Scatter Search for the Minimum Leaf Spanning Tree Problem

Yogita Singh Kardam, Kamal Srivastava*

Department of Mathematics, Dayalbagh Educational Institute, Agra, India

Pallavi Jain

Indian Institute of Technology Jodhpur, India

Rafael Martí

Department of Statistics and Operations Research, University of Valencia, Spain

Abstract

Given an undirected connected graph G, the Minimum Leaf Spanning Tree Problem (MLSTP) consists in finding a spanning tree T of G with minimum number of leaves. This is an NP-hard problem with applications in communications and water supply networks. In this paper, we propose a heuristic algorithm to provide high-quality solutions (spanning trees with low number of leaves) for an input graph. Our heuristic is based on the scatter search methodology, and it combines different elements to perform an efficient search of the solution space. In particular, it applies both randomized and deterministic strategies in the construction methods to generate an initial set of solutions. A combination method specifically designed for trees coupled with two local searches with a diversity evaluation function, provides a good balance between search intensification and diversification. Experiments conducted on a large set of graphs indicate that our algorithm is able to generate spanning trees with a lower number of leaves than previous methods. Additionally, it is able to match the optimal solution in

*Corresponding author.

Email address: yogita.singh27@gmail.com (Y.S. Kardam), kamal.sri@dei.ac.in (K. Srivastava), pallavi@iitj.ac.in (P. Jain), rafael.marti@uv.es (R. Martí) (Yogita Singh Kardam)

most of the instances for which it is known, outperforming the existing methods.

Keywords: Spanning Tree, Metaheuristics, Scatter Search

1. Introduction

Spanning tree problems constitute an important field in optimization due to their numerous applications in graphs and networks. Different objective functions have been defined to model a given problem, giving rise to a variety of spanning tree optimization problems, such as the well-known Minimum Weighted Spanning Tree Problem, in which we minimize the total edge weight in a spanning tree; the Minimum Diameter Spanning Tree Problem, where we minimize the maximum distance between the pairs of vertices in a spanning tree [1], or the Minimum Congestion Spanning Tree Problem, where we minimize the maximum congestion over all edges in a spanning tree [2], to name a few.

Some real world problems, especially in the area of networking, involve restrictions on the degree of vertices leading to degree based spanning tree problems. In this paper, we study the Minimum Leaf Spanning Tree Problem (MLSTP), which consists in minimizing the number of degree one vertices (leaves) in the spanning tree of a given undirected connected graph. This problem is a natural generalization of the well-studied Hamiltonian Path Problem, given the characterization that a graph has a Hamiltonian path if and only if it has a spanning tree with exactly two leaves [3]. On the other hand, MLSTP is the complement of the Maximum Internal Spanning Tree Problem (MISTP) that finds a spanning tree of an input graph with maximum number of internal (non-leaf) nodes. From an optimization point of view, MLSTP and MISTP are NP-hard [4] equivalent problems, though they are different from an approximation and parametrization point of view [5].

MLSTP has applications in communication [6] and water supply networks [7, 8], and it is relevant in designing cost effective optical networks [9]. In a communication network, the nodes (terminals) communicate through optical fibre cables through its spanning tree. The network nodes connected to more than two nodes require costly switch devices to be placed. Thus, to reduce the associated cost, a spanning tree of the network having minimum number of such nodes is needed. Network nodes and fibre cables between them are respectively mapped into vertices and edges of a graph G. Then, the number of switches to be placed on each branch vertex (vertex with degree greater than two) of the spanning tree ST depends on its degree. Hence, not only the number of branch vertices but also the degree of each branch vertex needs to be minimized since the exact number of switches needed on each branch vertex v in the spanning tree is its degree minus two.

The literature on the MLSTP is scarce. Several approaches have been proposed for the MISTP, including approximation algorithm, but little attention has been paid to the MLSTP, with the exception of a Memetic Algorithm (MA) designed to tackle three degree related spanning tree problems [9]. We will include MA in our computational experimentation.

In this paper, we propose a heuristic algorithm based on the Scatter Search methodology (SS) for the MLSTP. SS is a metaheuristic that operates on a small set of solutions, called the Reference Set (RS), by applying combination and improvement operators. RS collects and evolves the best solutions found so far (both in terms of quality and diversity) initially selected from a large population generated with a constructive method. One of its main characteristics is that their methods, namely solution generation, improvement, and combination, are based on problem dependent elements and strategies to perform an efficient search exploration. This is an important difference with other population based methods, such as the well-known Genetic Algorithms, mostly based on randomization.

In our SS algorithm for the MLSTP, we propose several construction heuristics to generate the initial population. Based on their performance, we consider a proportion of solutions to be constructed with each of these heuristics. We combine problem dependent strategies with random elements to construct solutions of relatively good quality and diversity. On the other hand, we propose two procedures to create the reference set, one having diversity as primary objective and the other one balancing elitism and diversity of solutions. These methods are coupled with a new combination method and two local improvement operators. The effectiveness of these operators is empirically evaluated in our computational testing.

As it is customary in heuristic papers, our experimentation is divided into two parts, scientific and competitive testing. The first set of experiments is designed to decide the strategy for generating initial population and reference set formation. After this, the best combination of these strategies is put together in the SS algorithm, and further experiments are performed on a large set of instances to compare its performance with the existing state of the art metaheuristic. The experiments have been conducted on four sets of public-domain graphs. The comparison shows that our algorithm outperforms the existing method for MLSTP. Additionally, we believe that the SS designs proposed here can be extended to other settings, thus providing valuable lessons to the researchers interested on this methodology.

The paper is organized as follows. In this introduction we briefly describe the problem and our contributions; then, in Section 2 we provide background to the readers by introducing mathematical notation and previous work on this problem. Section 3 is the core of the paper, where we describe our SS algorithm. As it is customary in heuristic papers, this is followed by a section with experimental results. Section 4 details our extensive experimentation, and the paper finishes in the following section with the associated conclusions.

2. Background

In this section, we first introduce some basic definitions and notations to model the problem in mathematical terms, and then describe the previous method proposed for this problem.

2.1. Preliminaries and Definitions

Let $G = (V, E)$ be an undirected graph, where $V(G) = \{v_1, v_2, \ldots, v_n\}$ is the set of vertices, and $E(G) = \{(u, v) : u, v \in V(G)\}$ is the set of edges

$(E(G) \subset V(G) \times V(G))$. The size of graph G is defined as $|V(G)| = n$. The set of adjacent vertices (neighbours) of a vertex $v \in V(G)$ is defined as $N_G(v) = \{u \in V(G) : (u, v) \in E(G)\}$, and its degree as $deg_G(v) = |N_G(v)|$.

Given a graph $G = (V, E)$, a spanning tree ST is a subgraph with set of vertices $V(ST)$ and edges $E(ST)$ that is a tree and includes all the vertices in G (i.e., $V(ST) = V(G)$). In a partial spanning tree PT, $V(PT) \subseteq V(G)$.

A vertex $v \in V(G)$ is a leaf in ST if $deg_{ST}(v) = 1$, and it is a branch vertex if $deg_{ST}(v) > 2$. Non-leaf vertices are called internal vertices. An edge $(u, v) \in E(G) \backslash E(ST)$ is a chord for the spanning tree ST.

Let $\phi(G)$ be the set of all spanning trees of graph G. The Minimum Leaf Spanning Tree Problem (MLSTP) can be simply stated as finding the spanning tree in $\phi(G)$ with minimum number of leaves. To formulate it in mathematical terms, we define $nLeaf(ST)$ as the number of leaves of spanning tree ST ($nLeaf(ST) = |\{v \in V(ST) : deg_{ST}(v) = 1\}|$). Then, the MLSTP can be formulated as:

$$\min_{\forall ST \in \phi(G)} \{nLeaf(ST)\}.$$

Throughout this paper, a solution to the problem refers to a spanning tree ST of the input graph G, and its *cost* refers to the number of leaves in it, $nLeaf(ST)$. The optimal solution ST^* is then the spanning tree with minimum $nLeaf$-value:

$$ST^* = argmin_{ST \in \Phi(G)}\ nLeaf(ST).$$

2.2. Related Work

As mentioned, the Minimum Leaf Spanning Tree Problem (MLSTP) is equivalent in optimization terms to the Maximum Internal Spanning Tree Problem (MISTP). In this section we comment on both problems.

Lu and Ravi [10] proved that the MLSTP has no constant approximation factor, unless $P = NP$, while the MISTP has several constant factor approximation algorithms. As a matter of fact, parametric [11, 12] and weighted versions [13, 14] of MISTP have been proposed in literature, together with exact exponential algorithms [8]. For a detailed survey on MISTP we refer the reader to the excellent work by Salamon [5].

A simplified and faster version of Salamon's algorithm for weighted and un-weighted MISTP with better approximation ratio has been given in [6] on general graphs. Later on, a 4/3-approximation algorithm for MISTP was devised, which is an improvement over the Salamon's algorithm [15]. In [16], it is proved that MISTP can be solved in polynomial time on interval graphs. For this, an $O(n^2)$ algorithm is presented on these graphs, which finds a spanning tree with number of internal vertices equal to the number of edges in a maximum path cover of the graph minus one. Improved versions of parameterized algorithm (with kernel of size 2k vertices) and approximation algorithm (with approximation ratio 1.5) for MISTP are proposed in [17] using deeper local search. This approximation ratio is further reduced to 13/17 in [18] by developing an algorithm which explores much deeper structure of MISTP as compared to the previous algorithms. However, in spite of this abundant literature on approximation methods, no metaheuristic has been proposed for this problem.

Chen et al. [19] recently proposed an algorithm based on a novel relationship between maximum weight internal spanning tree and maximum weight matching for the weighted version of MISTP. The authors proved that their method has an approximation ratio of 1/2. Additionally, a 7/12-approximation algorithm is also designed for claw-free graphs in [19]. In [20], the parameterized complexity of MLSTP for independency and cliquy trees has been characterised. Parameterized algorithms for MISTP are well summarized in the survey paper [21]. In [22], some bounds on the minimum leaf count for the spanning trees of connected cubic graphs and 2-connected graphs are proved which are improvements over the previous best known bounds ([23, 24]). Two algorithms have been proposed for maximum weight internal spanning tree problem which improve the existing approximation factors for cubic and claw-free graphs of degree at least three [25]. The most recent work on MISTP includes an approximation algorithm [26] with a performance ratio of 4/3.

2.3. A Memetic Algorithm for the MLSTP

In our revision of the literature for MLSTP, we only found a metaheuristic algorithm. The memetic algorithm (MA) by Cerrone et al. [9] is essentially a genetic algorithm encompassing the usual features of tournament selection, binary crossover, and mutation, applied iteratively and coupled with a local improvement. MA starts with a randomly generated population Pop of p_size solutions (spanning trees of the input graph G), and then performs consecutive iterations as briefly described below.

The first step of the iterations is the application of the **crossover operator** to a pair of solutions randomly selected from the population as follows. A set with x solutions randomly selected from Pop is formed, and the best solution of this set becomes the first parent P_1. Now $x/2$ solutions are randomly selected from $Pop\backslash\{P_1\}$ to form another set, and the best member of this set becomes the second parent P_2. The rationale behind this selection is to involve at least one “good" quality parent in the crossover procedure. Unlike classical crossover operators, this crossover operator is not a recombination of parent solutions. It tries to construct a child C that inherits “good" characteristics from its parents. This is accomplished by assigning weights to each edge of the input graph G such that the edges belonging to a path (vertices with degree less than 3 in the chains remaining after removing all the edges incident on branch vertices) in a parent solution are promoted, whereas the edges incident on the internal vertices of these paths (connected to chains) are penalized since they create new branch vertices. Edges of G which are incident on the branch vertices in a parent are also promoted. Promotion of an edge is accomplished by assigning negative weights, whereas positive weights indicate penalties. The union of these weighted trees results in an edge weighted graph G_w, which contains the edges from the parents P_1 and P_2 and the weight on each edge is the sum of the weights assigned to the tree edges. It is to be mentioned here that initially each edge is assigned zero weight. The offspring is obtained by finding a minimum weight spanning tree of G_w.

Mutation is now performed on the solution C obtained with crossover to obtain a new solution C'. MA implements two mutation operators M_1 and M_2. In M_1, a leaf l in C is randomly selected and it is connected to a randomly selected vertex v through the edge (l, v) of G. Now the resulting cycle is randomly broken to generate a new spanning tree C'. M_2 on the other hand, tries to eliminate branch vertices. To do it, edges are randomly selected and deleted from C for a fixed number of iterations, resulting in three components connected by two edges, which are randomly selected to obtain a new tree. In the mutation phase of the algorithm, M_1 is always executed on C while M_2 is executed after M_1 with some probability.

The iterative phase of MA finishes with the application of a **local search** to the solution C' previously obtained. In this phase, one edge of the current solution is switched with a new one, but instead of selecting edges randomly (as mutation M_1 does), the edges are switched in a way that leads to an improvement of the original tree. In this method, the introduction of new nodes of degree at most two is promoted and the introduction of new branch vertices is penalized. The local search performs iterations as long as an improvement move is found in the neighborhood of the current solution. The resulting solution C^* is inserted into Pop, replacing the worst solution from a set of y randomly selected solutions from Pop. The algorithm terminates when no improvement is observed for Max_iter consecutive iterations.

It is worth mentioning that the implementation proposed in [9] involves a large number of parameters: four in the main block and five in the weight functions of crossover and local search phases, which need to be tuned in order to have an efficient performance.

3. Scatter Search for the MLSTP

Scatter Search (SS) is a population based metaheuristic which was first introduced in 1977 by Fred Glover [27]. It is one of the widely applied metaheuristic methodologies for finding high-quality solutions to NP-hard combinatorial optimization problems [28, 29, 30, 31].

SS starts by generating an initial population of solutions and improving them using a local search method. Next, a reference set (*RefSet*) is created to extract a diverse set from these improved solutions. Now, a collection of subsets of reference set is built. The members of each subset combine to produce new solutions, which are in turn submitted to a local improvement phase. The reference set is continuously updated using these improved solutions. The process is repeated until no further improvement is observed in the reference set updating. The initialization of our SS algorithm for the MLSTP is outlined in Algorithm 1, and the iterative procedure SS_Iter is given in Algorithm 2.

Algorithm 1 Scatter Search initialization

1: Initialize (pop_size), $RefSet$ size (rs), and (max_iter)
2: $pop \leftarrow$ Generate initial solutions
3: $pop' \leftarrow$ IMPROVEMENT1(pop)
4: $pop'' \leftarrow$ IMPROVEMENT2(pop')
5: $S_{best} \leftarrow$ Best solution of pop''
6: $RefSet \leftarrow$ Generate RefSet from pop''
7: $Init\_Div \leftarrow Div\_eval(RefSet)$
8: $Algorithm2(max\_iter, Ref\_Set, S_{best}, Init\_Div)$

Algorithm 1 starts by generating an initial population *pop* consisting of pop_size solutions (Line 2). To accomplish this, we apply four construction heuristics described in Section 3.1. In Lines 3 and 4, the solutions of this population are sequentially submitted to two local improvement methods, namely IMPROVEMENT1 and IMPROVEMENT2, resulting in an improved set of solutions pop'' (discussed in Section 3.3). The best solution obtained so far is stored in S_{best} (Line 5). Now, $rs = |RefSet|$ solutions are selected from pop'' based on a certain criterion to populate the new *RefSet* (Line 6). We have designed and implemented two procedures for building the *RefSet*, which are explained in detail in Section 3.4. The diversity of the *RefSet* is measured with the Div_eval function detailed out in Section 3.5, and the method finishes by calling Algorithm 2 (Line 8) that performs the scatter search iterations.

Algorithm 2 iterates over the *RefSet* to improve its solutions. It takes as the input the initial *RefSet* generated with the initialization, and returns as the output the best solution in the final *RefSet*. In particular, it first creates its $rs(rs-1)/2$ subsets of pairs of solutions (Line 4), denoted as $Ref\_Subsets$. Now, solutions within each subset are submitted to the combination method, *Combine* (Line 6), described in Section 3.2. The combination of a pair of solutions $(C_1, C_2) \in Ref\_Subsets$ results in two new solutions, S_1 and S_2, which are further improved in Lines 8 and 9. The best of these two improved solutions S'' (Line 11) is now used to update the *RefSet*. The solution S'' replaces the worst solution (S_{worst}) in the *RefSet* if it is better than it.

The best solution in the *RefSet*, S_{best} is updated whenever any new incumbent solution improves it. The diversity of the updated reference set is again computed (Line 22). As maintaining diversity of the reference set is a key strategy of the method, if it decreases in a 50% of its initial value, then $rs/2$ new solutions are randomly generated (Line 24), with a method described in Section 3.1. The new solutions replace the worst ones in *RefSet*, thus increasing its diversity without deteriorating its quality. The algorithm terminates after a specified number of iterations with no improvement.

3.1. Construction Heuristics

Construction heuristics play an important role in population based metaheuristics. Quality and diversity are recommended to create a good set of initial solutions. On one hand, we need relatively good solutions, but it is also true that without a certain level of diversity, evolutionary methods easily get trapped in sub-optimal solutions. To achieve a good balance between quality and diversity in the initial population that results in an efficient exploration of the search space, we consider five construction heuristics, avoiding in this way a premature convergence, and at the same time, keeping the running time relatively low.

- **H1**. In this constructive method, spanning trees are constructed using the well-known DFS procedure. The selection of the vertex to be visited at each step is based on the degree of the vertices, where priority is given to

Algorithm 2 Scatter Search iterations

$iter \leftarrow 1$
while $iter \leq max\_iter$ **do**
  $Update \leftarrow 0$
  $Ref\_Subsets \leftarrow$ Generate subsets of $RefSet$
  for each $(C_1, C_2) \in Ref\_Subsets$ **do**
    $[S_1, S_2] \leftarrow Combine(C_1, C_2)$
    for $i \leftarrow 1$ to 2 **do**
      $S'_i \leftarrow$ IMPROVEMENT1(S_i)
      $S''_i \leftarrow$ IMPROVEMENT2(S'_i)
    end for
    $S'' \leftarrow$ Best of $S''_1,\ S''_2$
    $S_{worst} \leftarrow$ Worst solution of $RefSet$
    if $S'' \notin Ref\_Set$ and $cost(S'') < cost(S_{worst})$ **then**
      $RefSet \leftarrow (RefSet \backslash \{S_{worst}\}) \cup S''$
      $Update \leftarrow 1$
    end if
    if $cost(S'') < cost(S_{best})$ **then**
      $S_{best} \leftarrow S''$
      $iter \leftarrow 0$
    end if
  end for
  $Final\_Div \leftarrow Div\_eval(RefSet)$
  if $Final\_Div < Init\_Div/2$ **then**
    $new\_sol \leftarrow$ Generate new solutions with H_5
    $RefSet \leftarrow$ Modify $RefSet$ with new_sol
  end if
  if $Update == 0$ or $iter \neq 0$ **then**
    $iter \leftarrow iter + 1$
  end if
end while
return S_{best}

lower degree vertices. This may lead to a spanning tree with more depth and may result in fewer leaves. Thus, minimum degree vertex is chosen as the root vertex, and ties are randomly broken. From any current vertex the search proceeds towards an adjacent unvisited vertex with minimum degree.

- **H2**. This heuristic constructs a spanning tree of a given graph using the well known Prim's algorithm. Initially, a random vertex u is placed in U, then a neighbor v of this vertex from V is selected randomly and added to U. The remaining vertices of V which are adjacent to the vertices of U are added in a similar fashion until all the vertices of V are placed in U. The set of these edges forms a spanning tree.

- **H3**. This method is inspired by Kruskal's algorithm that forms a spanning tree of a graph based on the edges weights. Initially, an empty set of edges is taken, then an edge with the smallest weight in the input graph is added to this set iteratively as long as its addition produces no cycles. However, since in our case the underlying graph is unweighted, the selection of edges is completely random. This process is iterated until $|V| - 1$ edges have been added.

- **H4**. This heuristic implements the Dijkstra's algorithm by considering unit weight on each edge. The vertices from set V are added to the set U (initially empty) on the basis of their distance from a fixed vertex u randomly chosen. At every step of the algorithm, a vertex not included in U and having minimum distance from u is obtained and added to U. The whole process is repeated until all vertices of V are included in U. Note that the output of Dijkstra's algorithm is an arborescence (i.e., a directed tree with a root), so we transform the solution in a tree by considering the arcs of the solution as undirected edges, and ignoring the root.

- **H5**. This heuristic generates the spanning tree by applying a Depth First Search algorithm. In this, all the vertices of the input graph are traversed moving from one vertex to its un-visited adjacent vertex. At each step of the heuristic, vertices are randomly selected. Therefore this is a completely random constructive method.

3.2. Combination Method

Producing new solutions by combining two or more existing solutions is an effective method to explore the search space. We have developed a new combination method, *Combine*, (outlined in Algorithm 3) that takes two spanning trees ST_1, ST_2 as the input, and produces two new solutions by combining them. Specifically, two vertices are initially selected at random, say v_i and v_j. Then, two partial spanning trees PT_1 and PT_2 are created by finding the path from v_i *to* v_j in ST_1 and ST_2 respectively. Now, the partial tree PT_1 is transformed into a complete spanning tree ST_1' by adding the remaining edges sequentially (following the canonical order) from ST_2. In the same manner, ST_2' is formed using PT_2 and ST_1. ST_1' and ST_2' are referred as the children trees. The process of adding the remaining edges from the spanning tree ST to the partial tree PT is given in the procedure *Generate_Child* (see Algorithm 4).

To illustrate the combination method, consider the graph in Figure 1(a) where $v_1 = 5$ and $v_2 = 4$. Let PT_1 and PT_2 be partial trees (see Figure 1(c)) obtained from the spanning trees ST_1 and ST_2 shown in Figure 1(b). PT_1 and PT_2 are the paths from 5 to 4 in the spanning trees ST_1 and ST_2 respectively. Here, $PT_1 = \{(5,3),(3,1),(1,4)\}$ and $PT_2 = \{(5,2),(2,1),(1,7),(7,4)\}$.

The child tree ST_1' of Fig. 2(a) is obtained from the partial vertex set $PV = \{5,3,1,4\}$ and $V'(G) = V(G)\backslash PV = \{2,6,7\}$. First consider vertex $2 \in V'(G)$ since it is the first one in the canonical order, $N_{ST_2}(2) = \{1,5\}$, $deg_{PT_1}(1) = 2$ and $deg_{PT_1}(5) = 1$. Since vertex 5 has the least degree, vertex 2 is added to PV through the edge $(2,5)$ and hence $PV = \{5,3,1,4,2\}$ and $PT_1 = \{(5,3),(3,1),(1,4),(2,5)\}$. We next consider vertex 6, which has neighbor 3 in PV. So, 6 is added to PV through edge $(3,6)$. Thus, $PV = \{5,3,1,4,2,6\}$ and $PT_1 = \{(5,3),(3,1),(1,4),(2,5),(3,6)\}$. Similarly, vertex 7 is added to PV and the complete spanning tree is $ST_1' = \{(5,3),(3,1),(1,4),(2,5),(3,6),(4,7)\}$.

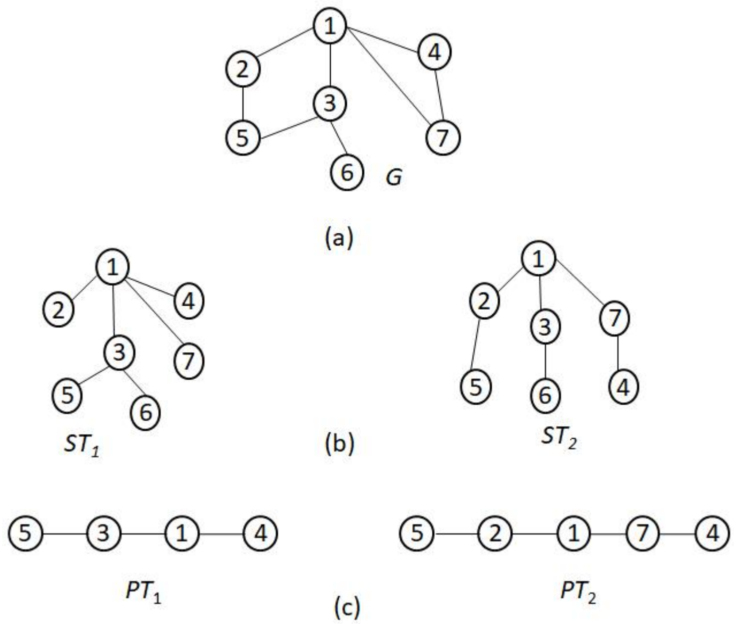

Figure 1: (a) A graph G, (b) spanning trees ST_1 and ST_2 of the graph G, (c) partial trees PT_1 and PT_2 obtained from spanning trees ST_1 and ST_2 respectively.

Similarly, the other child tree ST_2' is obtained as shown in Fig. 2(b). In Fig. 2, edges added to the paths PT_1 and PT_2 are shown by dotted lines.

It is important to mention here that choosing minimum degree vertex in the partial spanning tree while a new vertex is being added to it helps in keeping the leaf count low.

Time Complexity: In Lines 2 and 3 of Algorithm 3, the path is found using a DFS algorithm, which takes $O(|V|)$ time. In Lines 4 and 5, Algorithm 4 *Generate_Child* is called. In our implementation, we have used a 2 $\times$ (n -1) matrix to store the edges of spanning tree. Since the size of PT can be at most $|V|$-1, Line 1 of this algorithm takes $O(|V|)$ time. The time required by Lines 2, 4 and 7 is $O(|V|)$. Lines 5 and 6 take $O(1)$ time. Lines 4-7 are performed $O(|V|)$ times. Therefore, time required by this algorithm is $O(|V|^2)$.

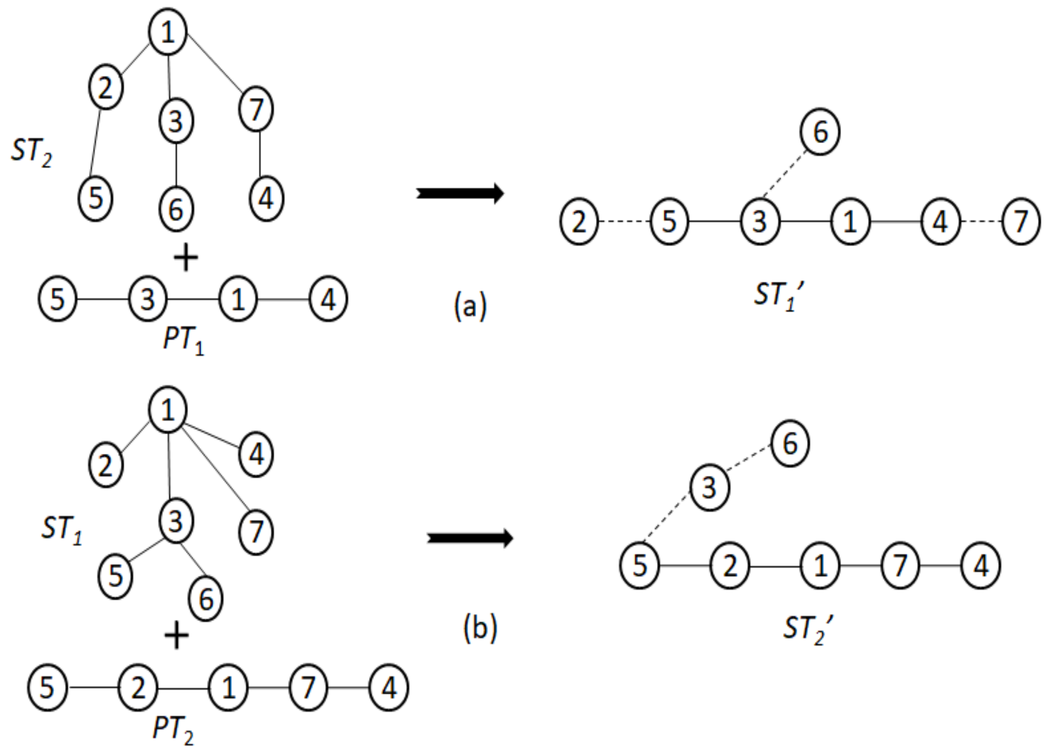

Figure 2: Child spanning trees (a) ST_1' obtained from ST_2 and PT_1 and (b) ST_2' obtained from ST_1 and PT_2

Algorithm 3 $Combine(ST_1, ST_2)$

Generate two random numbers $i, j \in \{1, 2, \ldots, n\}$
$PT_1 \leftarrow$ path in ST_1 from v_i to v_j
$PT_2 \leftarrow$ path in ST_2 from v_i to v_j
$ST_1' \leftarrow Generate\_Child(PT_1, ST_2)$
$ST_2' \leftarrow Generate\_Child(PT_2, ST_1)$
return $ST_1',\ ST_2'$

Algorithm 4 $Generate\_Child(PT, ST)$

$PV \leftarrow$ vertices in PT
$V'(G) \leftarrow V(G) \backslash PV$
for $v \in V'(G)$ such that $PV \cap N_{ST}(v) \neq \emptyset$ **do**
    Find $u \in PV \cap N_{ST}(v)$ with the lowest degree in PT
    $PT \leftarrow PT \cup (u, v)$
    $PV \leftarrow PV \cup \{v\}$
    $V'(G) \leftarrow V'(G) \backslash \{v\}$
end for
$Child\_Tree \leftarrow PT$
return $Child\_Tree$

3.3. Local Improvement Methods

We propose two local improvement operators that are applied successively to a solution in order to improve its quality. The rational behind both methods is to reduce the leaf count by making leaves as internal vertices and, at the same time, decreasing the degree of branch vertices. This is accomplished by suitable cycle exchanges. Since both are iterative procedures, a number of new solutions are generated during the process. Thus, the search is further enhanced through these exploratory procedures.

3.3.1. IMPROVEMENT1

This operator reduces the leaf count, $nLeaf$-value, by first adding a chord (l_i, l_j), where l_i and l_j are leaves in the spanning tree, and then removing a suitable edge from the tree. Thus, for a given spanning tree, ST, the procedure finds a pair of leaves l_i and l_j such that $(l_i, l_j) \in E(G)$ and there exists v such that $(v, l_i), (v, l_j) \in E(ST)$. Now, the edge (l_i, l_j) is added to, and the edge (v, l_i) is removed from $E(ST)$. This reduces the leaf count by one as the removal of (v, l_i) and the addition of (l_i, l_j) make l_j as an internal vertex. The process is repeated until no such pairs of vertices are present in the ST (see Algorithm 5).

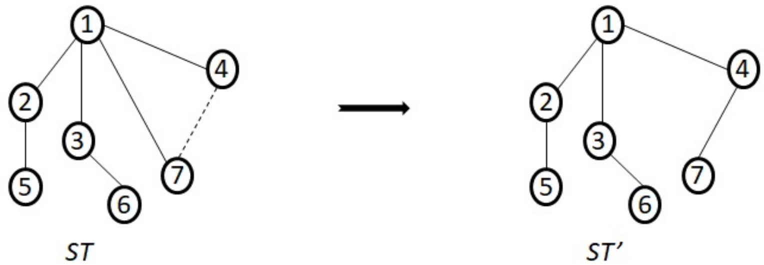

Figure 3: Applying Improvement1 on ST.

In Fig. 3, leaves 4 and 7 are connected to vertex 1 in ST of the given graph G (figure 1(a)). Here, the dashed line shows that 4 and 7 are neighbors in the graph. The addition of edge $(4, 7)$ and removal of edge $(1, 7)$ in $E(ST)$ results in the reduction of one leaf in ST'.

Algorithm 5 Improvement1 (ST)

1: $leaves \leftarrow \{v \in V(G) : deg_{ST}(v){=}1\}$
2: **for all** $l_i, l_j \in leaves$ such that $(\mathrm{l}_i, l_j) \in E(G)$ and $N_{ST}(l_i) == N_{ST}(l_j)$ **do**
3: $\quad E(ST) \leftarrow (E(ST) \cup (l_i, l_j)) \backslash (N_{ST}(l_i), l_i)$
4: $\quad leaves \leftarrow leaves \backslash \{l_j\}$
5: **end for**
6: **return** ST

Time Complexity: The degree of vertices in a ST can be found in $O(|V|$-1$)$ time, therefore Line 1 takes $O(|V|)$ time. In Line 3, removal and addition of edges to the spanning tree takes $O(|V| - 1)$ time. If the number of leaves in ST is l, then Line 4 is executed in $O(l)$ time. Lines 3 and 4 are repeated l times as each time the number of leaves is reduced by one (in Line 2). Since l is always less than $|V|$, therefore the time complexity of this method is $O(|V|^2)$.

3.3.2. Improvement2

This operator works by first adding a chord (v, l), where l is a leaf, to form a cycle in ST, and then removing an edge from the cycle to reduce the degree of branch vertices in the spanning tree, which helps to reduce the leaf count, as described in the following cases below. It is an iterative method where in each iteration the neighbors of a leaf in the given graph are searched. Then, an edge from one of the neighbors to that leaf is added in the ST, which creates a cycle. Based on the number of branch vertices present in the newly created cycle, the following cases arise:

Case 1: If there are at least two branch vertices which are connected by an edge in the cycle, then this edge is removed from the ST resulting in reduction in the degree of branch vertices and subsequently in the number of leaves. Figure 4(b) shows an improvement in a given spanning tree ST of the graph G' (Fig. 4(a)) using this method. Solid lines in the figure indicate tree edges whereas the dotted one represents the edge which will be added to form a cycle. Here, 1 and 5 are two branch vertices in a cycle created by joining the leaf (vertex 3) of the spanning tree with one of its neighbors (vertex 6) in the graph. Now, removing the edge $(1, 5)$ and adding $(3, 6)$ results in number of leaves being reduced from 4 to 2.

Case 2: If there is at least one branch vertex in the cycle, then the edge from the branch vertex to one of its neighbor in that cycle is removed. This process may lead to an improvement of ST depending on the degree of the neighboring vertex. Figures 4(c) and 4(d) illustrate this case, having vertex 1 as the only branch vertex in the cycle created after the addition of edge $(4, 1)$ and $(4, 5)$ to ST in these figures respectively. The spanning tree ST in Fig. 4(c) gives no improvement after removal of edge $(1, 2)$ as the process ends up with vertex 2 as a leaf since its degree is decremented by one, while ST in figure 4(d) results into an improvement when the edge $(1, 5)$ is removed, as degree of vertex 5 remains the same as the added edge has 5 as one of its end points.

Case 3: If there is no branch vertex in the cycle, then the process is repeated for another leaf in the tree (see Algorithm 6).

Algorithm 6 IMPROVEMENT2 (ST)

1: $leaves \leftarrow \{v \in V(G) : deg_{ST}(v){=}1\}$
2: **for all** $l \in leaves$ **do**
3:     $nbrs \leftarrow N_G(l)$
4:     mark all $v \in nbrs$ as unvisited
5:     $flag \leftarrow 0$
6:     **while** $flag{==}0$ and $\exists$ a vertex $v \in nbrs$ which is unvisited **do**
7:         Choose an unvisited $v \in nbrs$ and mark it as visited
8:         **if** $(v, l) \notin E(ST)$ **then**
9:             $C \leftarrow$ Cycle obtained by adding edge (v, l) to ST
10:             $br \leftarrow \{v \in V(C) : deg_{ST}(v) > 2\}$
11:             $br' \leftarrow V(C) \backslash br$
12:             **if** $\exists\ b_i, b_j \in br$ s.t. $(b_i, b_j) \in E(C)$ **then**
13:                 $E(ST) \leftarrow (E(ST) \cup (v, l)) \backslash (b_i, b_j)$
14:                 $flag \leftarrow 1$
15:             **else if** $\exists\ p \in br, q \in br'$ s.t. $(p, q) \in E(C)$ **then**
16:                 $E(ST) \leftarrow (E(ST) \cup (v, l))\ \backslash (p, q)$
17:                 $flag \leftarrow 1$
18:             **end if**
19:         **end if**
20:     **end while**
21: **end for**
22: **return** ST

Time Complexity: Finding leaves in Line 1 takes $O(|V|)$ time. As a vertex can have maximum $|V|$-1 number of neighbors, Line 3 takes $O(|V|)$ time. Line 4 takes $O(\Delta(G))$ time, where $\Delta(G)$ is maximum degree of a vertex in graph G. The time required by each Line 7-19 (except the Lines 14 and 17) is $O(|V|)$. Lines 7-19 are repeated $\Delta(G)$ number of times and Lines 3-20 are performed for

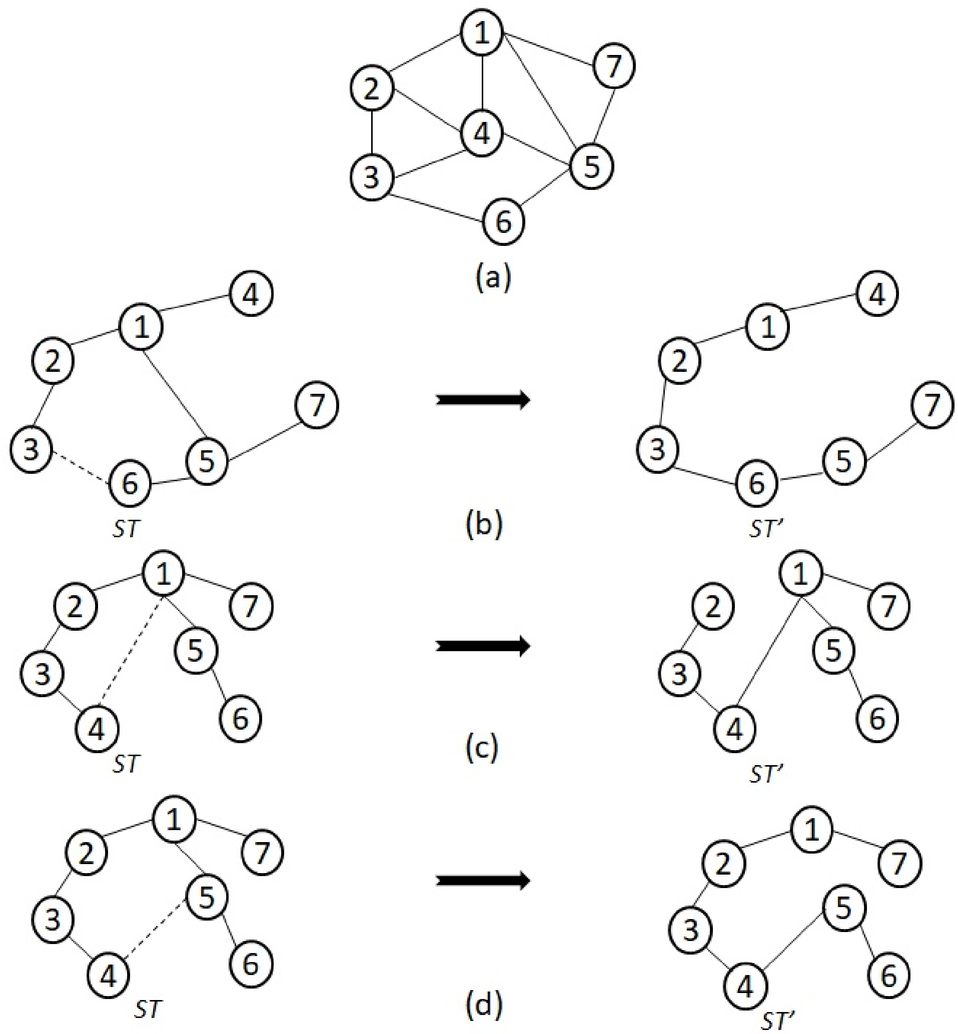

Figure 4: Applying IMPROVEMENT2 on ST of (a) a graph G' (b) when there are two branch vertices in cycle, (c) and (d) when there is one branch vertex in cycle.

all leaves in the ST, so a maximum $|V|-1$ number of times. Therefore, time complexity of this method is $O(|V|*(|V|*\Delta(G)))$.

3.4. Reference Set Formation

Considering that in SS a small fraction of the population, called the reference set, takes part in the evolution process; the selection of the solutions that form this set significantly influences the search process. In other words, the creation of the reference set is a key component of the algorithm. We have devised two methods for generating it.

3.4.1. Method 1 (RM_1)

This method for reference set initialization is inspired in [29]. After creating and improving the initial solutions to obtain population pop'',the Ref_Set is created from this population by considering both the quality and diversity of solutions. To achieve quality, the best $rs/2$ solutions are selected from the population. This operation can be performed in $min(O(pop\_size*(rs/2)), O(pop\_size*log(pop\_size)))$ time. The remaining $rs/2$ solutions are selected according to a distance function designed for MLSTP to maintain the diversity in the Ref_Set.

The distance between a spanning tree ST and the Ref_Set is defined as:

$$d(ST, Ref\_Set) = \min_{ST'\in Ref\_Set} Count(ST, ST')$$

where,

$$\begin{aligned} Count(ST, ST') = &|\{(u,v)\in E(ST') : (u,v)\notin E(ST)\}\cup \\ &\{(u,v)\in E(ST) : (u,v)\notin E(ST')\}| \end{aligned}$$

Clearly, $Count(ST, ST')$ gives the number of edges which are not common in ST and ST', and this computation can be performed in $O(|V|*log|V|)$ time (by sorting one of them in lexicographical order and searching edges of the other in this sorted tree using Binary search). For each $ST \in pop''\backslash Ref\_Set$, we compute $d(ST, Ref\_Set)$. It takes $O(|V|*log|V|(pop\_size-rs/2)rs/2)$

time. Since solutions which are distant from Ref_Set are preferred for maintaining the diversity, the best $rs/2$ solutions (solutions which maximize the minimum distance from Ref_Set) are added to the Ref_Set. Note that it takes $O((pop\_size - rs/2)rs/2)$ time. Hence, reference set can be constructed in $O(|V| * log|V|(pop\_size - rs/2)rs/2)$ time.

3.4.2. Method 2 (RM_2)

This method is based on the clustering algorithm of NSGA-II [32]. Initially, pop_size number of clusters are formed by simply putting each solution of the population into a separate cluster. Then, for each pair of clusters C_i and C_j, where $1 \leq i, j \leq pop\_size$ and $i \neq j$ their distance cd_{ij} is computed as:

$$cd_{ij} = \frac{1}{|C_i||C_j|} \sum_{ST \in C_i, ST' \in C_j} Count(ST, ST')$$

After computing the distance for all the pairs of clusters, the pair with minimum distance is merged (i.e., their clusters are combined into a single one). This reduces the number of clusters by one. The distances are again calculated with pop_size-1 clusters, and the process is repeated until the number of clusters is rs. Then, a solution is selected from each cluster as a member of reference set. In particular, the solution with minimum average distance to the other solutions in that cluster is the one chosen. The time complexity of this method is $O(pop\_size^2(rs + |V| * log|V|))$ [32].

3.5. Diversity Evaluation Function

As mentioned above, the convergence of the SS algorithm mainly depends on the diversity of the reference set. To maintain it during the search process, we consider a diversity evaluation function, Div_eval, that calculates the diversity of reference set after every iteration.

To compute the diversity evaluation we first compute the average diversity avg_div of reference set over the $rs(rs-1)/2$ number of possible solution pairs. In mathematical terms:

$$avg\_div = \frac{\sum_{ST,ST' \in Ref\_Set} Count(ST, ST')}{rs(rs-1)/2}$$

Then, this value is normalized between 0 and 1 as follows:

$$Div\_eval = \frac{avg\_div}{|V|-1}$$

As shown in Algorithm 2, where the main steps of our SS method are detailed, the Div_eval manages the convergence of the algorithm and therefore is responsible of its termination.

4. Experiments and Results

This section presents the computational experiments carried out on various sets of graphs to test the performance of the proposed SS algorithm. To compare it with the existing state of the art approach (the MA in [9]), we implemented both algorithms in C++ on ubuntu 16.04 LTS machine with Intel(R) Core(TM) i5-2400 CPU @3.10×4 GHz and 7.7 GiB of RAM.

The experiments are performed on four classes of graphs given below.

1. **Harwell-Boeing (HB) Graphs:** This set contains 62 instances of sparse matrices taken from the public domain Matrix Market library (available at https://math.nist.gov/MatrixMarket/data/Harwell-Boeing/) that are widely used in scientific and engineering problems. The size of graphs ranges from 24 to 918. Optimal results for these instances are not known.

2. **Cap graphs C[k]:** A Cap graph C[k] consists of $3k+1$ vertices. It has 4 levels with one vertex at level 1 and k vertices each in the remaining levels with edges as shown in Fig. 5. In this set, the instances are generated by varying the value of k between 2 to 4000 with a total of 100 instances. The optimal spanning tree for this graph has k leaves [23].

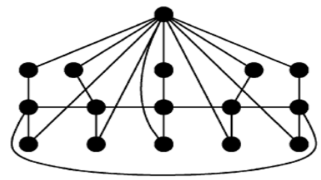

Figure 5: Cap graph with $3k+1$ vertices, $k=5$.

3. **Type 1 Graphs:** Type I graphs were originally generated by Carrabs et al. [33] to compare the performance of different relaxations for several formulations of the minimum branch vertices spanning tree problem. To obtain a meaningful comparison, sparse graphs with number of vertices n, and edge density less than 0.2 were generated, where the density is computed as $2|E|/\left[n(n-1)\right]$. Thus, the graph generator computes $|E| = \lfloor (n-1) + 1.5k\sqrt{n} \rfloor$ for $k = 1, 2, 3, 4, 5$. We consider in our experiments the instances in [9], which were obtained with that generator for 14 values of n (ranging from 150 to 1000), and generating five instances for each value of n and k, totalizing 350 graphs. These instances have known optimal values.

4. **Type 2 Graphs:** There are 94 instances in this set, generated by Cerrone et al [9] considering challenging scenarios. Specifically, starting with a cycle graph of four vertices, an instance with n vertices is constructed iteratively by adding edges incident on randomly selected non-adjacent vertices (see Figure 6). This set contains graphs with n ranging between 50 to 1500 with optimal values known (except for the instances with $n = 1500$).

The last two sets of test instances are also considered in [9] to test MA, the best method previously published. It is worth to mention here that they have also given a mathematical formulation of MLSTP and the optimal results of these instances were obtained with CPLEX, which can only find optimal results

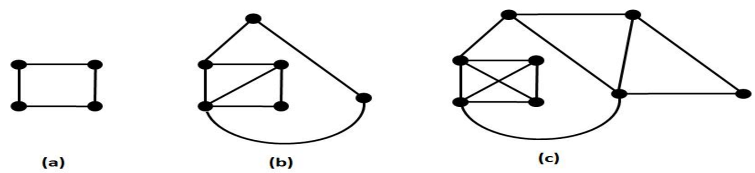

Figure 6: Construction process of Type 2 graphs.

for instances with size (number of nodes) less than 1500.

Our experimentation is divided into two parts: preliminary experiments (scientific testing) and main experiments (comparative testing). To avoid the over-training of our algorithm, we perform the preliminary experimentation on a small set of instances, and the comparison with the previous method on the entire set.

4.1. Scientific testing

For the initial set of experiments, 20 instances of HB graphs are considered to form a representative set. Specifically, it consists of the following graphs with different densities and sizes: *ash*85, *bcspwr*03, *bcsstk*01, *bcsstk*05, *bus*685, *can*_24, *can*_73, *can*_715, *dwt*_66, *dwt*_162, *dwt*_245, *dwt*_310, *dwt*_503, *dwt*_869, *gr*_30_30, *lshp*265, *lshp*577, *nos*4, *nos*7, *plat*362. As we have five construction heuristics and two reference set generation methods, we perform a thorough analysis to develop the final configuration of the SS method by using an appropriate combination of its components.

4.1.1. Size of Reference Set

This section describes the experiments performed in order to set the size rs of the $RefSet$. This is typically taken between 10 and 20 [30], so we test the values $rs = 5, 10, 15$, and 20. On the other hand, we set the population size

Table 1: Average results of SS on 20 HB graphs

	rs	**5**	**10**	**15**	**20**
small	***nLeaf***	7.14	6.92	6.75	6.65
	time (sec.)	1.86	16.79	85.54	302.38
large	***nLeaf***	17.37	14.64	13.61	12.96
	time (sec.)	125.1	302.97	384.8	561.1

$pop\_size = 100$, as recommended in [30]. For each value of rs, 20 independent runs of SS are performed on each instance. Table 1 reports the associated results.

The test set is divided into two subsets based on the size of the instances: small instances ($|V| \leq 300$) and large instances ($|V| > 300$). The results are shown in Table 1, where $nLeaf$ represents the average objective function values (number of leaves in the spanning tree), and $time$ represents the average CPU time for different values of rs.

We apply a two-way ANOVA with repetition to the data in Table 1, and at a 5% level of significance, it concludes that there is a significant difference in the $nLeaf$ obtained with these values of rs for both large and small instances. We also apply a pairwise comparison with TUKEY's HSD test, and it is found that each pair of means is significantly different (with the exception of $rs = 15$ and 20). Similar analysis and conclusions is achieved with CPU times. Considering that the running time substantially increases with rs, we initially set the value of $rs = 5$. However, preliminary experiments on large graphs disclosed that $rs = 5$ resulted in a premature convergence of SS, we therefore set $rs = 10$ for large graphs.

4.1.2. Comparison of Construction Heuristics

Our preliminary investigation with SS indicated a sharp decline in the diversity even though the process starts with diverse solutions. To overcome this limitation, we reinitialize the population at suitable intervals. In particular, we apply $H5$ since it is a completely random process to generate diverse solutions. On the other hand, to have a wide range of solutions, the remaining four construction heuristics (described in Section 3.1) are applied for the initial population.

To statistically analyse the performance of the construction heuristics, we replicate them 30 times on each instance in our representative set. Table 2 reports the mean value of $nLeaf$ and Div_eval for the four constructive methods (H_1, H_2, H_3, and H_4). A two-way ANOVA statistical test is applied to these mean values at a 5% level of significance. It concludes that the heuristics are significantly different. The TUKEY's HSD test confirms this conclusion.

A bi-objective analysis of the objective and diversity values shows that the pairs $(0.46, 16.27)$, $(0.60, 109.69)$ and $(0.64, 129.14)$ are non-dominated points that correspond to H_1, H_2 and H_3 respectively. Since the $nLeaf$ value obtained with H_1 is substantially lower than the remaining heuristics, we consider two different ways to create the initial population:to apply only H_1, and to use all the heuristics (in some suitable proportions). To decide the fraction of solutions to be contributed by each of the heuristic in the population a dense ranking scheme is used (see [34]), based on the following formula to determine the proportions:

$$pH_i = \frac{80 - RH_i}{\sum_{j=1}^{4}(80 - RH_j)}$$

where, pH_i and RH_i are the proportion and rank sum of the heuristic H_i respectively. With 4 heuristics and 20 instances, the maximum possible rank sum that a heuristic can have is 80. The 3rd row of Table 2, shows the rank sum of each heuristic and in the 4th row the corresponding proportion of each of these heuristics is given.

4.1.3. Reference Set Diversity Analysis

This section describes the experiments carried out to evaluate the effect of different combinations of construction heuristics (H_1, H_2, H_3 and H_4) and reference set formation methods (RM_1 and RM_2, detailed out in Section 3.4) on

Table 2: Comparison of construction heuristics

	H_1	H_2	H_3	H_4
Div_eval	0.46	0.60	**0.64**	0.42
nLeaf	**16.27**	109.69	129.14	179.16
Rank sum	20	57	46	77
pH_i (%)	50	19	28	3

the diversity of the reference set. The following four combinations are tested through the experiments:

(a) $COMB1$: All construction heuristics and RM_1

(b) $COMB2$: Construction heuristic H_1 and RM_1

(c) $COMB3$: All construction heuristics and RM_2

(d) $COMB4$: Construction heuristic H_1 and RM_2

For each combination above, a population of 100 solutions is generated using the construction heuristics, and a reference set is created from this population with the reference set formation method. Now, the diversity of this reference set is computed with the Div_eval function (see Section 3.5). Five independent runs of this experiment are performed on the instances of the representative set. For small and large instances the size of reference set is taken as 5 and 10 respectively (as decided in Section 4.1.1). Fig. 7 shows the average diversity of reference sets over the 20 instances and 5 trials for all the combinations. The mean diversities over all the instances of these combinations are given in Table 3. The results show that the diversity is maximum for $COMB3$ and minimum for $COMB2$.

We perform pairwise comparisons between the following combinations: (COMB1 COMB3) and (COMB2, COMB4) to compare the strategies for reference set

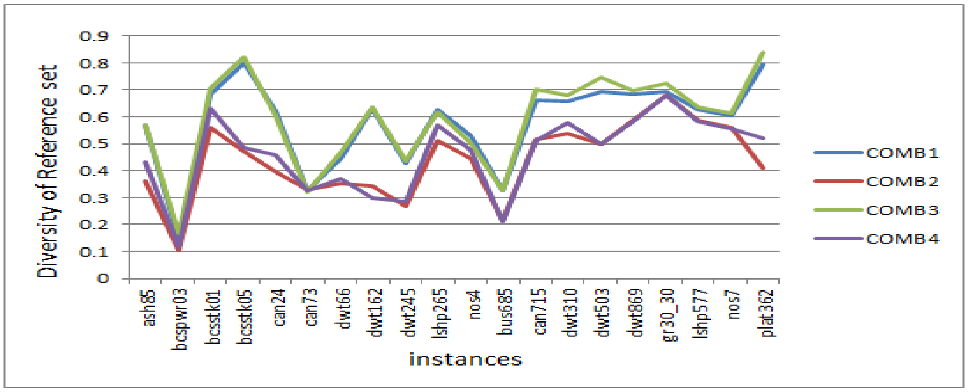

Figure 7: Reference set diversity for different combinations of constructions

Table 3: Different combinations of constructions

Combinations	**Mean Diversity**
$COMB1$	0.58
$COMB2$	0.44
$COMB3$	0.59
$COMB4$	0.46

formation, (COMB1, COMB2) and (COMB3, COMB4) to find out if a single construction heuristic or a group of different heuristics are able to produce diverse solutions for the reference set. The diversity of the reference set is compared for each of the pairs using a statistical paired two-sample t-test with 5% level of significance. A t-test is a type of inferential statistic used to determine if there is a significant difference between the means of two groups. It concludes in our case that all the pairs are significantly different. We therefore consider the four variants in our final experiments with SS.

4.1.4. Termination Criterion

Trials are conducted on the representative set instances to decide the number of iterations, max_iter. It is observed that if the reference set is not updated

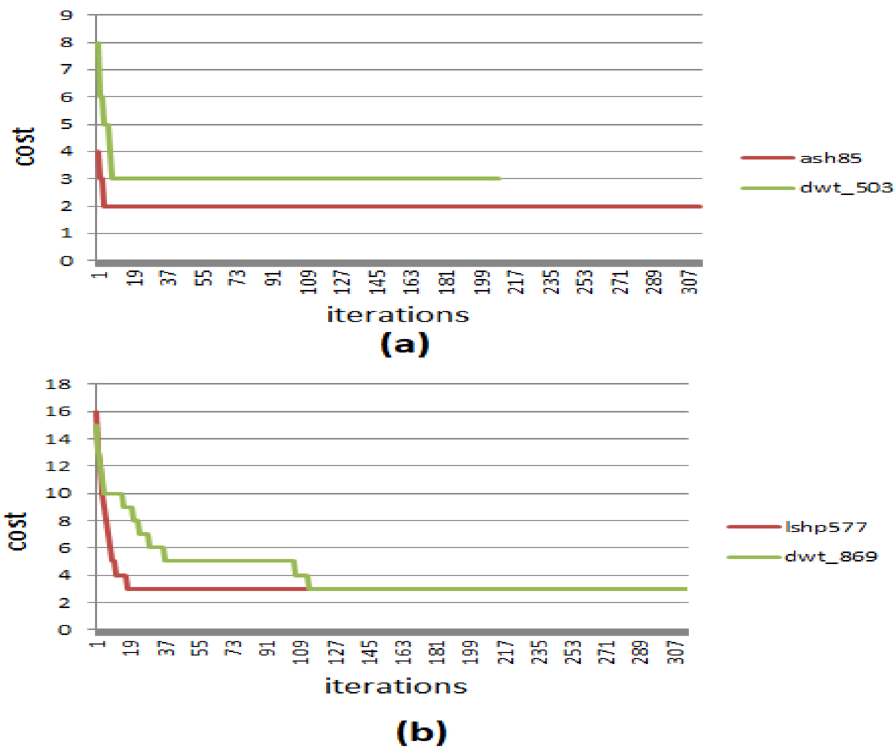

Figure 8: Convergence graphs of some HB instances

for 100 consecutive iterations, then the search can be terminated. This observation is based on the experiments in which the procedure is allowed to run for up to 200 iterations. It is seen that during the later part of the iterative procedure neither the best objective value improves nor an update of reference set takes place. Some convergence patterns are shown in Fig. 8.

4.2. Comparative testing

In this section, we first compare four variants of SS, namely $SS1$, $SS2$, $SS3$, and $SS4$, developed by generating the initial population in our SS method with $COMB1, COMB2, COMB3$, and $COMB4$ respectively (discussed in Section

4.1.3), and then use the best variant to compare SS with MA, the best previous method. Since some instances in our test-bed are very large and difficult to solve, we limit, if necessary, the execution of all methods to 1 hour of CPU time on each instance. Note however, that we only check the running time after each global iteration, and therefore the total elapsed time may be longer than the specified limit of 1 hour. We report in the tables the total time required by each method on each instance.

In our first comparative experiment, we apply the four SS variants on Type 1 graphs. We compare them with a two way-ANOVA as in the preliminary experiments. The test indicates rejection of the null hypothesis. A pairwise comparison is done by applying TUKEY's HSD test which shows that the performance of pairs $(SS1, SS2)$ and $(SS1, SS3)$ is significantly different. The mean values of $nLeaf$ obtained by these variants is given in Table 4. The 1st and 2nd columns in this table report the graph size and number of instances of that size, respectively. The average of the optimal values for each set is shown in 3rd column. Columns 4th to 7th give the results obtained with $SS1$, $SS2$, $SS3$, and $SS4$ respectively.

Results in Table 4 indicate that the diversity of the reference set formed using a single construction heuristic is lower than when it is created using all the construction heuristics (see Table 3). A diverse population is needed in SS for creating a reference set, so using all construction heuristics gives better performance than a single one. Though, $SS1$ and $SS3$ both generate the initial population using all the construction heuristics but $SS1$ outperforms $SS3$ (see Table 4). $SS1$ and $SS3$ differ over the reference set formation strategies. Better performance of $SS1$ may be attributed to the fact that reference set in $SS1$ contains both types of solutions, namely, elite and diverse in equal proportion since it employs RM_1 for reference set formation while method RM_2 focuses only on the diversity of solutions in the reference set, and hence may not contain the elite solutions of initial population. Clearly, $SS1$ invariably gives the lowest mean value of $nLeaf$ (shown in bold) over all the instances, so the rest of the experiments is done with this variant. In the remaining part of this paper $SS1$

Table 4: SS variants on Type 1 graphs

$\lvert V \rvert$	**#ins**	**Opt**	**SS1**	**SS2**	**SS3**	**SS4**
150	25	50.40	**50.40**	50.80	50.64	50.68
160	25	55.12	**55.12**	55.28	55.24	55.32
170	25	58.20	**58.36**	58.56	58.64	58.52
180	25	64.60	**64.72**	64.76	64.84	64.80
190	25	70.36	**70.48**	70.56	70.64	70.60
200	25	72.80	**72.84**	72.96	72.96	72.92
250	25	97.68	**97.72**	97.88	97.96	97.88
300	25	123.36	**123.52**	123.56	123.68	123.64
350	25	146.00	146.12	146.32	**146.04**	146.12
400	25	174.52	**174.56**	**174.56**	174.60	174.60
450	25	197.64	**197.68**	197.72	197.76	197.72
500	25	226.44	**226.48**	226.68	226.64	226.60
750	25	437.00	**437.00**	**437.00**	**437.00**	**437.00**
1000	25	595.00	**595.00**	**595.00**	**595.00**	**595.00**
avg		169.22	**169.29**	169.40	169.40	169.39

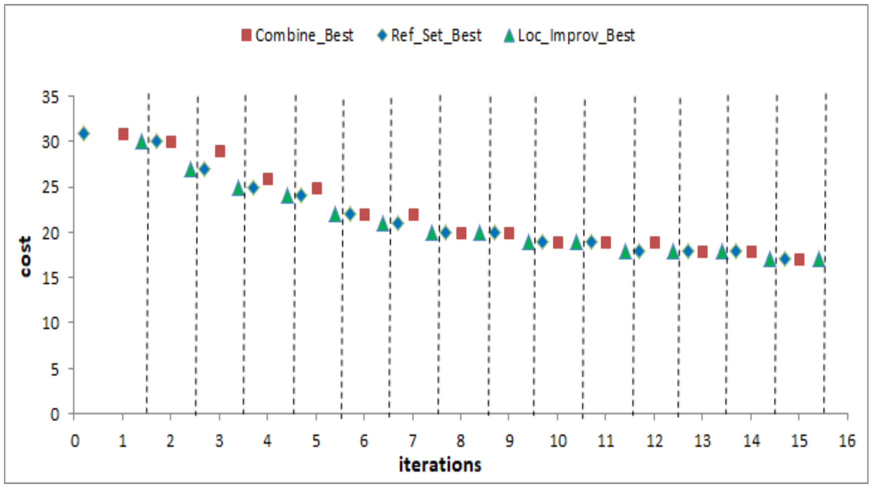

Figure 9: Evolution of the best solution by combination and improvement methods.

will be referred as SS.

Figure 9 shows the evolution of the SS solutions over a standard run. During the initial iterations, the best solution in the $RefSet$ significantly improves by means of the combination and improvement operators. In this figure, $Combine\_Best$ and Loc_Improv_Best refer to the best values of the solutions after the application of these operators respectively. This diagram clearly shows the evolution of the best solution found, and the contribution of the combination and improvement operators to it.

We now compare the performance of SS with MA on Type 1 and Type 2 graphs. The results of our experiment are reported in Tables 5 and 6 respectively. Columns '$Best$' and 'Avg' report the mean values of the minimum and average of $nLeaf$ over 10 runs in each group of instances with the same size (each row in the table). The number of instances for each group in table 5 is 25. In Table 6, the number of instances for the group with $|V| = 1000$ and 1250 is 8 and 6 respectively, whereas for rest of the graphs it is 10. The column '$time$' reports the average running time of the algorithms over 10 runs. The last row of these tables ('avg') shows the average of the results over all the instances. The values in bold show the improvement of SS over MA. From the results, it

Table 5: Comparison of SS and MA for Type 1 graphs

$\lvert V \rvert$		SS			MA		
	Opt	*Best*	*Avg*	*time*	*Best*	*Avg*	*time*
150	50.40	50.40	50.80	4.24	51.08	51.34	155.37
160	55.12	55.12	55.38	3.87	55.88	56.06	183.11
170	58.20	58.36	58.76	5.10	59.36	59.68	214.49
180	64.60	64.72	65.04	5.13	65.52	65.90	249.01
190	70.36	70.48	70.70	5.49	71.52	71.70	287.15
200	72.80	72.84	73.06	5.52	74.20	74.56	329.32
250	97.68	97.72	97.90	8.40	100.56	100.90	604.09
300	123.36	123.52	123.76	10.64	127.76	128.10	1000.97
350	146.00	146.12	146.19	38.58	151.60	151.98	1544.40
400	174.52	174.56	174.62	47.17	181.16	181.78	2261.59
450	197.64	197.68	197.72	48.14	206.00	206.64	3174.78
500	226.44	226.48	226.54	-	237.32	237.88	-
750	437.00	437.00	437.00	-	437.08	437.28	-
1000	595.00	595.00	595.00	-	603.52	603.98	-
avg	169.22	**169.29**	**169.46**	-	173.04	173.41	-

is clear that SS performs better than MA for both classes of graphs.

The following experiment is performed on the Cap graphs. In this set of instances, SS achieves optimal results in all the instances tested with $k = 2$ to 500. Further insight into the procedure reveals that these optimal values are attained after on the initial population, just with the application of improvement operators. This was further verified by extending the observations for values of k up to 4000. This shows that our improvement operators are quite effective in reducing the number of leaves, especially in Cap graphs. It is important to remark here that no optimal solution is present in the initial population in any of the above mentioned instances. Considering that the optimal results are achieved with just applying the improvement operator to the solutions in the

Table 6: Comparison of SS-MLSTP and MA for Type 2 graphs

$\lvert V\rvert$		SS-MLSTP			MA		
	Opt	*Best*	*Avg*	*time*	*Best*	*Avg*	*time*
50	7.40	7.40	7.73	1.64	7.80	7.97	18.53
100	17.30	17.30	17.91	7.01	17.60	17.80	61.75
200	32.20	32.50	33.67	49.58	32.60	33.07	339.17
300	51.30	51.90	54.03	196.50	52.30	53.13	1024.42
400	65.70	66.60	68.12	1566.10	67.00	68.09	2298.23
500	88.00	89.20	90.43	-	90.90	92.10	-
750	135.20	141.50	143.07	-	144.50	146.20	-
1000	194.75	208.75	210.79	-	217.50	220.29	-
1250	252.83	284.17	285.87	-	358.00	362.28	-
1500	-	333.40	336.47	-	708.70	712.00	-
avg	-	**123.27**	**124.81**	-	169.69	171.29	-

initial population, we do not report them in our tables to compare SS and MA.

Tables 7 and 8 report the results of our last experiment on small ($|V| \leq 300$) and large ($300 < |V| \leq 1000$) HB graphs respectively. From the 62 instances of HB graphs tested, SS is able to achieve optimal results in 45 instances, whereas in MA this value is 25. For the remaining instances, the results of SS are quite close to the known lower bounds. The comparison of results obtained by both algorithms shows that SS outperforms MA in both sets of graphs.

Table 7: Comparison of SS and MA for small HB graphs

Graphs	SS-MLSTP			MA		
	Best	*Avg*	*time*	*Best*	*Avg*	*time*
$\lvert V\rvert \leq 100$	6.47	6.50	7.33	6.60	6.65	30.50
$100 < \lvert V\rvert \leq 200$	3.75	3.89	47.83	4.13	4.30	195.00
$200 < \lvert V\rvert \leq 300$	10.00	10.36	253.95	9.80	10.51	672.78
avg	**6.74**	**6.92**	**103.04**	6.84	7.15	299.43

Table 8: Comparison of SS-MLSTP and MA for large HB graphs

Graphs	SS-MLSTP			MA		
	Best	*Avg*	*time*	*Best*	*Avg*	*time*
$\lvert V\rvert \leq 500$	23.75	24.39	-	25.08	25.93	-
$500 < \lvert V\rvert \leq 750$	21.89	22.73	-	25.78	27.46	-
$750 < \lvert V\rvert \leq 1000$	3.50	5.18	-	21.38	25.85	-
avg	**16.38**	**17.43**	-	24.08	26.41	-

Table 9: Summary of results

	Type 1		Type 2		HB	
	SS	MA	SS	MA	SS	MA
$nLeaf$	**169.29**	173.04	**123.27**	169.69	**11.56**	15.46
dev					-	-
n_best					**61**	31
n_opt						
$time$						

The overall comparison of SS and MA on all the instances of three sets of graphs is shown in Table 9. The first two rows compares the average value of best $nLeaf$ and average percentage deviation (dev) from the best known/ optimal of the results obtained by both the methods. Since optimal results are not available for HB graphs, therefore dev is not computed for this class of graphs. 'n_best' and 'n_opt' represent the number of best and optimal solutions found by the algorithms respectively. The execution time, $time$, in seconds is reported in the last row. This table clearly shows that SS on average obtains better results than MA, thus showing the practical contribution of our method.

5. Conclusion

In spite of having significant applications in communication networks, the MLSTP has been tackled with very few approaches. This paper proposes an efficient solving method based on the Scatter Search methodology for the MLSTP. The algorithm generates an initial population of diverse solutions, and then it explores and exploits this population by means of combination and local improvement operators. Emphasis on a good reference set formation has been given by admitting solutions in such a way that the method keeps a good balance between the quality and diversity of the population.

We perform extensive experiments in order to obtain the best configuration of the SS operators for MLSTP. Preliminary experimentation is performed on HB graphs, as they are frequently used as benchmarks for similar optimization problems. These experiments help us to disclose the values of key search parameters and the best strategies for our final SS method. Final experiments are conducted, first on Cap graphs with optimal known, and then on three classes of graphs namely, Type 1, Type 2 and HB graphs, consisting of 506 instances. It shows that SS performs better than MA in terms of the objective function but it requires significantly lower running times.

This study goes beyond the application of a well-established methodology, Scatter Search in our case, to solve a difficult problem, the MLSTP. It actually investigate alternatives to the classic design, proposed in [35] and applied in many recent papers, such as [36], or [31]. The strategies proposed here, after tested in other problems, can be the foundations of advanced Scatter Search designs.

Acknowledgment

The first author would like to thank University Grants Commission (UGC), India for providing RGNF fellowship [RGNF-2014-15-SC-UTT-74975] during the research. Prof. Martí research has been partially supported by the Spanish

Ministry of Science, Innovation, and Universities with grant refs. PGC2018-0953322-B-C21/MCIU/AEI/FEDER-UE.

References

[1] B. Y. Wu, K. M. Chao, Spanning trees and optimization problems: Discrete mathematics and its applications, Washington, D.C., 2004.

[2] M. I. Ostrovskii, Minimal congestion trees, Discrete Mathematics 285 (1-3) (2004) 219–226. `doi:https://doi.org/10.1016/j.disc.2004.02.009`.

[3] M. R. Garey, D. S. Johnson, Computers and intractability: A Guide to the Theory of NP Completeness, W. H. freeman, New York, 1979.

[4] M. S. Rahman, M. Kaykobad, Complexities of some interesting problems on spanning trees, Information Processing Letters 94 (2) (2005) 93–97. `doi:https://doi.org/10.1016/j.ipl.2004.12.016`.

[5] G. Salamon, A survey on algorithms for the maximum internal spanning tree and related problems, Electronic Notes in Discrete Mathematics 36 (2010) 1209–1216. `doi:https://doi.org/10.1016/j.endm.2010.05.153`.

[6] M. Knauer, J. Spoerhase, Better approximation algorithms for the maximum internal spanning tree problem, Algorithmica 71 (4) (2015) 797–811. `doi:https://doi.org/10.1007/s00453-013-9827-7`.

[7] X. LI, D. Zhu, A 4/3-approximation algorithm for the maximum internal spanning tree problem on graphs without leaves, Journal of Computational Information Systems 11 (15) (2015) 5607–5617.

[8] D. B. Raible, H. Fernau, S. Gaspers, M. Liedloff, Exact and parameterized algorithms for max internal spanning tree, Algorithmica 65 (1) (2013) 95–128. `doi:https://doi.org/10.1007/s00453-011-9575-5`.

[9] C. Cerrone, R. Cerulli, A. Raiconi, Relations, models and a memetic approach for three degree-dependent spanning tree problems, European Journal of Operational Research 232 (3) (2014) 442–453. doi:https://doi.org/10.1016/j.ejor.2013.07.029.

[10] H. Lu, R. Ravi, The power of local optimization: Approximation algorithms for maximum-leaf spanning tree (draft), in: Proc. 30th Annual Allerton Conference on Communication Control and Computing, 1996, pp. 533–542.

[11] E. Prieto, C. Sloper, Either/or: Using vertex cover structure in designing fpt algorithms the case of k-internal spanning tree, in: Workshop on Algorithms and Data Structures. WADS (Lecture Notes in Computer Science, vol. 2748), Springer, 2003, pp. 474–483. doi:https://doi.org/10.1007/978-3-540-45078-8_41.

[12] E. Prieto, Systematic kernelization in FPT algorithm design, Ph.D. dissertation, School of Electrical Engineering and Computer Science, The University of Newcastle, NSW, Australia, 2005.

[13] G. Salamon, Approximation algorithms for the maximum internal spanning tree problem, in: Mathematical Foundations of Computer Science. MFCS Lecture Notes in Computer Science, vol. 4708, L. Kucera and A. Kucera, Eds., Springer, Berlin/Heidelberg, Germany, 2007, pp. 90–102. doi:https://doi.org/10.1007/978-3-540-74456-6_10.

[14] G. Salamon, Approximating the maximum internal spanning tree problem, Theoretical Computer Science 410 (50) (2009) 5273–5284. doi:https://doi.org/10.1016/j.tcs.2009.08.029.

[15] X. Li, D. Zhu, A 4/3-approximation algorithm for finding a spanning tree to maximize its internal vertices, arXiv preprint arXiv:1409.3700, 2014.

[16] X. Li, H. Feng, H. Jiang, B. Zhu, A polynomial time algorithm for finding a spanning tree with maximum number of internal vertices on interval graphs, in: 10th International Workshop on Frontiers in Algorithmics.

FAW (Lecture Notes in Computer Science, vol. 9711), D. Zhu and S. Bereg, Eds., Springer, Qingdao, China, 2016, pp. 92–101. doi:https://doi.org/10.1007/978-3-319-39817-4_10.

[17] W. Li, Y. Cao, J. Chen, J. Wang, Deeper local search for parameterized and approximation algorithms for maximum internal spanning tree, Information and Computation 252 (2017) 187–200. doi:https://doi.org/10.1016/j.ic.2016.11.003.

[18] Z. Z. Chen, Y. Harada, F. Guo, L. Wang, An approximation algorithm for maximum internal spanning tree, Journal of Combinatorial Optimization 35 (2018) 955âĂŞ979. doi:https://doi.org/10.1007/s10878-017-0245-7.

[19] Z. Z. Chen, G. Lin, L. Wang, Y. Chen, D. Wang, Approximation algorithms for the maximum weight internal spanning tree problem, Algorithmica 81 (2019) 4167–4199. doi:https://doi.org/10.1007/s00453-018-00533-w.

[20] K. Casel, J. Dreier, H. Fernau, M. Gobbert, P. Kuinke, F. S. Villaamil, M. L. Schmid, E. J. Leeuwen, Complexity of independency and cliquy trees, Discrete Applied Mathematics 272 (2020) 2–15, 15th Cologne-Twente Workshop on Graphs and Combinatorial Optimization (CTW 2017). doi:https://doi.org/10.1016/j.dam.2018.08.011.

[21] W. Li, Y. Ding, Y. Yang, R. S. Sherratt, J. H. Park, J. Wang, Parameterized algorithms of fundamental np-hard problems: a survey, Human Centric Computing and Information Sciences 10 (2020). doi:https://doi.org/10.1186/s13673-020-00226-w.

[22] J. Goedgebeur, K. Ozeki, N. V. Cleemput, G. Wiener, On the minimum leaf number of cubic graphs, Discrete Mathematics 342 (11) (2019) 3000–3005. doi:https://doi.org/10.1016/j.disc.2019.06.005.

[23] G. Salamon, G. Wiener, On finding spanning trees with few leaves, Information Processing Letters 105 (5) (2008) 164–169. doi:https://doi.org/10.1016/j.ipl.2007.08.030.

[24] S. Boyd, R. Sitters, S. V. Ster, L. Stougie, The traveling salesman problem on cubic and subcubic graphs, Mathematical Programming 144 (2014) 227–245. doi:https://doi.org/10.1007/s10107-012-0620-1.

[25] A. Biniaz, Better approximation algorithms for maximum weight internal spanning trees in cubic graphs and claw-free graphs, arXiv preprint arXiv:2006.12561, 2020.

[26] X. Li, D. Zhu, L. Wang, A 4/3-approximation algorithm for the maximum internal spanning tree problem, Journal of Computer and System Sciences 118 (2021) 131–140. doi:https://doi.org/10.1016/j.jcss.2021.01.001.

[27] F. Glover, Heuristics for integer programming using surrogate constraints, Decision Sciences 8 (1) (1977) 156–166. doi:https://doi.org/10.1111/j.1540-5915.1977.tb01074.x.

[28] M. A. González, A. Oddi, R. Rasconi, R. Varela, Scatter search with path relinking for the job shop with time lags and setup times, Computers & Operations Research 60 (2015) 37–54. doi:https://doi.org/10.1016/j.cor.2015.02.005.

[29] J. S. Oro, M. Laguna, A. Duarte, R. Martí, Scatter search for the profile minimization problem, Networks 65 (1) (2015) 10–21. doi:https://doi.org/10.1002/net.21571.

[30] F. Glover, M. Laguna, R. Martí, Scatter search and path relinking: Advances and applications, in: Handbook of metaheuristics (International Series in Operations Research and Management Science, vol. 57), F. Glover and G. Kochenberger (Eds.), New York, NY, USA: Springer, 2003, pp. 1–35. doi:https://doi.org/10.1007/0-306-48056-5_1.

[31] J. S. Oro, M. Laguna, R. Martí, A. Duarte, Scatter search for the bandpass problem, Journal of Global Optimization 66 (4) (2016) 769–790. doi:https://doi.org/10.1007/s10898-016-0446-0.

[32] K. Deb, Multi-objective optimization using evolutionary algorithms. Wiley-Interscience series in systems and optimization, Vol. 16, 2001.

[33] F. Carrabs, R. Cerulli, M. Gaudioso, M. Gentili, Lower and upper bounds for the spanning tree with minimum branch vertices, Computational Optimization and Applications 56 (2013) 405–438. doi:https://doi.org/10.1007/s10589-013-9556-5.

[34] P. Jain, K. Srivastava, G. Saran, Minimizing cyclic cutwidth of graphs using a memetic algorithm, Journal of Heuristics 22 (6) (2016) 815–848. doi:https://doi.org/10.1007/s10732-016-9319-4.

[35] M. Laguna, R. Martí, Scatter Search. Methodology and Implementations in C, Kluwer Academic Publishers, Springer, 2003.

[36] J. S. Oro, A. M. Gavara, M. Laguna, R. Martí, A. Duarte, Variable neighborhood scatter search for the incremental graph drawing problem, Computational Optimization and Applications 68 (3) (2017) 775–797. doi:https://doi.org/10.1007/s10589-017-9926-5.