3464760(3464814.9)**	3465855(3465914)	N/A
web-uk-2005	**170958(170958)**	**170958**(170958.8)	**170958(170958)**
web-wikipedia_link	**17428644(17452302.6)**	17888836(17889610.3)	N/A
web-wikipedia-growth	**10192627(10212490.8)**	10592826(10594605.3)	N/A
web-wikipedia2009	**37603865(37659492.6)**	38742158(38746820.3)	N/A
wikipedia_link_en	**21240536(21242706.8)**	21362465(21363129.3)	N/A

6.4 Effectiveness of Proposed Strategies

To confirm the effectiveness of *ReSmooth*, we compare NestedLS with its modified versions where NoSmooth does not use this strategy and adopt previous weight smoothing mechanisms SWT and PAWS from [4] and [20], respectively.

The excellent results of NestedLS on massive graphs are mainly due to the inner-layer local search. To confirm its effectiveness, five modified versions are proposed for comparison as follows.

- To confirm the overall effectiveness of inner-layer local search, Alg_1 replaces inner-layer local search with breadth first search to construct the spanning tree for the current candidate solution, as the traditional construction method in [13].
- To confirm the effectiveness of the scoring function in inner-layer local search, Alg_2 and Alg_3 modifies inner-layer local search by not applying the second and third scoring criterion respectively.
- To confirm the effectiveness of WMST selection rule, Alg_4 adopts the same selection rule mentioned as in Section 2.
- To confirm that local search can improve the quality of the spanning tree, Alg_5 constructs the spanning tree without improving it by local search.

The results are shown in Tables 6 and 7. We report the number of instances where NestedLS finds better (worse) solutions than its modified versions, denoted as #better (#worse). The results shown in Table 6 confirm that *ReSmooth* is effective on both classic and massive graphs, and the results shown in Table 7 validate the effectiveness of inner-layer local search on massive graphs.

Table 6 Effectiveness of *ReSmooth*.

		Classic	SNAP	DIMACS	NDR
vs. NoSmooth	#better	59	16	20	37
	#worse	0	3	5	21
vs. SWT	#better	50	17	19	43
	#worse	0	0	5	14
vs. PAWS	#better	14	15	25	52
	#worse	3	6	5	10

Table 7 Effectiveness of *InnerSearch*.

		vs. Alg_1	vs. Alg_2	vs. Alg_3	vs. Alg_4	vs. Alg_5
SNAP	#better	17	16	15	19	15
	#worse	4	5	0	2	6
DIMACS	#better	27	22	18	26	23
	#worse	2	8	11	3	6
NDR	#better	41	41	50	56	48
	#worse	17	20	11	5	12

7 Conclusion

We proposed a local search algorithm NestedLS for MWCDS based on two main ideas, including the restart based smoothing mechanism and the inner-layer local search method. Experiments on classic benchmarks and massive graphs showed its superiority over previous algorithms for MWCDS.

Two proposed ideas can be generally applied to other heuristic algorithms. Specifically, the inner-layer local search method is a general method for maintaining the connectivity constraint when dealing with massive graphs. It contributes to constraint programming by providing not only a better strategy of maintaining the connectivity constraint when dealing with massive instances, but also insights for future study on the connectivity constraints. In addition, the restart based smoothing mechanism provides a novel diversification scheme for restart-based heuristic algorithms.

References

1 Jamal N Al-Karaki and Ahmed E Kamal. Efficient virtual-backbone routing in mobile ad hoc networks. *Computer Networks*, 52(2):327–350, 2008.

2 Christoph Ambühl, Thomas Erlebach, Matúš Mihalák, and Marc Nunkesser. Constant-factor approximation for minimum-weight (connected) dominating sets in unit disk graphs. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 3–14. Springer, 2006.

3 Salim Bouamama, Christian Blum, and Jean-Guillaume Fages. An algorithm based on ant colony optimization for the minimum connected dominating set problem. *Applied Soft Computing*, 80:672–686, 2019.

4 Shaowei Cai and Kaile Su. Local search for boolean satisfiability with configuration checking and subscore. *Artif. Intell.*, 204:75–98, 2013.

5 Orhan Dagdeviren, Kayhan Erciyes, and Savio Tse. Semi-asynchronous and distributed weighted connected dominating set algorithms for wireless sensor networks. *Computer Standards & Interfaces*, 42:143–156, 2015.

6 Zuleyha Akusta Dagdeviren, Dogan Aydin, and Muhammed Cinsdikici. Two population-based optimization algorithms for minimum weight connected dominating set problem. *Appl. Soft Comput.*, 59:644–658, 2017.

7 Tetsuya Fujie. An exact algorithm for the maximum leaf spanning tree problem. *Computers & Operations Research*, 30(13):1931–1944, 2003.

8 Fred Glover, Manuel Laguna, et al. Handbook of combinatorial optimization. *Springer*, pages 2093–2229, 1998.

9 Sudipto Guha and Samir Khuller. Approximation algorithms for connected dominating sets. *Algorithmica*, 20(4):374–387, 1998.

10 John Hopcroft and Robert Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.

11 Raka Jovanovic and Milan Tuba. Ant colony optimization algorithm with pheromone correction strategy for the minimum connected dominating set problem. *Computer Science and Information Systems*, 10(1):133–149, 2013.

12 Michael Levandowsky and David Winter. Distance between sets. *Nature*, 234(5323):34–35, 1971.

13 Bohan Li, Xindi Zhang, Shaowei Cai, Jinkun Lin, Yiyuan Wang, and Christian Blum. Nucds: An efficient local search algorithm for minimum connected dominating set. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, pages 1503–1510, 2020.

14 Jiawei Li, Xiangxi Wen, Minggong Wu, Fei Liu, and Shuangfeng Li. Identification of key nodes and vital edges in aviation network based on minimum connected dominating set. *Physica A: Statistical Mechanics and its Applications*, 541:123340, 2020.

15 Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

16 Tijana Milenković, Vesna Memišević, Anthony Bonato, and Nataša Pržulj. Dominating biological networks. *PloS one*, 6(8):e23016, 2011.

17 Ryan Rossi and Nesreen Ahmed. The network data repository with interactive graph analytics and visualization. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 4292–4293, 2015.

18 Lu Ruan, Hongwei Du, Xiaohua Jia, Weili Wu, Yingshu Li, and Ker-I Ko. A greedy approximation for minimum connected dominating sets. *Theoretical Computer Science*, 329(1-3):325–330, 2004.

19 Shyong Jian Shyu, Peng-Yeng Yin, and Bertrand MT Lin. An ant colony optimization algorithm for the minimum weight vertex cover problem. *Annals of Operations Research*, 131(1-4):283–304, 2004.

20 John Thornton, Duc Nghia Pham, Stuart Bain, and Valnir Ferreira Jr. Additive versus multiplicative clause weighting for sat. In *AAAI*, volume 4, pages 191–196, 2004.

21 Mustafa Tosun and Elif Haytaoglu. A new distributed weighted connected dominating set algorithm for wsns. In *2018 IEEE 8th International Conference on Consumer Electronics-Berlin (ICCE-Berlin)*, pages 1–6. IEEE, 2018.

22 Sattar Vakili and Qing Zhao. Distributed node-weighted connected dominating set problems. In *2013 Asilomar Conference on Signals, Systems and Computers*, pages 238–241, 2013.

23 Dingding Wang, Tao Li, and Mitsunori Ogihara. Generating pictorial storylines via minimum-weight connected dominating set approximation in multi-view graphs. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*, pages 683–689, 2012.

24 Yiyuan Wang, Shaowei Cai, Jiejiang Chen, and Minghao Yin. A fast local search algorithm for minimum weight dominating set problem on massive graphs. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pages 1514–1522, 2018.

25 Yiyuan Wang, Shaowei Cai, Jiejiang Chen, and Minghao Yin. Sccwalk: An efficient local search algorithm and its improvements for maximum weight clique problem. *Artificial Intelligence*, 280:103230, 2020.

26 Yiyuan Wang, Shaowei Cai, and Minghao Yin. Local search for minimum weight dominating set with two-level configuration checking and frequency based scoring function. *Journal of Artificial Intelligence Research*, 58:267–295, 2017.

27 Yu Wang, Weizhao Wang, and Xiang-Yang Li. Efficient distributed low-cost backbone formation for wireless networks. *IEEE Transactions on Parallel and Distributed Systems*, 17(7):681–693, 2006.

28 Yu Wang, Weizhao Wang, and Xiang-Yang Li. *Weighted Connected Dominating Set*, pages 1020–1023. Springer US, 2008.

29 Feng Zou, Yuexuan Wang, XiaoHua Xu, Xianyue Li, Hongwei Du, Peng-Jun Wan, and Weili Wu. New approximations for minimum-weighted dominating sets and minimum-weighted connected dominating sets on unit disk graphs. *Theor. Comput. Sci.*, 412(3):198–208, 2011.