The technical details for paper *A Two-level Meta-heuristic Approach for the Minimum Dominating Tree Problem*

Caiquan XIONG , Hang LIU , and Xinyun WU[*]

School of Computer Science, Hubei University of Technology,
Wuhan, China, xinyun@hbut.edu.cn
[*]Corresponding author

October 5, 2022

1 Highlights

1. A hierarchical meta-heuristic framework consisting of two nested local search procedures to tackle the MDT problem.
2. An effective neighborhood structure for the inner layer local search and a fast neighborhood evaluation method that enable TLMH to achieve a better balance between exploitation and exploration.
3. A sampling phase that quickly determines the promising search space providing a better convergence for TLMH.
4. Tests conducted on 72 public instances of the MDT problem, disclosing that TLMH improves the best-known solutions from heuristics in 14 instances.
5. Benchmark *RangL* providing 18 large instances for future comparison.
6. Extensive experiments to analyze the contributions of key features of our algorithm establishing that these primary ingredients when worked in unison outperform the cases where they are applied in isolation.

2 Preliminary Discussion

2.1 Definitions

Given an undirected weighted graph $G = (V, E)$ where each edge is assigned a positive weight, a subgraph of G denoted as $T = (V', E')$ is said to be a dominating tree if any vertex not in V' is adjacent to at least one vertex of V' with

the condition that T is connected and without any cycle. The *minimum dominating tree* (MDT) problem aims to find a dominating tree with the minimum weight for a given graph.

In this paper, the problem instance we operate on is a simple undirected weighted graph $G = (V, E)$ where V is the set of vertices and E is a set of weighted edges. A weight function $w : V \to R^+$ associates each edge $e \in E$ a positive weight. To better describe our proposed algorithm for MDT problem, we give the following definitions.

Articulation Points A vertex $v \in V$ is an articulation point of G if its removal disconnects the graph.

Connected Vertices Set We say that a vertices set $X \subseteq V$ is a connected vertices set if the subgraph $G[X]$ deduced by X is connected.

Dominating Set A vertices set $X \subseteq V$ is a dominating set of $G = (V, E)$ if its closed neighborhood $\Gamma_G[X]$ equals to V, where $\Gamma_G[X] = X \cup \{v \in V | u \in X, \{u, v\} \in E\}$.

Connected Dominating Set If a connected vertices set is also a dominating set, we call it a Connected Dominating Set (CDS).

Spanning Tree The acyclic graph $T = (V^T, E^T)$ is a spanning tree of graph $G = (V, E)$ if $V^T = V$ and $E^T \subseteq E$.

Dominating Tree Given a CDS X for graph G, a dominating tree is a spanning tree of the subgraph of G deduced by X ($G[X]$).

Minimum Dominating Tree The dominating tree for a graph with the smallest edge weights summation, MDT for short.

2.2 MDTP and MCDSP

The MDT problem and the MCDS problem share some similarities with each other. Both problems aim to find a sub-structure that dominates the whole vertices set. Given an undirected graph $G = (V, E)$ and its dominating set X, a feasible dominating tree can get from a feasible dominating set X by determining the spanning tree of the graph $G[X]$. For unweighted graphs or weighted graphs with similar edge weights, an algorithm for the MCDS problem can be used to solve the MDT problem. Since, in this case, the MDT problem is looking for the dominating tree with the minimum number of edges (or the minimum number of vertices), i.e., the minimum dominating set.

If the edge weights are different in the instance graph, the dominating tree weight tends to be smaller if deduced from a smaller CDS. To check this, we perform a simple experiment to compare the weight of dominating trees deduced from CDSs with different sizes. We find CDSs with different sizes as much as possible for one problem instance and record the corresponding dominating tree

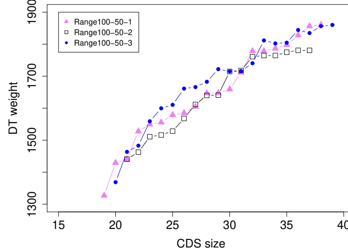
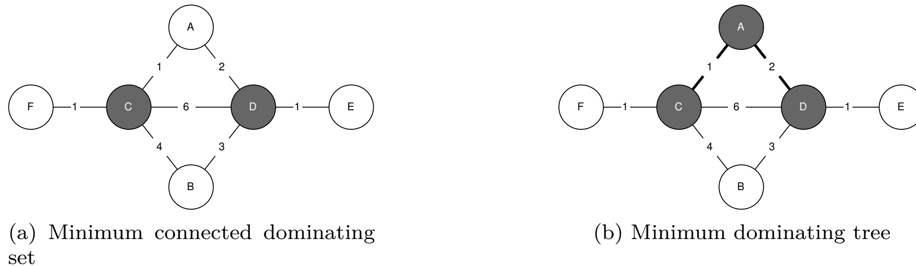


Figure 1: Relationship between dominating tree weights and the CDS sizes

weights. Figure 1 illustrates the results for three problem instances (They are from the public benchmark introduced in Section 4).

From Figure 1, we observe that, although it is not always the case, the CDS with a smaller size usually produces a smaller dominating tree. This phenomenon appears on other instances if the same experiment is performed. However, for most cases, the minimum spanning tree from the minimum dominating set may be not the MDT. The dominating tree with more vertices may get a smaller weight compared to those with fewer vertices. For example, in Fig. 2, the MDT is the minimum spanning tree of $\{A, C, D\}$ rather than the MCDS $\{C, D\}$.



(a) Minimum connected dominating set

(b) Minimum dominating tree

Figure 2: MCDS vs MDT

Although the algorithms for MCDS cannot be used to solve the MDT directly, we can use them to facilitate the solving of the MDT problem. Given a CDS configuration, the minimum spanning tree of the subgraph deduced from this CDS is always a feasible dominating tree of the original graph. Finding a minimum spanning tree of a subgraph can be done in $O(|V|^2)$ time by implementing Kruskal's or Prim's algorithm on the subgraph. We have already known that the minimum spanning tree of the MCDS may not be the optimal MDT. However, from the experiment, we observe that the dominating trees with fewer vertices usually get less weight than those with more vertices. Thus, we can focus on exploring the feasible dominating trees among the configura-

tions of fewer vertices. More specifically, we explore the feasible dominating trees among the CDS configurations of which the vertices number is close to the optimal MCDS configuration. This indicates that the approach of determining k-CDS is critical to our research. In the following sections, we describe how the algorithm of k-CDS is embedded into our approach.

3 Solution Method

In this section, we describe the proposed TLMH algorithm in detail. The TLMH algorithm consists of two local search processes constructed in a multi-layered structure. The outer layer is a simple local search maintaining a solution pool. The inner one is another more sophisticated local search, aiming to find a MDT with a fixed number of vertices.

3.1 Main Framework

The proposed TLMH algorithm follows a general framework consisting of two phases: Sampling and Local-Search[1], as described in Algorithm 1. The configuration $T = (X, E')$ that TLMH manipulates is a compound structure consisting of a vertex set X and the minimum spanning tree E' of the deduced graph $G[X]$. During the process of TLMH, X for all the configurations are restricted to be a connected vertices set of graph G. If X is a CDS of graph G, the minimum spanning tree in that configuration is a dominating tree of graph G. The configuration pool $\mathcal{T}$ maintains one best configuration for each CDS size, e.g., $\mathcal{T}[i]$ represents the best configuration with i vertices.

The initialization and sampling phase (line 2) quickly provides a pool of initial dominating tree configurations ($\mathcal{T}$) and determines the CDS sizes that are promising to produce a dominating tree with less weight. Lines 3-11 are the processes of the outer layer local search. In each iteration, we retrieve the best configuration T_0 from $\mathcal{T}$ and try to improve this configuration using the inner layer local search process (Section 3.3.1). Then we retrieve the neighbor configurations (T_ι) of T_0 from $\mathcal{T}$, of which the CDS size differs one or two compared to T_0, and improve them also using the same method. Finally, $\mathcal{T}$ is updated if any newly obtained configuration is better than the one in it for the specific CDS size. In Algorithm 1, $|T_\iota|$ represents the number of vertices of the dominating tree of the configuration T. Note that, in Line 6, if the configuration $\mathcal{T}[k]$ does not exist, a random configuration with $|X| = k$ is provided to the procedure TryKDTP. This outer local search process is designed as simply as possible to provide better performance. The termination condition for TLMH algorithm can be set as the time limit.

In the following, we first describe how the procedure InitAndSample works, and then discuss the core part of the proposed TLMH, i.e., the procedure TryKDTP looking for a MDT with a fixed number of vertices.

Algorithm 1 TLMH

Input: The instance graph $G(V, E)$
Output: A DPT configuration T_b

```
 1: procedure TLMH(G)
 2:     𝒯 ← InitAndSample(G)
 3:     repeat
 4:         T_b ← the best configuration in 𝒯
 5:         for ι ∈ {0, 1, −1, 2, −2} do
 6:             T_ι ← TryKDTP(G, 𝒯[|T_b| + ι], 𝒯)
 7:             if T_ι is better than 𝒯[|T_ι|] then
 8:                 𝒯[|T_ι|] ← T_ι
 9:             end if
10:         end for
11:     until The termination condition is met
12:     T_b ← the best configuration in 𝒯
13:     return T_b
14: end procedure
```

3.2 Initialization and Sampling

The procedure InitAndSample samples each possible CDS size a corresponding dominating tree and groups these dominating trees as an initial solution pool for later use. This solution pool also provides an estimation about the CDS size that is promising to produce better dominating trees. Algorithm 2 describes the process of InitAndSample.

Let us denote g_t the minimum spanning tree of G. The sub-structure obtained from removing the leaves of g_t is a feasible dominating tree of G. Thus, we find the first feasible solution $(V', E'(V'))$ where V' is the non-leaf vertices of g_t and $E'(V')$ is the minimum spanning tree of the deduced graph $G[V']$, i.e., g_t removing leaves. Starting from the k value of $|V'| - 1$, we find a CDS X with k vertices and calculate the MDT of the deduced graph $G[X]$. Then this spanning tree is temporarily considered the best one for the size k and stored into the solution pool. The iteration stops when no feasible CDS can be found for the current k value. The solution pool $\mathcal{T}$ is returned for later use. In this way, we quickly find the proper CDS sizes that may produce better dominating trees.

To find a CDS, an MCDS solver can be used. Since this procedure only provides an estimation and, as described in Section 2.2, the MCDS does not usually produce the MDT, we do not need a strong MCDS solver but a fast one. In this paper, we use a simplified version of the RNS-TS algorithm introduced in the literature [2], the RNS-TS without tabu mechanism and perturbation operations.

Algorithm 2 Initialization and Sampling

Input: $G(V, E)$
Output: The configuration pool $\mathcal{T}$

1: **procedure** InitAndSample(G)
2: $\mathcal{T} \leftarrow []$
3: $g_t \leftarrow$ the minimum spanning tree of G
4: $V' \leftarrow$ the non-leaf vertices in g_t
5: $\mathcal{T}[|V'|] \leftarrow (V', E'(V'))$
6: $k \leftarrow |V'| - 1$
7: **repeat**
8: Find a CDS X of G with k vertices.
9: $\mathcal{T}[k] \leftarrow$ the minimum spanning tree of $G[X]$
10: $k \leftarrow k - 1$
11: **until** No feasible X exists
12: **return** $\mathcal{T}$
13: **end procedure**

3.3 Solving k-DTP

Our approach to tackling the MDT problem is based on a serial minimization approach denoted as k-DTP. k-DTP searches the MDT of a graph while the number of vertices is restricted to k. Our algorithm repeatedly tries to find an MDT of k vertices in G while the value k is heuristically varied during the search process. In this section, we present the inner local search procedure of the TLMH (TryKDTP) to tackle the k-DTP problem.

3.3.1 Inner local search procedure for k-DTP

Starting from an initial configuration T_0, procedure TryKDTP tries to find a MDT with $|T_0|$ vertices. It is described in Algorithm 3. In each iteration, the algorithm applies one neighborhood move that improves the current configuration most. The process continues until the termination condition is met. We introduce a procedure Perturbation to diversify the search process, thus obtaining better results. It maintains the solution pool $\mathcal{T}$ and the best configuration T_{best} found in this round and triggers a diversification operator when the process is trapped in a local optimum.

The termination condition here can be set as the time consumed, the overall iterations or the number of perturbations performed, etc. In this paper, we set the condition to be the number of perturbations. More specifically, TryKDTP terminates if the perturbation has been triggered over five times. Note that in the scope of TryKDTP procedure, the number of vertices in X does not change.

3.3.2 Search space and cost function

The search space (set of configurations) explored by this inner local search procedure is denoted by S notation. A configuration $T = (X, E')$ is a compound

Algorithm 3 Local Search k-DPT

Input: $G = (V, E)$, Initial configuration T_0, $\mathcal{T}$
Output: A DPT configuration T_{best}

1: **procedure** TryKDTP($G, T_0, \mathcal{T}$)
2: $T_{best} \leftarrow T \leftarrow T_0$
3: **repeat**
4: $mv \leftarrow$ FindMove(G, T)
5: $T \leftarrow$ MakeMove(G, T, mv)
6: $T, T_{best}, \mathcal{T} \leftarrow$ Perturbation($T, T_{best}, \mathcal{T}$)
7: **until** The termination condition is met
8: **return** X_{best}
9: **end procedure**

structure where $X \in V$ is any connected set of k vertices, and E' is the minimum spanning tree of the deduced graph $G[X]$. Given a configuration $T = (X, E') \in S$, we denote by $X^+ = \Gamma_G(X) \backslash X$ the set of vertices that do not belong to X and are dominated by X, and by $X^- = V \backslash \Gamma_G(X)$ the set of vertices that are not dominated by X.

The cost $f(T)$ of configuration T is defined as

$$f(T) = \alpha f_1(X) + f_2(E') \quad (1)$$

where $f_1(X) = |X^-|$ represents the number of vertices that are not dominated by T and $f_2(E') = \sum_{e \in E'} c(e)$ represents the weight of the minimum spanning tree of $G[X]$. α is a large constant to make f_2 inferior to f_1. Please note that a configuration T represents a feasible solution of k-DTP if and only if $f_1(X) = 0$.

3.3.3 Neighborhood definition

Given a configuration $T = (X, E') \in S$, we denote by $< x, y, \psi >$ the move operator, by which a vertex $x \in X$ is removed from X, another vertex $y \in V \backslash X$ is added into X, and ψ is the process of determining the minimum spanning tree of graph $G(X)$. In addition, $T \oplus < x, y, \psi >$ denotes the configuration obtained by applying the operator $< x, y, \psi >$ to T. Thus we have

$$T \oplus < x, y, \psi >= (X \backslash \{x\} \cup \{y\}, \psi(G(X))) \quad (2)$$

However, as changing vertices in X may disconnect the configuration, such that the spanning tree determining process becomes impossible, the set of operators applicable to T is defined as a restricted set. This set, denoted by $M(T)$, consists of the set of operators $< x, y, \psi >$ satisfying the following three properties: (1) x must not be an articulation vertex of graph $G[X]$; (2) y must belong to set X^+; (3) if x is the only vertex that y connects to X, this operator is excluded from $M(T)$. Figure 3 illustrates a move operation of the proposed algorithm.

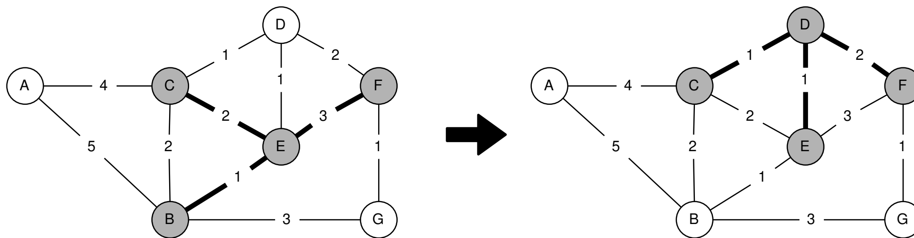


Figure 3: Move $< B, D, \psi >$

3.3.4 Neighborhood evaluation

Given a move $m \in M(S)$, we denote by $\delta(m)$ the score of m, i.e., its impact on the score of the configuration, defined according to

$$\delta(m) = f(T \oplus m) - f(T) \tag{3}$$

However, it is time consuming to use this equation to calculate δ directly, since the time complexity of calculating f_1 is $O(|V|)$ and f_2 is at least $O(|E(X)| + |X| \log |X|)$. There is a method to reduce the calculation by an incremental evaluation. In most of the case, one operator only changes a small part of the configuration. Thus, if these changed parts can be determined, δ can be calculated in a faster manner. We will describe how the fast incremental evaluation is implemented for f_1 and f_2 respectively in Section 3.3.5 and Section 3.3.6.

The procedure FindMove represents the process of neighborhood evaluation of the current configuration, i.e., how the best move for the current configuration is found. It returns a move with the least δ value. Procedure MakeMove applies the chosen move to the current configuration.

To prevent the search process trapped in the local optimum trap too early, we implement a tabu mechanism [3] in this local search procedure. Each time when a move $< x, y, \psi >$ is performed, we set vertex y a tabu state with a tabu tenure tt randomly between $ttMin$ and $ttMax$, where $ttMin$ and $ttMax$ are tunable parameters of the proposed TLMH. For the following tt iterations, any moves related to y are excluded from the neighborhood $M(T)$.

3.3.5 Fast incremental evaluation for f_1

For f_1, we use the same fast incremental evaluation technique from one of the state-of-the-art MCDS algorithms RSN-TS[2]. The neighborhood evaluation procedure maintains an array $L[i]$ measuring the number of vertices in X connected to vertex i.

$$L[i] = |\{j|\{i, j\} \in E, j \in E)\}| \tag{4}$$

With the information from $L[i]$, we can quickly assert that the vertex i is in $\Gamma_G(X)$ as long as $L[i]$ is greater than 0. Thus, f_1 can be calculated by:

$$\delta(m) = \delta^-(x) + \delta^+(y) \tag{5}$$

where $\delta^-(x)$ and $\delta^+(y)$ represents the changing value caused by removing or inserting operation. $\delta^-(x)$ can be calculated by counting the neighbors of x, which belongs to X^+, and the corresponding L value is 1. $\delta^+(y)$ can be calculated by counting the neighbors of y, which belongs to X^-, and the corresponding L value is 0. There is one exception when calculating $\delta^-(x)$, if the vertex x is connected to y, then it isn't taken into count. From the above description, we know that the time complexity of evaluating $\delta(m)$ is $O(|C_x| + |C_y|)$, where C_x and C_y represent the set of neighbors for x and y respectively.

After a move $< x, y, \psi >$ is performed, L needs to be updated to make the value consistent with the current configuration. The L values correspond to the neighbors of x and y need to be refreshed. Specifically, for all the neighbors of x, the L value is decreased by 1, while for all the neighbors of y, the L value is increased by 1.

3.3.6 Evaluation for f_2

The core of the calculation for f_2 is how to determine the minimum spanning tree for the sub-graph $G(X)$, i.e., the process ψ in Equation (2). Although the algorithm of determining the minimum spanning tree is quite mature and can be done in polynomial time using Prim's or Kruskal's algorithm, it is still time-consuming if it has to be executed once for the evaluation of each candidate move. Considering that there are a large number of candidate moves to be evaluated for each iteration of the neighborhood evaluation, the time consumed by the minimum spanning tree determination is undesirably too much.

To alleviate the calculation workload, we only calculate f_2 when $f_1 = 0$. During the evaluation, f_1 is always calculated before f_2. If $f_1 > 0$, it indicates that there exists vertex that is not dominated. Thus, in this situation, the minimum spanning tree of X does not dominate the whole graph, i.e., the tree weight is insignificant because of its impracticality. Therefore, we do not calculate f_2 here and simply assign $f_2 = 0$. The value of f_2 is calculated when $f_1 = 0$, i.e., the spanning tree dominates all the vertices. From Equation (1), we know that there is a large constant parameter α on f_1, thus, the value of f_2 does not impact f much and the evaluation results depend on f_1 only if $f_1 > 0$. Which is to say, when $f_1 > 0$, we can neglect f_2 safely. Our algorithm performs Kruskal's algorithm on the current $G[X]$ when f_2 is needed for the evaluation. We implement Kruskal's algorithm [4] with the disjoint-set data structure [5] such that the time complexity is $O(n \log n)$. Please note that Prim's algorithm [6] is also acceptable here.

3.4 Diversification

We implement a random perturbation operator as a diversification mechanism to improve further the solution quality. Perturbation can be considered as a jump in the search space. It is triggered when the search process is trapped in a local optimum. Inspired by the breakout local search algorithm [7, 8, 9, 10], the perturbation operator consists of a series of random moves of which the

number depends on the current search state. As described in Algorithm 3, the PERTURBATION procedure is executed in each iteration of the TRYKDTP. The details of PERTURBATION procedure are described in Algorithm 4.

Algorithm 4 The procedure of the perturbation operator PERTURBDIV

Input: Current configuration T, Best configuration for this round T_{best}, $\mathcal{T}$
Output: The updated X, T_{best} and $\mathcal{X}$
1: **procedure** PERTURBATION($X, T_{best}, \mathcal{X}$)
2:     **if** $(f(T) = f(\mathcal{T}[|T|])$ **and** $T \neq \mathcal{T}[|T|])$ **or** $f(T) < f(\mathcal{T}[|T|])$ **then**
3:         $\mathcal{T}[|T|] \leftarrow T$, $g_str \leftarrow minStr$
4:     **else if** $T = \mathcal{T}[|T|]$ **then**
5:         $g_str \leftarrow \min\{g_str + 1, maxStr\}$
6:     **end if**
7:     **if** $f(T) < f(T_{best})$ **then**
8:         $T_{best} \leftarrow T$, $g_stagIters \leftarrow 0$
9:     **else**
10:         $g_stagIters \leftarrow g_stagIters + 1$
11:     **end if**
12:     **if** $g_stagIters > \alpha$ **then**
13:         $T \leftarrow \mathcal{T}[|T|]$
14:         **repeat**
15:             $mv \leftarrow$ generate a random move
16:             $X \leftarrow$ MAKEMOVE(X, mv)
17:         **until** g_str times
18:         $g_stagIters \leftarrow 0$
19:     **end if**
20:     **return** T, T_{best}, $\mathcal{T}$
21: **end procedure**

PERTURBATION maintains two global variables, g_str measuring the number of random moves to be performed and $g_stagIters$ counting the number of iterations failed to improve the best configuration T_{best}. g_str is initialized as $minStr$ at the beginning of TRYKDTP procedure. If the current configuration is identical to the one stored in pool $\mathcal{T}$, g_str is incremented by 1 while bounded by $maxStr$ (line 5 in Algorithm 4). It is reset to $minStr$ when a configuration better than $\mathcal{T}[|T|]$ or an equally good but different one is found (line 2). $minStr$ and $maxStr$ are determined by the equations $minStr = perturbMin \times |X|$ and $maxStr = perturbMax \times |X|$, where $perturbMin$ and $perturbMax$ are tunable parameters of the proposed TLMH. Variable $g_stagIters$ is also initialized at the beginning of TRYKDTP with value 0. It is reset to 0 when the current best configuration T_{best} updates, and is increased by 1 for other cases (lines 7 to 11). The actual diversification process (lines 12 to 19) is only executed when $g_stagIters$ exceed the threshold α, where α is a tunable parameter of the proposed TLMH. The core idea of this perturbation operation lies in that if the search process keeps meeting the same local optimum, it indicates a local optimum trap. Thus, a more intense perturbation should be performed to help the algorithm jump out of this local optimum trap. Note that the diversification

is always performed on the ever best configuration but not the current one (line 13).

4 Computational Results and Comparisons

In this section, we report the experimental results of TLMH on the benchmark instances widely used in the literature and compare the results with several reference algorithms.

4.1 Problem instances and experiment protocol

We used the *DTP* benchmark introduced by Dražić et al. [11] and *Range* benchmark introduced by Sundar and Singh [12]. For the randomly generated benchmark *DTP*, we used the larger-scale portion (18 instances) of the benchmark where vertex-set size of the instances is in $\{100, 200, 300\}$ and the edge-set size is in $\{150, 200, 400, 600, 1000\}$. The *Range* benchmark (54 instances) is also randomly generated but according to a certain vertex transmission range. The number of vertices is in $\{50, 100, 200, 300, 500\}$ and the transmission range is in $\{100m, 125m, 150m\}$. We also introduce *RangeL* benchmark (18 instances) in this paper similarly generated as *Range* but with much larger graphs. All these instances can be downloaded online or obtained from the author.

The TLMH algorithm is programmed in the Java programming language [1]. All experiments for TLMH were tested on a PC with Intel Core i7 2.9GHz CPU and 16GB RAM with JDK 11. The referenced meta-heuristics ABC_DT, AOC_DT and EA/G-MP are implemented in C and tested on a 3.0GHz Intel Core 2 Duo processor-based system with 2GB RAM, while ABC_DTP is also implemented in C but tested on a 1.6 GHz Core 2 Duo system with 1GB RAM. The other two referenced meta-heuristics VNS and GAITLS are implemented in C and C++ and tested on an Intel(R) Xeon(R) CPU E7-4803 2.13GHz with 8GB memory. We also compared our proposed TLMH with an exact approach introduced by Álvarez-Miranda et al. [13]. Their algorithm was tested on an Intel Core i7-4702MQ with 2.2GHz and 4GB RAM.

4.2 Parameter tuning

In this section, we conduct a preliminary experiment to identify and fix the values of key parameters used in the TLMH algorithm.

- Parameters $ttMin$ and $ttMax$ determine the range of the tabu tenure (tt). We have tested three possible values for these parameters: $[ttMin, ttMax] = [5, 10], [10, 50]$ and $[50, 100]$
- Parameter α corresponds to the period used to apply a perturbation. We have tested three values for this parameter: $\alpha = 100, 250$ and 500.

[1]The executable file and the source code can be downloaded from https://github.com/xavierwoo/DTPSolver (uploaded and accessed on July 11th 2022)

- Parameter $perturbMin$ and $perturbMax$ bound the perturbation strength. We have tested three possible values for these parameters $[perturbMin, perturbMax] = [0.1, 0.3], [0.3, 0.5]$ and $[0.5, 0.8]$
- The time limit is set to be 1000 seconds for all the experiments in this paper.

In this experiment, we have used 18 representative instances from the *Range* benchmark, which are 50-1, 100-1, 200-1, 300-1, 400-1, and 500-1 from the three transmission range sets $\{100m, 125m, 150m\}$.

We used the scientific tuning tool *irace* [2] [14] to do the automatic parameter tuning work. *irace* automatically generated 71 configurations (combinations between parameter settings and instances) for the evaluation. From the report of *irace*, the parameter setting $[ttMin, ttMax] = [5, 10]$, $\alpha = 250$ and $[perturbMin, perturbMax] = [0.5, 0.8]$ fits this benchmark most. In the following experiment, we fix the parameter to this setting. Moreover, we observe that no matter how the setting changes, the results do not show much difference on the smaller scaled instances. All the settings produce the same best and average result on all 50-1 instances, and they produce the same best result on all 100-1 and 200-1 instances. This indicates the robustness of the proposed TLMH algorithm.

Note that this experiment does not guarantee the best parameter values. The best one may vary on different benchmarks.

4.3 Experiment on DTP large benchmark

In this section, we tested the proposed TLMH on the DPT benchmark. Since all the algorithms produce the same results very quickly for the small-scale instances of this benchmark, we only show the results for the large-scale portion. The detailed experimental results and the comparison among VNS, GAITLS, and the exact algorithm are presented in Table 1. The start marks that TLMH obtains a better best value than all the other meta-heuristics in this table. The bolded value represents that the corresponding algorithm performs no worse than the other meta-heuristics in this criterion. The value with a + mark in the EXACT column represents that it is not proven optimal by the algorithm. The time column for TLMH represents the average time in seconds that TLMH consumed to obtain the best results. This format applies to all the following tables.

From Table 1, we observe that TLMH outperforms VNS with a large margin. Comparing to GAITLS, TLMH produces better best results by outperform on five instances (200-400-1, 200-400-2, 300-600-1, 300-600-2, and 300-1000-0) while GAITLS produces better overall average results. Comparing to the exact algorithm, TLMH is able to obtain the same results for instances that the exact algorithm can proof optimality. For the last three instances that the exact

[2] The documentation for *irace* can be found at https://www.rdocumentation.org/packages/irace/versions/3.4.1 (accessed on July 11th 2022)

Table 1: Computational results of TLMH and comparisons on DTP

instance	TLMH			VNS		GAITLS		EXACT
	best	average	time	best	average	best	average	
100-150-0	**152.57**	**152.57**	2	**152.57**	154.61	**152.57**	**152.57**	152.57
100-150-1	**192.21**	**192.21**	11	**192.21**	194.22	**192.21**	**192.21**	192.21
100-150-2	**146.34**	**146.34**	87	**146.34**	148.35	**146.34**	**146.34**	146.34
100-200-0	**135.04**	**135.04**	60	**135.04**	136.41	**135.04**	**135.04**	135.04
100-200-1	**91.88**	**91.88**	13	**91.88**	92.03	**91.88**	**91.88**	91.88
100-200-2	**115.93**	**115.93**	9	115.93	117.11	**115.93**	**115.93**	115.93
200-400-0	**257.09**	257.23	370	306.06	343.95	**257.09**	**257.09**	257.09
200-400-1	***258.77**	**258.88**	486	303.53	331.1	258.93	258.93	258.77
200-400-2	***238.27**	241.72	370	274.37	389.51	238.29	**238.29**	238.27
200-600-0	**121.62**	127.73	460	132.49	150.39	**121.62**	**121.62**	121.62
200-600-1	**135.08**	145.20	441	162.92	198.21	**135.08**	**135.08**	135.08
200-600-2	**123.31**	123.70	264	139.08	154.36	**123.31**	**123.31**	123.31
300-600-0	**348.03**	351.22	529	471.69	494.62	**348.03**	**348.03**	348.03
300-600-1	***413.93**	416.64	753	494.91	542.46	415.32	**415.32**	413.93
300-600-2	***352.15**	**353.77**	760	500.72	535.3	385.53	385.53	352.15
300-1000-0	***148.63**	150.10	629	257.72	264.33	149.57	**149.57**	147.17^{+}
300-1000-1	165.21	165.91	477	242.79	325.16	**165.19**	**165.19**	165.32^{+}
300-1000-2	154.64	169.39	595	233.18	251.41	**154.61**	**154.61**	154.59^{+}
average	***197.26**	199.75	351	241.86	267.97	199.25	**199.25**	197.18

algorithm fails to proof optimality, TLMH produces better result on instance 300-1000-1 while slightly worse results on the other two.

4.4 Experiment on Range benchmark

In this section, we tested the proposed TLMH on 54 public problem instances from *Range* widely used in the literature [15, 16]. The parameters of TLMH have been fixed guided by the preliminary experiment reported in Section 4.2. The detailed experimental results and the comparison among ABC_DT, ACO_DT, EA/G-MP, ABC_DTP and the exact algorithm are presented in Table 2, Table 3 and Table 4.

Table 2: Computational results of TLMH and comparisons on Range-100

instance	TLMH			ABC_DT		ACO_DT		EA/G-MP		ABC_DTP		EXACT
	best	average	time	best	average	best	average	best	average	best	average	
50-1	**1204.41**	**1204.41**	1	**1204.41**	**1204.41**	**1204.41**	**1204.41**	**1204.41**	**1204.41**	**1204.41**	**1204.41**	1204.41
50-2	**1340.44**	**1340.44**	<1	**1340.44**	**1340.44**	**1340.44**	**1340.44**	**1340.44**	**1340.44**	**1340.44**	1340.69	1340.44
50-3	**1316.39**	**1316.39**	<1	**1316.39**	**1316.39**	**1316.39**	**1316.39**	**1316.39**	**1316.39**	**1316.39**	**1316.39**	1316.39
100-1	**1217.47**	**1217.47**	17	**1217.47**	1218.15	**1217.47**	**1217.47**	**1217.47**	1217.61	**1217.47**	1218.59	1217.47
100-2	**1128.40**	**1128.40**	44	**1128.40**	1128.42	1152.85	1152.85	**1128.40**	1128.54	**1128.40**	1136.50	1128.40
100-3	**1252.99**	1253.41	202	**1252.99**	**1253.14**	1253.49	1253.49	1253.49	1257.37	**1252.99**	1253.30	1252.99
200-1	**1206.79**	**1206.80**	515	**1206.79**	1209.52	**1206.79**	1207.61	**1206.79**	1208.26	**1206.79**	1210.25	1206.79
200-2	***1213.24**	**1213.27**	395	1216.41	1219.74	1216.23	1217.73	1216.41	1222.23	1216.41	1219.38	1213.24
200-3	**1247.25**	**1247.41**	313	1253.02	1258.06	**1247.25**	1248.94	1247.63	1250.78	1247.73	1252.15	1247.25
300-1	**1215.48**	**1217.40**	564	1229.97	1237.47	1228.24	1243.70	1225.22	1230.48	**1215.48**	1220.39	1215.48
300-2	**1170.85**	**1171.08**	341	1182.52	1200.79	1176.45	1193.95	**1170.85**	1171.30	**1170.85**	1171.15	1170.85
300-3	***1247.51**	**1249.51**	348	1257.21	1271.20	1261.18	1276.75	1252.14	1260.83	1249.54	1254.67	1247.51
400-1	***1211.33**	**1213.51**	502	1223.61	1241.75	1220.62	1237.45	1211.72	1220.79	1212.51	1214.36	1211.33
400-2	***1197.66**	**1198.99**	432	1220.54	1235.29	1209.69	1246.14	1199.92	1202.82	1199.23	1202.9	1197.66
400-3	***1245.31**	**1248.47**	633	1266.41	1276.80	1254.10	1270.34	1248.29	1268.38	1246.94	1258.76	1245.25
500-1	***1197.26**	**1202.81**	678	1233.14	1241.60	1219.66	1240.05	1206.07	1222.12	1200.06	1208.73	1201.31^{+}
500-2	1221.76	**1226.81**	570	1245.59	1258.33	1273.86	1295.51	1226.78	1240.62	**1220.68**	1230.07	1220.47
500-3	***1231.84**	1236.64	583	1249.17	1278.67	1232.71	1259.08	1232.15	1250.48	1231.95	**1236.33**	1231.81
average	***1225.91**	**1227.32**	348	1235.80	1243.90	1235.10	1245.68	1228.03	1234.10	1226.57	1230.50	1226.06

From Table 2, we observe that TLMH outperforms other meta-heuristics in terms of the best and the average value obtained, for the larger scaled instances,

i.e., the instances over 200 vertices. TLMH fails on instance rang100/500-2 giving the result of 1225.29 where ABC_DTP produces a better result of 1220.68. However, TLMH produces a better average result in this instance comparing to others. In terms of the overall average values, TLMH outperforms the other meta-heuristics. And the average best of TLMH is better than the exact algorithm because there is one instance (500-1) that the exact algorithm cannot solve to optimal.

Table 3: Computational results of TLMH and comparisons on Range-125

instance	TLMH			ABC_DT		ACO_DT		EA/G-MP		ABC_DTP		EXACT
	best	average	time	best	average	best	average	best	average	best	average	
50-1	**802.95**	**802.95**	1	**802.95**	**802.95**	**802.95**	803.26	**802.95**	**802.95**	**802.95**	**802.95**	802.95
50-2	**1055.10**	**1055.10**	2	**1055.10**	**1055.10**	**1055.10**	**1055.10**	**1055.10**	**1055.10**	**1055.10**	**1055.10**	1055.10
50-3	**877.77**	**877.77**	4	**877.77**	**877.77**	**877.77**	**877.77**	**877.77**	**877.77**	**877.77**	877.83	877.77
100-1	**943.01**	**943.01**	102	**943.01**	**943.01**	**943.01**	946.37	**943.01**	**943.01**	**943.01**	**943.01**	943.01
100-2	**917.00**	917.23	281	**917.00**	**917.03**	935.71	938.71	917.95	917.95	**917.00**	917.38	917.00
100-3	**998.18**	**998.18**	44	**998.18**	998.82	**998.18**	1006.11	**998.18**	**998.18**	**998.18**	999.91	988.18
200-1	**910.17**	**910.17**	195	**910.17**	911.61	**910.17**	910.50	**910.17**	**910.17**	**910.17**	911.66	910.17
200-2	**921.76**	**921.76**	184	**921.76**	922.62	928.84	942.72	**921.76**	923.03	**921.76**	925.38	921.76
200-3	939.60	**939.61**	333	942.32	944.93	951.36	959.63	**939.58**	949.18	**939.58**	943.20	939.58
300-1	**977.65**	**977.65**	416	981.31	984.63	978.91	980.11	**977.65**	981.04	979.81	981.85	977.65
300-2	**913.01**	**913.01**	402	917.31	926.87	918.40	949.05	**913.01**	914.08	**913.01**	913.88	913.01
300-3	**974.78**	**974.78**	315	974.98	979.95	981.15	981.33	974.85	979.34	**974.78**	978.35	974.78
400-1	966.01	**966.03**	225	967.34	971.07	968.66	980.6	**965.99**	966.59	**965.99**	966.71	965.99
400-2	***934.17**	**937.88**	506	947.57	952.49	941.52	961.71	941.02	943.53	941.02	942.59	934.17
400-3	**1002.61**	**1002.67**	525	1003.24	1007.05	**1002.61**	1009.07	1002.97	1003.62	**1002.61**	1003.33	1002.61
500-1	**963.89**	965.91	272	967.32	975.25	986.49	991.85	**963.89**	**963.89**	**963.89**	964.80	963.89
500-2	**948.57**	**949.57**	457	954.89	965.45	953.77	996.85	**948.57**	952.96	948.96	950.12	948.57
500-3	**980.67**	**982.73**	553	992.3	1001.21	1006.23	1007.36	**980.67**	992.64	981.90	986.01	980.67
average	***945.94**	**946.45**	283	948.58	952.10	952.27	961.01	946.39	948.61	846.53	948.00	945.38

From Table 3, we observe that TLMH produces no worse result on most of the instances of transmission range 125m set than the others. It produces comparable results with EA/G-MP and ABC_DTP. Comparing to ABC_DT and ACO_DT, the results from TLMH are better for most of the instances in this set. TLMH updates the ever best objective value of instance range125/400-2 over other meta-heuristics. TLMH fails to produce the same good objective value as EA/G-MP and ABC_DTP on instances range125/200-3 and range125/400-1. But it produces better average results on these two instances comparing to the others. In terms of the overall average values, TLMH outperforms the other meta-heuristics while it is slightly inferior to the exact algorithm.

The results in Table 4 are similar with those in Table 3. We observe that TLMH produces comparable results with EA/G-MP and ABC_DTP, while it outperforms ABC_DT and ACO_DT on larger scaled instances. TLMH updates the ever best objective value for instance rang150/500-3. It fails on three instances, range150/50-3, range150/400-2, and range150/500-2, in terms of best objective value. For instance range150/400-2, TLMH produces a better average result than the others. In terms of the overall average values, TLMH outperforms the other meta-heuristics, while the round value is comparable to the exact algorithm.

Table 5 reports the summary information of the compared algorithms on each data set in *Range* benchmark. Columns BT (better) reports the number of instances that the corresponding algorithm produces better results than the others in the data set. Columns NW (no worse) reports the number of instances

Table 4: Computational results of TLMH and comparisons on Range-150

instance	TLMH			ABC_DT		ACO_DT		EA/G-MP		ABC_DTP		EXACT
	best	average	time	best	average	best	average	best	average	best	average	
50-1	**647.75**	**647.75**	1	**647.75**	**647.75**	**647.75**	**647.75**	**647.75**	**647.75**	**647.75**	**647.75**	647.75
50-2	**863.69**	**863.69**	2	**863.69**	**863.69**	**863.69**	**863.69**	**863.69**	**863.69**	**863.69**	864.04	863.69
50-3	**743.94**	**743.94**	2	**743.94**	**743.94**	**743.94**	**743.94**	**743.94**	**743.94**	**743.94**	745.68	743.94
100-1	**876.69**	876.79	297	**876.69**	876.85	881.37	885.36	**876.69**	**876.69**	**876.69**	877.02	876.69
100-2	**657.35**	**657.35**	11	**657.35**	**657.35**	**657.35**	**657.35**	**657.35**	657.53	**657.35**	657.53	657.35
100-3	**722.87**	**722.87**	2	**722.87**	**722.87**	**722.87**	**722.87**	**722.87**	**722.87**	**722.87**	**722.87**	722.87
200-1	**809.90**	**809.90**	138	**809.90**	**809.90**	**809.90**	810.87	**809.90**	810.49	**809.90**	**809.90**	809.90
200-2	**736.23**	**736.23**	354	**736.23**	736.27	**736.23**	**736.23**	**736.23**	**736.23**	**736.23**	**736.23**	736.23
200-3	**792.71**	**792.71**	97	792.73	797.00	**792.71**	793.73	**792.71**	795.65	**792.71**	793.48	792.71
300-1	**796.15**	**796.15**	283	796.70	797.94	796.70	797.17	**796.15**	798.12	796.29	796.99	796.15
300-2	**741.02**	**741.03**	298	**741.02**	743.20	748.94	752.33	**741.02**	743.05	**741.02**	742.88	741.02
300-3	**819.76**	**819.78**	129	**819.76**	823.76	826.48	826.56	**819.76**	821.67	**819.76**	820.45	819.76
400-1	**795.53**	**795.88**	445	796.70	801.57	796.70	798.24	**795.53**	798.82	**795.53**	797.92	795.53
400-2	779.67	**779.67**	388	781.20	782.28	782.91	787.66	**779.63**	783.14	**779.63**	781.40	779.63
400-3	**814.14**	**814.18**	388	816.53	822.64	826.48	831.32	**814.14**	817.38	**814.14**	815.35	814.14
500-1	**792.21**	**792.31**	357	796.50	800.25	794.47	797.13	**792.21**	793.59	793.98	796.16	792.21
500-2	779.38	**779.41**	274	**779.35**	785.10	**779.35**	791.20	**779.35**	781.28	**779.35**	780.04	779.35
500-3	***808.37**	**808.39**	281	809.65	811.08	808.50	811.35	808.50	810.27	808.50	808.50	808.37
average	**776.52**	**776.56**	208	777.14	779.08	778.69	780.82	**776.52**	777.90	776.53	777.46	776.52

that the corresponding algorithm produces no worse results comparing to the others in the data set. The best and average objective values are counted respectively in the table. From the information showed in Table 5, we observe that TLMH outperforms others by producing the most better and no worse results. In terms of the best objective value, TLMH produces better results in nine instances and no worse results in 48 ones, compared to all other meta-heuristics. In terms of the average objective value, TLMH produces 29 better results and 45 no worse results. We also observe that the advantage of TLMH mostly shows in range100 data set where the transmission range is $100m$. This indicates that TLMH performs better on sparse graphs than the existing meta-heuristics while it provides comparable performances on denser graphs.

Table 5: Summary on each data set in *Range*

criterion	data set	TLMH		ABC_DT		ACO_DT		EA/G-MP		ABC_DTP	
		BT	NW	BT	NW	BT	NW	BT	NW	BT	NW
best	range100	**7**	**17**	0	7	0	6	0	7	1	10
	range125	**1**	**16**	0	8	0	7	0	14	0	14
	range150	**1**	16	0	11	0	9	0	**17**	0	15
	total	**9**	**49**	0	26	0	22	0	38	1	39
average	range100	**12**	**16**	1	4	0	4	0	3	1	3
	range125	**10**	**16**	1	5	0	2	1	7	0	3
	range150	**10**	**17**	0	6	0	6	1	6	0	4
	total	**32**	**49**	2	15	0	12	2	16	1	10

4.5 Experiment on *RangeL* instances

From the previous experiments, we observe that large dense graphs are relatively difficult for our proposed TLMH algorithm. In this subsection, we generate *RangeL* benchmark to test its performance on a new dataset with large instances, providing a benchmark for future comparison. The *RangeL* dataset

can be obtained online[3]. We generated these instances such that vertices are deployed randomly in a 1000×1000 area, and each vertices pair have an edge if they are within a specific transmission range. The edge weight is defined as the Euclidean distance between the vertices. The number of vertices is in $\{1000, 1500, 3000\}$ and the transmission range is in $\{100, 300\}$. We generated three instances for each combination of the vertices number and the transmission range with different random seeds. Table 6 presents the computational results of TLMH on *RangeL*, where the last column shows the average vertices number of the MDT produced by TLMH. The last three instances (Range300-3000-1, Range300-3000-2, Range300-3000-3) seem rather difficult for TLMH, thus the time limit for these three is set to one hour. Table 6 shows that TLMH can solve the MDT problem with large instances. We also observe that the vertices number of the solution is much smaller compared to the whole vertices set (last column of Table 6). This is another proof of the observation mentioned in Section 2.2 that the dominating tree deduced from a smaller CDS is usually better.

Table 6: Computational results of TLMH on RangeL

instance	TLMH			
	best	average	time	X size
Range100-1000-1	5268.92	5429.09	757	85.5
Range100-1000-2	5417.50	5513.31	784	88.7
Range100-1000-3	5278.20	5551.54	753	88.8
Range100-1500-1	5354.81	5475.80	846	86.1
Range100-1500-2	5399.26	5531.28	759	88
Range100-1500-3	5459.77	5646.65	802	91.2
Range100-3000-1	5717.55	5875.34	934	93.2
Range100-3000-2	5676.88	5850.55	920	92
Range100-3000-3	5696.92	5824.12	900	90.1
Range300-1000-1	1631.03	1638.17	677	11.4
Range300-1000-2	1633.60	1636.24	618	11.9
Range300-1000-3	1599.42	1605.76	530	11.8
Range300-1500-1	1599.87	1609.74	643	11.6
Range300-1500-2	1674.58	1713.67	703	12.2
Range300-1500-3	1656.90	1680.26	744	11.7
Range300-3000-1	1980.85	2271.97	2231	10.4
Range300-3000-2	2047.46	2291.71	2181	10.3
Range300-3000-3	2121.76	2414.10	2162	11.5
average	3623.07	3753.29	997	50.36

5 Discussion and Analysis

In this section, we evaluate the primary strategies of TLMH, which are the sampling phase, the fast neighborhood evaluation method, and the perturbation operator. The experiments are performed on selected representative instances, but it is to be noted that similar results can be observed for other ones as well.

[3]https://github.com/xavierwoo/DTPSolver/tree/master/instances/RangeL (Uploaded and accessed on July 14th 2022)

To evaluate the merits of primary TLMH, we compare TLMH with its simplified versions obtained by

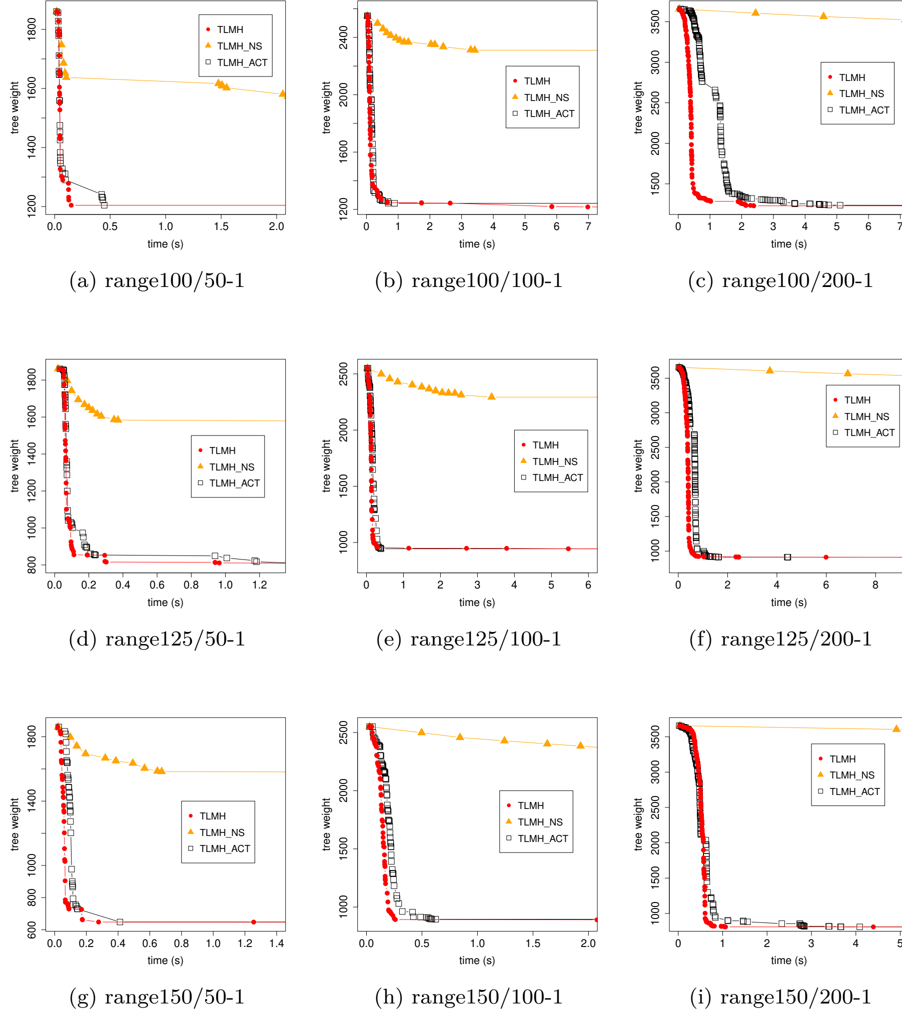
- Removing the sampling process (TLMH_NS) where the algorithm is started from the whole vertex set V.
- Deactivating the fast evaluation for f_2 (TLMH_ACT) that f_2 is calculated for each move evaluation.
- Disabling the perturbation (TLMH_NP)

To evaluate the impact of these ingredients on the searching process, we analyzed the evaluation of the best objective value found so far with the computational time for TLMH, TLMH_NS, and TLMH_ACT and present the results in Figure 4.

To evaluate the impact of the ingredients on the results, we compared the best and average objective values obtained by TLMH, TLMH_ACT, and TLMH_NP within 1000 seconds, of which the results are presented in Table 7. The results from TLMH_NS are far worse than the others. Thus, we do not put them on the table. Table 7 shows that TLMH outperforms the other two simplified versions on both best and average objective values obtained. The time differences are not obvious. However, although it is not always the case, TLMH_NP tends to use less time due to its defect in jumping out of local optimum traps. Thus, it stops improving the solution earlier.

Table 7: Comparison among TLMH, TLMH_ACT and TLMH_NP

instance	TLMH			TLMH_ACT			TLMH_NP		
	best	average	time	best	average	time	best	average	time
range100/50-1	**1204.41**	**1204.41**	1	**1204.41**	1205.811	2	**1204.41**	1204.84	11
range100/100-1	**1217.47**	**1217.47**		**1217.47**	**1217.47**	81	**1217.47**	1217.79	77
range100/200-1	**1206.79**	**1206.80**	515	**1206.79**	1207.23	544	**1206.79**	1218.39	110
range100/300-1	**1215.48**	**1217.40**	564	**1215.48**	1221.45	339	1216.57	1237.44	366
range100/400-1	**1211.33**	**1213.51**	502	1211.34	1219.31	734	1217.60	1234.23	357
range100/500-1	**1197.26**	**1202.81**	678	1200.06	1211.95	695	1200.06	1219.27	316
range125/50-1	**802.95**	**802.95**	1	**802.95**	806.08	2	**802.95**	809.28	1
range125/100-1	**943.01**	**943.01**	102	**943.01**	952.44	201	**943.01**	943.02	188
range125/200-1	**910.17**	**910.17**	195	**910.17**	917.73	83	**910.17**	911.63	254
range125/300-1	**977.65**	**977.65**	416	977.66	979.08	420	**977.65**	982.59	196
range125/400-1	**966.01**	**966.03**	225	**966.01**	972.88	246	**966.01**	972.28	94
range125/500-1	**963.89**	**965.91**	272	**963.89**	966.70	466	**963.89**	971.07	296
range150/50-1	**647.75**	**647.75**	1	**647.75**	647.84	1	**647.75**	647.84	1
range150/100-1	**876.69**	**876.79**	297	877.02	880.89	13	**876.69**	877.62	155
range150/200-1	**809.90**	**809.90**	138	**809.90**	814.73	8	**809.90**	811.34	102
range150/300-1	**796.15**	**796.15**	283	**796.15**	796.17	442	**796.15**	796.46	95
range150/400-1	**795.53**	**795.89**	445	**795.53**	795.91	310	795.67	797.99	269
range150/500-1	**792.21**	**792.31**	357	**792.21**	800.75	428	792.32	795.579	257
dtp/200/400-0.txt	**257.09**	257.23	371	**257.09**	257.52	478	**257.09**	**257.14**	370
dtp/200/600-0.txt	**121.62**	127.73	460	**121.62**	**122.80**	556	**121.62**	125.41	374
dtp/300/600-0.txt	**348.03**	351.22	530	348.60	**350.79**	530	**348.03**	350.89	380
dtp/300/1000-0.txt	148.63	**150.10**	630	149.70	152.00	388	**148.04**	150.46	498
RangeL100/1000-1.txt	**5268.92**	**5429.09**	758	5371.31	5562.123	636	5508.51	5676.687	334
RangeL100/1500-1.txt	**5354.81**	**5475.80**	847	5485.53	5664.658	834	5596.66	5752.75	790
RangeL100/3000-1.txt	**5717.55**	**5875.34**	934	5924.22	6100.743	1002	5717.55	5873.588	945
RangeL300/1000-1.txt	**1631.03**	1638.17	677	**1631.03**	**1637.125**	444	1634.86	1682.662	403
average	**1399.32**	**1417.37**	392	1416.41	1440.85	381	1418.36	1443.01	175



(a) range100/50-1 (b) range100/100-1 (c) range100/200-1

(d) range125/50-1 (e) range125/100-1 (f) range125/200-1

(g) range150/50-1 (h) range150/100-1 (i) range150/200-1

Figure 4: Computational performance comparison

5.1 Importance of sampling process

The sampling phase embedded in InitAndSample procedure is one of the most ingredients of TLMH. It roughly looks for dominating trees with different sizes of vertices set and provides an initial solution pool. The algorithm then starts searching from the solution pool offered by this sampling phase.

From Figure 4, one finds that algorithms with the sampling phase converge much faster than those without it. The difference is obvious from the very early stage. This indicates the sampling phase is a critical part of the proposed

TLMH.

The evaluation of f_2 is much slower than f_1. During the sampling phase, there is no evaluation of f_2. The algorithm picks any feasible CDS for each vertex set size and generates a dominating tree for the solution pool. Although this dominating tree may not be optimal or near-optimal for this vertex set size, it is usually not that unacceptable. More importantly, the sampling phase can be done very quickly. As discussed in Section 2.2, the MCDS does not usually produce the MDT. Thus, we do not need the sampling phase to dig too much for the MCDS. This makes the process even faster. Therefore, the result from the sampling phase can be an estimation up to scratch for the promising search space for the TLMH.

5.2 Efficacy of the fast evaluation

When evaluating a move operator, two terms need to be calculated in the objective function. For the evaluation of f_1, TLMH adopts the technique from the RNS-TS algorithm for the MCDS problem. This technique has been proven to be highly effective in the literature [2]. Here, we focus on the fast evaluation method implemented for f_2.

From Figure 4, we find that TLMH_ACT, the one without the fast evaluation for f_2, converges slightly slower than TLMH. This indicates that the fast evaluation for f_2 improves the performance to some extent. Although TLMH_ACT and TLMH seem to converge to the same final objective as shown in Figure 4, after comparing the results between TLMH_ACT and TLMH in Table 7, we find that the algorithm loses robustness to produce the same best objective if the fast evaluation method for f_2 is deactivated. TLMH_ACT produces worse average results than the full-featured TLMH.

The fast evaluation method for f_2 shows efficacy when the process is near the final best solution. This is reasonable that the near-optimal solutions usually contain fewer vertices. The algorithm takes a lot of effort to find a CDS with a limited number of vertices at this stage, i.e., most of the time f_2 does not matter for the evaluation. Thus, neglecting the f_2 alleviates the calculation a lot. For the early stage where the algorithm focus on larger connected vertices set, it is easier to find a CDS. Thus, the calculation for f_2 is needed more often. This is why we see from Figure 4 that TLMH and TLMH_ACT almost overlap with each other for the early stage. All these explain why TLMH_ACT can produce comparable best objective values as TLMH but with less stability.

5.3 Importance of perturbation

TLMH implements a random perturbation operator as a diversification mechanism. This perturbation operator applies several random moves of which the number depends on the current search state. If the search process keeps meeting the same local optimum configuration, TLMH strengthens the perturbation by performing more random moves. Thus, there is a larger chance for TLMH to jump out of this local optimum trap.

We observe from Table 7 that TLMH_NP can obtain the best result as the full-featured TLMH for some instances. However, the results from TLMH_NP are not stable. One finds that the average objective values obtained by TLMH_NP are much worse than TLMH and TLMH_ACT. The differences mainly show in the larger instances, where TLMH_NP fails to produce the same best result as TLMH. This analysis shows that the perturbation phase is critical for TLMH.

5.4 Statistical significance testing among versions of TLMH

We performed Two-sample T-Tests with unequal variance on the three versions of TLMH (The full-featured TLMH, TLMH_ACT, and TLMH_NP) on several representative instances from the three benchmarks to check if the differences in the results were caused merely by randomness. The tests are performed on each representative instance using the outputs from ten independent runs from each version of the TLMH. Table 8 presents the p values of the test.

Table 8: p values of T-Tests on each instance between versions of TLMH

instance	TLMH vs ACT	TLMH vs NP	ACT vs NP
Range100/ins-050-1	0.33	0.34	0.51
Range100/ins-100-1	1.00	0.11	0.11
Range100/ins-200-1	0.26	0.11	0.12
Range100/ins-300-1	0.05	0.00	0.02
Range100/ins-400-1	0.02	0.00	0.02
Range100/ins-500-1	0.01	0.01	0.23
Range125/ins-50-1	0.01	0.00	0.02
Range125/ins-100-1	0.22	0.08	0.22
Range125/ins-200-1	0.22	0.15	0.32
Range125/ins-300-1	0.03	0.03	0.11
Range125/ins-400-1	0.19	0.16	0.92
Range125/ins-500-1	0.76	0.24	0.30
Range150/ins-50-1	0.17	0.17	1.00
Range150/ins-100-1	0.02	0.02	0.06
Range150/ins-200-1	0.11	0.30	0.28
Range150/ins-300-1	0.34	0.02	0.03
Range150/ins-400-1	0.87	0.26	0.27
Range150/ins-500-1	0.09	0.19	0.32
DTP-200-400-0	0.04	0.19	0.01
DTP-200-600-0	0.31	0.69	0.49
DTP-300-600-0	0.88	0.93	0.97
DTP-300-1000-0	0.00	0.65	0.07
RangeL100-1000-1	0.01	0.00	0.02
RangeL100-1500-1	0.00	0.00	0.16
RangeL100-3000-1	0.00	0.97	0.00
RangeL300-1000-1	0.78	0.19	0.18

From Table 8, we can observe that some results from the full featured TLMH are significantly different ($p \leq 0.05$) from the other two versions, e.g., Range-100/ins-300-1, DTP-200-400-0, RangeL100-1500-1, etc.. For most of the instances, the results are similar but different. It indicates that the different versions of the proposed algorithm produce significantly different results for some instances, which is another proof of the criticalness of each component of the TLMH.

References

[1] T. Stützle and R. Ruiz. *Handbook of Heuristics*, chapter Iterated Local Search, pages 579–605. Springer International Publishing, 2018.

[2] X. Wu, Z. Lü, and P. Galinier. Restricted swap-based neighborhood search for the minimum connected dominating set problem. *Networks*, 69(2):222–236, 2017.

[3] F. Glover and M. Laguna. *Handbood of combinatorial optimization*, chapter Tabu search, pages 2093–2229. Springer US, 1999.

[4] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[5] Bernard A. Galler and Michael J. Fisher. An improved equivalence algorithm. *Commun. ACM*, 7(5):301–303, may 1964.

[6] R. C. Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, 1957.

[7] U. Benlic and J. Hao. Breakout local search for the quadratic assignment problem. *Applied Mathematics and Computation*, 219(9):4800–4815, 2013.

[8] U. Benlic and J. Hao. Breakout local search for the max-cut problem. *Engineering Applications of Artificial Intelligence*, 26(3):1162–1173, 2013.

[9] U. Benlic and J. Hao. Breakout local search for maximum clique problems. *Computers & Operations Research*, 40(1):192–206, 2013.

[10] B. Peng, D. Liu, Z. Lü, R. Martí, and J. Ding. Adaptive memory programming for the dynamic bipartite drawing problem. *Information Sciences*, 517:183 – 197, 2020.

[11] Z. Dražić, M. Čangalović, and V. Kovačević-Vujčić. A metaheuristic approach to the dominating tree problem. *Optimization Letters*, 11(6):1155–1167, 2017.

[12] S. Sundar and A. Singh. New heuristic approaches for the dominating tree problem. *Applied Soft Computing*, 13(12):4695–4703, 2013.

[13] E. Álvarez-Miranda, M. Luipersbeck, and M. Sinnl. An exact solution framework for the minimum cost dominating tree problem. *Optimization Letters*, 12(7):1669–1681, 2018.

[14] 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.

[15] S.N. Chaurasia and A. Singh. A hybrid heuristic for dominating tree problem. *Soft Computing*, 20(1):377–397, 2016.

[16] K. Singh and S. Sundar. Two new heuristics for the dominating tree problem. *Applied Intelligence*, 48(8):2247–2267, 2018.