Hindawi
Complexity
Volume 2020, Article ID 7628105, 25 pages
https://doi.org/10.1155/2020/7628105

WILEY | Hindawi

Research Article

Degree-Constrained k-Minimum Spanning Tree Problem

Pablo Adasme[1] **and Ali Dehghan Firoozabadi**[2]

[1]*Department of Electrical Engineering, Universidad de Santiago de Chile, Avenida Ecuador 3519, Santiago, Chile*
[2]*Department of Electricity, Universidad Tecnológica Metropolitana, Av. Jose Pedro Alessandri 1242, 7800002 Santiago, Chile*

Correspondence should be addressed to Pablo Adasme; pablo.adasme@usach.cl

Received 15 June 2020; Revised 25 August 2020; Accepted 10 October 2020; Published 9 November 2020

Academic Editor: Qingling Wang

Copyright © 2020 Pablo Adasme and Ali Dehghan Firoozabadi. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Let $G(V, E)$ be a simple undirected complete graph with vertex and edge sets V and E, respectively. In this paper, we consider the degree-constrained k-minimum spanning tree (DCkMST) problem which consists of finding a minimum cost subtree of G formed with at least k vertices of V where the degree of each vertex is less than or equal to an integer value $d \leq k - 2$. In particular, in this paper, we consider degree values of $d \in \{2, 3\}$. Notice that DCkMST generalizes both the classical degree-constrained and k-minimum spanning tree problems simultaneously. In particular, when $d = 2$, it reduces to a k-Hamiltonian path problem. Application domains where DCkMST can be adapted or directly utilized include backbone network structures in telecommunications, facility location, and transportation networks, to name a few. It is easy to see from the literature that the DCkMST problem has not been studied in depth so far. Thus, our main contributions in this paper can be highlighted as follows. We propose three mixed-integer linear programming (MILP) models for the DCkMST problem and derive for each one an equivalent counterpart by using the handshaking lemma. Then, we further propose ant colony optimization (ACO) and variable neighborhood search (VNS) algorithms. Each proposed ACO and VNS method is also compared with another variant of it which is obtained while embedding a Q-learning strategy. We also propose a pure Q-learning algorithm that is competitive with the ACO ones. Finally, we conduct substantial numerical experiments using benchmark input graph instances from TSPLIB and randomly generated ones with uniform and Euclidean distance costs with up to 400 nodes. Our numerical results indicate that the proposed models and algorithms allow obtaining optimal and near-optimal solutions, respectively. Moreover, we report better solutions than CPLEX for the large-size instances. Ultimately, the empirical evidence shows that the proposed Q-learning strategies can bring considerable improvements.

1. Introduction

Intelligent communication systems will be mandatorily required in the next decades to provide low-cost connectivity within many application domains involving network structures in the form of ring, tree, and star topologies [1–3], for instance, when designing network structures in telecommunications, facility location, electrical power systems, water and transportation networks, and for networks to be constructed under the Internet of Things (IoT) paradigm [2–10]. A particular example related with wireless sensor networks is due to density control methods [1, 11, 12]. These methods allow reducing energy consumption in sensor networks by means of activation or deactivation mechanisms which mainly consist of putting into sleep mode some of the nodes of the network while ensuring sensing operations, communication, and connectivity [1–3, 12–16].

Let $G(V, E)$ represent a simple undirected complete graph with set of nodes $V = \{1, \ldots, n\}$ and set of edges $E = \{1, \ldots, m\}$. In this paper, we consider the degree-constrained k-minimum spanning tree problem (DCkMST), which consists of finding a minimum cost subtree of G formed with at least $k \leq n$ vertices of V where each vertex has a degree lower than or equal to $d \in \{2, \ldots, k - 2\}$. Notice that, for $d = 2$, DCkMST reduces to find a minimum cost Hamiltonian path with cardinality k. The DCkMST problem generalizes two classical combinatorial optimization

problems, namely, the degree-constrained and k-minimum spanning tree problems (resp., DCMST and kMST). Recently, a variant of DCkMST was studied in [17], where the authors assume that there exists a predefined root node that should not satisfy the degree constraint. In this paper, we consider the more general case where all nodes should satisfy this condition. Proofs related with the NP-hardness of DCMST and kMST can be found, respectively, in [18–20]. However, other applications related to the kMST problem can be consulted in [17, 20]. Notice that the degree constraints ensure that no overloaded nodes are present in the network.

It is easy to check from the literature that the DCkMST problem has not been addressed in depth so far. Consequently, our main contributions in this paper can be highlighted as follows. First, we propose three mixed-integer linear programming (MILP) models for the DCkMST problem and derive for each one an equivalent formulation using the handshaking lemma [21–23]. More precisely, we propose four compact polynomial formulations and two exponential models. Two of the compact models are formulated based on a Miller–Tucker–Zemlin-constrained approach, whilst the two remaining ones are flow-based models. We solve our exponential models with an exact iterative method adapted from [2, 24]. In order to obtain feasible solutions alternatively, we further propose ant colony optimization and variable neighborhood search (resp., ACO and VNS for short) algorithms for $d = 2$ and $d = 3$, respectively [25–29]. We choose VNS and ACO metaheuristics as they are well-known techniques in the operations research community which have proved to be highly efficient in order to find feasible solutions for many hard combinatorial optimization problems [25–29]. In particular, within each iteration of the VNS algorithm, for each random k–tuple of vertices generated, we obtain degree-constrained spanning trees by using a modified penalty approach proposed in [30]. However, for the VNS algorithm, we further introduce an embedded Q-learning strategy that allows performing a random local search based on the experience of previous solutions found [31]. The Q-learning approach is a simple reinforcement learning technique initially proposed in [31] that allows an agent to interact with its environment in order to learn from the experience of previous occurrences the best action to choose from a set of actions in order to maximize its revenue. The underlying idea in doing so is to construct optimization methods with learning capabilities in order to make them robust, self-adaptive, and independent from decision-makers. Notice that recently there is a growing interest from the operation research community in developing novel machine learning-based optimization methods that allow solving hard combinatorial optimization problems [32]. Consequently, embedding a Q-learning strategy in a random local search algorithm is a novel approach to the literature. Our embedded Q-learning strategy is then compared with a traditional VNS near-far random local search procedure [28]. Another paper dealing with a similar embedded reinforcement learning strategy using VNS is proposed in [33]. More precisely, the authors propose a reactive search VNS method that allows learning which is the order in which different local search heuristics must be applied in order to obtain better solutions based on the experience of previous trials. Their method is applied and tested on the symmetric traveling salesman problem obtaining good results in terms of solution quality and speed. The embedded Q-learning strategy in our VNS approach is different as we use it as a learning mechanism in order to perform a random local search. Besides, it does not require the use of specialized local search methods, and consequently, it can be extended and used straightforwardly for any other combinatorial optimization problems. Similarly, we also compare the proposed ACO algorithm with an adapted version of the generic ant-Q method proposed in [27]. Finally, we propose a pure Q-learning-based algorithm that is competitive with both ACO methods. We report numerical results for Euclidean benchmark instances from [34] and for randomly generated ones with both uniform and Euclidean distance costs with up to 400 nodes.

The paper is organized as follows: in Section 2, we present some related work, whilst in Section 3, we present the MILP formulations. Then, in Section 4, we present and explain each proposed algorithm. Next, in Section 5, we conduct substantial numerical experiments in order to compare all the proposed models and algorithms. Finally, in Section 6, we give the main conclusions of the paper and provide some insights into future research.

2. Related Work

As mentioned in Section 1, the DCkMST problem generalizes both the DCMST and kMST problems simultaneously. Consequently, previously related work is mainly focused on these graph combinatorial optimization problems. Notice that our work in this paper corresponds to an extended version of the preliminary work reported in [21]. However, now we generalize the previous formulations presented in [21] and allow each model to obtain feasible solutions with at least k vertices instead of using a unique value of k. Thus, the proposed models in this paper are more accurate with respect to the definition of the kMST problem [20]. In addition, we propose several algorithmic approaches for the DCkMST problem. Finally, we report numerical results for complete graph instances using degree values of $d \in \{2, 3\}$. Notice that solving complete graph instances for the DCMST problem while using degree values of $d \in \{2, 3\}$ implies solving the hardest type of instances as reported in the literature [35].

Regarding the complexity of the kMST problem, it has proved to be NP-hard by reduction from the Steiner tree problem for variable values of k [19, 20]. Consequently, it is hard to obtain an approximation ratio better than (96/95) [36]. In fact, the best-known approximation ratio for this problem is 2 and is reported in [37]. Some recent metaheuristic approaches are due to [38, 39]. In particular, the authors in [39] propose a new hybrid algorithm based on tabu search and ant colony optimization leading to better numerical results than state-of-the-art values reported in the literature. Similarly, in [38], the authors propose a hybrid

approach using a memetic algorithm where the genetic operator is based on a dynamic programming method. Numerical results show that their proposed method outperforms several existing algorithms in terms of solution quality and accuracy.

On the contrary, the DCMST problem has been studied extensively in the literature including exact and heuristic approaches such as Lagrangian relaxations, approximation algorithms, and branch-and-cut and metaheuristics approaches [22, 30, 35, 40–44]. In particular, the authors in [35] propose a Lagrangian-based heuristic for the DCMST problem that uses a greedy construction heuristic followed by a heuristic improvement procedure. They report extensive computational experiments and indicate that their solving method is competitive with the best heuristics and metaheuristics proposed in the literature. Instances with up to 2000 nodes are reported. Similarly, the authors in [22] propose a branch-and-cut algorithm for this problem dealing with instances with as many as 800 nodes. Concerning metaheuristic approaches, a novel genetic algorithm is proposed in [41]. More precisely, the authors first propose a transformation of the problem into a preference with two objective minimum spanning tree problems. Then, new crossover, mutation, and selection operators together with a local search approach are derived based on the preference of the two objectives. Finally, they show that their algorithm converges with probability one to a globally optimal solution. A more recent VNS approach is presented in [45] where the authors develop ideas that allow the enhancement of the performance of the VNS by guiding the search in the shaking phase while using second-order algorithms and skewed functions. They conduct computational experiments on hard benchmark instances from the literature and improve upon the best-known solutions. Finally, their solutions significantly reduced the gaps with respect to tight lower bounds reported in the literature as well. Other variants in the DCMST problem such as the multiperiod DCMST and min-degree-constrained minimum spanning tree problems are presented in [46, 47].

In this paper, our VNS approach for the DC*k*MST problem performs a traditional near-far random local search procedure as performed in the reduced VNS presented in [28, 29] and also a novel one which consists of embedding a Q-learning strategy [31] for each random generated k-tuple of vertices of V. In both cases, we use a modified penalty heuristic approach proposed in [30] in order to find spanning trees while satisfying the degree constraints. Finally, our ACO algorithms are adapted from [25–27] and incorporate an adaptive mechanism that allows obtaining consecutive k-Hamiltonian paths within each iteration in order to find feasible solutions. Practical applications related to the DCMST include the design of computer and communication networks, electric circuits, design of integrated circuits, road networks, and energy networks, among many others [17, 35]. Notice that, aside from these applications, all the proposed models and algorithms could also be extended and adapted for multiagent systems with varying graph topologies in order to deal with self-organization and congestion control in communication networks [48, 49].

3. Mathematical Formulations

In this section, for the sake of clarity, we first present a feasible solution for the DC*k*MST problem, and then, we present and explain the proposed MILP models.

3.1. A Feasible Solution for the DCkMST Problem. In Figure 1(a), we depict an input complete graph instance with 10 nodes where the coordinates of each node are randomly generated within a square area of $1\,\text{km}^2$, whereas in Figures 1(b) and 1(c), we show feasible solutions obtained for the instance in Figure 1(a) while using degree values of 3 and 2, respectively. For the sake of clarity, in Figures 1(b) and 1(c), we only draw the edges and highlight with green color the nodes which are part of the solutions, i.e., the nodes $\{1, 2, 3, 5, 6, 7\}$. Notice that the minimum cost value of an optimal solution in Figure 1(b) should be lower than or equal to the minimum cost value of an optimal solution in Figure 1(c) since the latter is also a feasible solution for the degree value of 3. On the opposite, the solution in Figure 1(b) is not feasible for the degree value of 2.

Proposition 1. *The following statements are true:*

(1) *If $k = n$ and $d \le n - 2$, the DCkMST problem reduces to the DCMST problem.*
(2) *If $k < n$ and $d \ge k - 1$, the DCkMST problem reduces to the kMST problem.*
(3) *If $k = n$ and $d \ge n - 1$, the problem reduces to the classical minimum spanning tree (MST) problem, which can be solved in polynomial time by a greedy algorithm [50, 51].*

From Proposition 1, it is easy to see that the DCkMST problem is NP-hard as it generalizes both the DCMST and KMST problems. Hereafter, we present the MILP models.

3.2. MILP Models. In order to formulate a first MILP model, let $H = (V, A)$ be the directed graph obtained from $G(V, E)$, where each edge $(i, j) \in E$ is replaced by the two arcs (i, j) and $(j, i) \in A$. Thus, we have the following proposition.

Proposition 2. *Model P_1 allows to obtain an optimal solution for the DCkMST problem with minimum cost:*

$$P_1: \min_{\{x,u,z\}} \sum_{(i,j)\in A} P_{ij} z_{ij}, \tag{1}$$

$$\text{s.t.} \quad \sum_{j\in V} x_j \ge k, \tag{2}$$

$$\sum_{(i,j)\in A} z_{ij} = \sum_{j\in V} x_j - 1, \tag{3}$$

$$u_i \le \sum_{j\in V} x_j, \quad \forall i \in V, \tag{4}$$

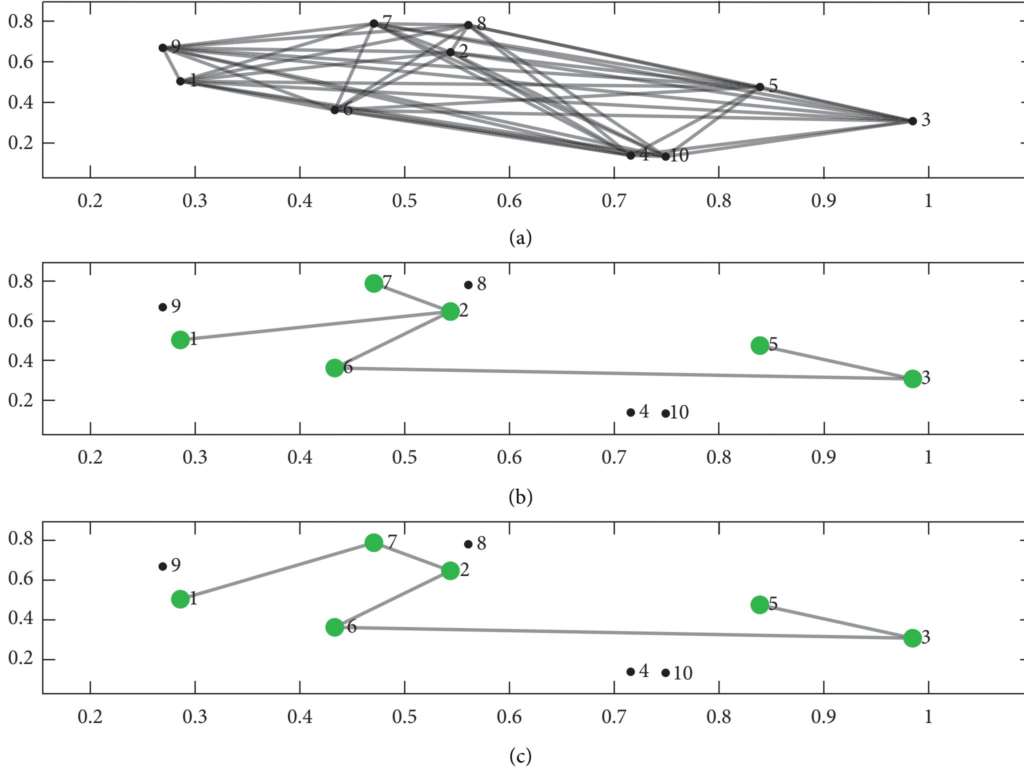

FIGURE 1: An input complete graph instance with feasible solutions for the DC*k*MST problem while using degree values of 3 and 2, respectively: (a) complete graph instance composed of 10 nodes with random uniform costs; (b) feasible solution with node degree values of at most 3; (c) feasible solution with node degree values of at most 2.

$$u_i \geq x_i, \quad \forall i \in V, \tag{5}$$

$$\sum_{i|(i,j)\in A} z_{ij} \leq x_j, \quad \forall j \in V, \tag{6}$$

$$u_j - u_i - (n-1)z_{ij} - (n-3)z_{ji} \geq 2-n, \quad \forall (i,j) \in A, \tag{7}$$

$$\sum_{i|(i,j)\in A} z_{ij} + \sum_{i|(j,i)\in A} z_{ji} \leq \mathrm{d}x_j, \quad \forall j \in V, \tag{8}$$

$$\sum_{i|(i,j)\in A} z_{ij} + \sum_{i|(j,i)\in A} z_{ji} \geq x_j, \quad \forall j \in V, \tag{9}$$

$$z_{ij} + z_{ji} \leq x_i, \quad \forall (i,j) \in A, \tag{10}$$

$$x \in \{0,1\}^n, \ z \in \{0,1\}^{n^2}, \ u \in [0,\infty)^n, \tag{11}$$

where $P = (P_{ij})$ for each $(i,j) \in A$ is defined as an input symmetric matrix denoting the connectivity costs between nodes $i, j \in V$. We also define the binary variable z_{ij} to be equal to one if and only if arc $(i,j) \in A$ belongs to the solution of the resulting spanning tree and $z_{ij} = 0$ otherwise.

Proof. Notice that the total connectivity cost is minimized in (1). Next, notice that constraints (2) and (3) ensure that at least k nodes from set V should belong to the output solution of the problem and that the number of arcs must be equal to the number of nodes minus one, respectively. For this purpose, we define the binary variable x_j for each $j \in V$ where $x_j = 1$ if and only if node j is in the solution and $x_j = 0$ otherwise. Subsequently, the constraints (4)–(7) ensure that the solution does not contain subtours. For this purpose, we also define the nonnegative variable u_j for each $j \in V$. To see how these constraints avoid cycles, let us assume that the subset of nodes $\{a_1, a_2, a_3, \ldots, a_t\}$ $(t \leq k)$ is part of the solution tree and that these nodes are connected according to the following sequence $\{a_1 - a_2 - a_3 -, \ldots, -a_t\}$. Then, the set of arcs $\{(a_1, a_2), (a_2, a_3), \ldots, (a_{t-1}, a_t)\}$ is also part of the solution tree and $z_{a_1,a_2} = z_{a_2,a_3} = z_{a_3,a_4} = \cdots = z_{a_{t-1},a_t} = 1$. Also, by constraint (7), we have the following relation with the u_j variables, $u_{a_1} < u_{a_2} < u_{a_3} < \cdots < u_{a_t}$. Notice that, for any node in this sequence, we cannot have any other incoming arc; otherwise, we would violate constraint (6) which ensures that each node cannot have more than one incoming arc if it belongs to the solution. In particular, if node a_1 is the root node, then no incoming arc can be connected to it either since this would imply that $u_{a_t} < u_{a_1}$, which is a contradiction. Finally, notice that any node that belongs to another branch of the resulting tree cannot be connected to any node of any other sequence of nodes forming another branch as this is prevented by constraint (6) too. Consequently, the resulting digraph has to be acyclic. Notice that the constraints (4) and (5) only apply for the nodes which are part of

the solution by restricting the values of each variable u_j, $j \in V$. Also, the degree constraints (8) and (9) ensure that the number of incoming plus outgoing arcs of each node is less than $d \in \{2, \ldots, k-2\}$ where $k \geq 4$. Notice that the degree constraints are redundant if a particular node $j \in V$ is not part of the solution, i.e., if $x_j = 0$. In fact, this also implies that $\sum_{i|(i,j)\in A} z_{ij} + \sum_{i|(j,i)\in A} z_{ji} = 0$. Next, constraint (10) indicates that at most one of the arcs $\{(i,j),(j,i)\} \in A$ can belong to the solution if and only if node $i \in V$ belongs to the solution. Constraint (11) is a domain constraint for the decision variables. Finally, it is observed that an undirected subtree graph can be obtained by dropping the direction of each arc in the resulting digraph, as required. □

Proposition 3. *Constraint (7) is tighter than constraint (35) presented in [2].*

Proof. Constraint (35) in [2] is written as

$$u_j - u_i \geq 1 - n\left(1 - z_{ij}\right), \quad \forall (i,j) \in A. \tag{12}$$

However, these constraints can be equivalently written as

$$u_j - u_i - (n-1)z_{ij} \geq 2 - n, \quad \forall (i,j) \in A. \tag{13}$$

Then, we have that

$$\begin{aligned} u_j - u_i - (n-1)z_{ij} &\geq u_j - u_i - (n-1)z_{ij} \\ &\quad - (n-3)z_{ji} \geq 2 - n, \quad \forall (i,j) \in A. \end{aligned} \tag{14}$$

More details related to constraint (7), when used for the traveling salesman problem and extended to other types of vehicle routing problems, can be consulted in [52]. Notice that model P_1 is a directed graph formulation constructed based on a Miller–Tucker–Zemlin characterization [2, 3, 13]. □

Proposition 4 (see [23]). *In a finite simple undirected graph, the sum of the degrees of every vertex $v \in V$ is twice the number of edges.*

Proposition 4 is known as the handshaking lemma, and it is also valid for directed graphs. In the latter case, the degree of each vertex is simply counted as the sum of incoming plus outgoing arcs. Consequently, this proposition allows us to write the degree constraints (8) and (9) in P_1 equivalently as follows:

$$\sum_{i|(i,j)\in A} z_{ij} + \sum_{i|(j,i)\in A} z_{ji} = \delta_j, \quad \forall j \in V, \tag{15}$$

$$\sum_{j\in V} \delta_j = 2\left(\sum_{j\in V} x_j - 1\right), \tag{16}$$

$$\delta_j \leq \mathrm{d}x_j, \quad \forall j \in V, \tag{17}$$

$$\delta_j \geq x_j, \quad \forall j \in V, \tag{18}$$

$$\delta \in [0, \infty)^n, \tag{19}$$

where δ_j, for each $j \in V$, is a continuous nonnegative variable used to denote the degree of each vertex. Thus, we obtain another MILP model that we denote hereafter by P_1^h. Notice that the variable δ_j, for each $j \in V$, need not be defined as an integer variable. In fact, this is ensured by the left-hand side of constraint (15) and the right-hand side of constraint (16) which are both integer values. Constraints (17) and (18) restrict the value of variable δ_j between 1 and d if and only if node $j \in V$ is part of the solution; otherwise, $\delta_j = 0$. Finally, constraint (19) is a domain constraint for each variable δ_j, $j \in V$.

In order to formulate a flow-based model for the DC*k*MST problem, let $\widehat{H} = (V \cup r, A \cup A_r)$ be an expanded digraph obtained from $H = (V, A)$ where we add to $\widehat{H}$ an artificial root node r and a set of arcs A_r with zero costs from r to every node $v \in V$. The underlying idea is thus to construct an arborescence rooted at r while expanding all nodes in the solution with exactly one arc leaving r. For this purpose, we expand the vector of node and arc variables to $x \in \{0,1\}^{n+1}$ and $z \in \{0,1\}^{(n+1)^2}$, respectively. We also define continuous nonnegative flow variables $f \in [0,\infty)^{(n+1)^2}$, where f_{ij} denotes the amount of flow on arc $(i,j) \in A \cup A_r$. Finally, if we represent the number of nodes in the solution by a nonnegative variable λ, then the idea is to send λ units of flow from r to these nodes, one unit of flow to every one of them. Consequently, a flow-based model can be verified by means of the following proposition.

Proposition 5. *Model P_2 allows to obtain an optimal solution for the DCkMST problem with minimum cost:*

$$P_2: \min_{\{x,z,f,\lambda\}} \sum_{(i,j)\in A} P_{ij} z_{ij} \text{s.t.} \quad \lambda \geq k, \tag{20}$$

$$\sum_{j\in V} x_j = \lambda, \tag{21}$$

$$\sum_{(i,j)\in A} z_{ij} = \lambda - 1, \tag{22}$$

$$\sum_{j|(r,j)\in A_r} f_{rj} = \lambda, \tag{23}$$

$$\sum_{j|(r,j)\in A_r} z_{rj} = 1, \tag{24}$$

$$\sum_{i|(i,j)\in A\cup A_r} f_{ij} - \sum_{i|(j,i)\in A\cup A_r} f_{ji} = x_j, \quad \forall j \in V, \tag{25}$$

$$f_{ij} \leq (n-1)z_{ij}, \quad \forall (i,j) \in A \cup A_r, \tag{26}$$

$$\sum_{i|(i,j)\in A\cup A_r} z_{ij} + \sum_{i|(j,i)\in A\cup A_r} z_{ji} \leq \mathrm{d}x_j, \quad \forall j \in V, \tag{27}$$

$$\sum_{i|(i,j)\in A\cup A_r} z_{ij} + \sum_{i|(j,i)\in A\cup A_r} z_{ji} \geq x_j, \quad \forall j \in V, \tag{28}$$

$$\begin{aligned} z_{ij} + z_{ji} \leq x_i, \quad \forall (i,j) \in A \cup A_r, \\ x_r = 1, \end{aligned} \tag{29}$$

$$x \in \{0,1\}^{n+1},\ z \in \{0,1\}^{(n+1)^2},\ f \in [0,\infty)^{(n+1)^2},\ \lambda \geq 0, \tag{30}$$

where the objective function is the same as for P_1.

Proof. In order to prove the correctness of model P_2, first we make the following observation. □

Observation 1. The number of leaf nodes (nodes with degree equal to one) in any spanning tree graph with $n \geq 3$ vertices is at least 2 and at most $n-1$, corresponding to the cases of a line and a star graph, respectively.

Next, notice that constraints (20) and (21) guarantee that λ is at least k units of flow and that it is equal to the number of active nodes in the resulting subtree, respectively, whilst constraint (22) states that the total number of arcs in the solution should be equal to the number of active nodes minus one. Constraints (23) and (24) ensure that the total amount of flow going out from node r equals λ and that it must be moved through a unique arc $(r,j) \in A_r$, $j \in V$, respectively. Similarly, constraint (25) states that the incoming minus the outgoing flow equals one if node $v \in V$ is part of the solution and equals zero otherwise. Constraint (26) ensures that the flow going out from i to j equals zero if and only if arc $(i,j) \in A \cup A_r$ is not part of the solution. Otherwise, it should be at most $(n-1)$ units. Constraints (27) and (28) are the degree constraints. Notice that the node index in each sum of these constraints applies for all arcs $(i,j) \in A \cup A_r$ in contrast to the node indexes in constraints (8) and (9) which only apply for the arcs $(i,j) \in A$. These degree constraints are valid since Observation 1 implies that the artificial node r can always be connected to a leaf node of the resulting subtree without affecting the maximum degree value d imposed to each node $v \in V$. Next, constraint (29) forces the fact that the root node r is always active. The remaining constraints are the same as for P_1 including constraint (30) which is a domain constraint for the decision variables. Finally, notice that the simultaneous conditions imposed by constraints (22)–(26) ensure that the resulting digraph obtained is a tree [3, 13]. Again, an undirected subtree can be obtained by dropping the directions of the arcs.

Proposition 6. *Constraints (27) and (9) are tighter than constraints (8) and (28), respectively.*

Proof. Let us assume that node $t \in V$ is part of the resulting subtree. Then, we have that

$$d \geq \sum_{i|(i,t)\in A\cup A_r} z_{it} + \sum_{i|(t,i)\in A\cup A_r} z_{ti} = \sum_{i|(i,t)\in A} z_{it} + z_{rt} + \sum_{i|(t,i)\in A} z_{ti} + z_{tr} \geq \sum_{i|(i,t)\in A} z_{it} + \sum_{i|(t,i)\in A} z_{ti} \geq 1. \tag{31}$$

Analogously as for P_1, we can replace the degree constraints (27) and (28) in P_2 by the set of constraints (15)–(19), leading to another MILP formulation we denote hereafter by P_2^h. Notice that, from the constructions of the above formulations, an exponential model can also be written as follows:

$$P_3\colon \min_{\{x,z\}} \sum_{(i,j)\in A} P_{ij} z_{ij} \text{s.t.} \quad \sum_{j\in V} x_j \geq k \sum_{(i,j)\in A} z_{ij} = \sum_{j\in V} x_j - 1, \tag{32}$$

$$\sum_{i|(i,j)\in A} z_{ij} \leq x_j, \quad \forall j \in V, \sum_{(i,j)\in A_S} z_{ij} \leq |S| - 1, \quad \forall S \subseteq V, \tag{33}$$

$$\sum_{i|(i,j)\in A} z_{ij} + \sum_{i|(j,i)\in A} z_{ji} \leq \mathrm{d}x_j, \quad \forall j \in V, \tag{34}$$

$$\sum_{i|(i,j)\in A} z_{ij} + \sum_{i|(j,i)\in A} z_{ji} \geq x_j, \quad \forall j \in V, \tag{35}$$

$$\begin{gathered} z_{ij} + z_{ji} \leq x_i, \quad \forall (i,j) \in A, \\ x \in \{0,1\}^n,\ z \in \{0,1\}^{n^2}, \end{gathered} \tag{36}$$

where inequalities (33) represent subtour elimination constraints. Similarly, as for the above models P_1 and P_2, we replace the degree constraints (34) and (35) in P_3 by the set of constraints (15)–(19). We denote by P_3^h this alternative exponential model. Hereafter, we also denote the corresponding linear programming (LP) relaxations of P_1, P_1^h, P_2, and P_2^h by LP_1, LP_1^h, LP_2, and LP_2^h, respectively. □

4. Proposed Algorithms

In this section, first we present and explain the pseudocode for the exact iterative approach used to solve the exponential models P_3 and P_3^h. Then, for the sake of clarity, we present the modified penalty approach proposed in [30] which allows to obtain feasible solutions for the DCMST problem. Then, we present our VNS and ACO algorithms while using traditional and embedded Q-learning random local search strategies. Finally, we present a pure Q-learning-based algorithm.

4.1. Exact Iterative Algorithm. As mentioned in Section 1, we solve our models P_3 and P_3^h with an exact iterative method adapted from [2, 24]. The method is simple and can be described as follows. It consists of solving first the MILP model P_3 (or P_3^h) without subtour elimination constraints (SECs). Then, new cycles are obtained from the current optimal solution found with a depth-first search procedure [53]. Next, for each cycle found, we write a new constraint of the form (33) and add it to the feasible set of P_3 (or P_3^h). Finally, the model P_3 (or P_3^h) including all the new added inequalities is reoptimized. This process goes on until the current solution found does not contain any cycle. When

this condition is met, it means the optimal solution has been reached. This procedure is depicted in Algorithm 1.

The convergence proof of this method is reported in Theorem 2 in [24]. Notice that, according to Theorem 2 in [24], the solutions obtained within each iteration of Algorithm 1 are lower bounds for the optimal solution of the problem. Further notice that a depth-first search algorithm will always find cycles within each iteration of Algorithm 1 if there exists at least one in the resulting digraph. This fact ensures the convergence of the method in a finite number of iterations (see Property 1 in [24]). We refer the reader to the works in [2, 24] for further details on how we obtain cycles.

4.2. Modified Penalty Approach. In particular, within each iteration of our VNS algorithms, for each random k–tuple of vertices generated, we obtain degree-constrained spanning trees by using a modified penalty approach proposed in [30]. The procedure of this method is depicted in Algorithm 2, and it is explained as follows.

Initially, we find a minimum spanning tree using the Kruskal method [53]. Then, we enter into a while loop doing the following. First, we check if each node satisfies the degree condition. If it is not the case for a particular node, then we update all the weights of the arcs which are incident to it using the formula $P_{ij} = P_{ij} + \theta((P_{ij} - e\text{Min})/(e\text{Max} - e\text{Min}))e\text{Max}$, where θ represents an arbitrary parameter which can be drawn from the interval (0;1) and eMin and eMax are the smallest and largest weights of the current spanning tree obtained, respectively. Subsequently, we symmetrize the updated arcs of matrix P_{ij} in order to avoid choosing previous arcs with opposite directions. Finally, if there still exists a node violating the degree condition, we continue with the process and find a new minimum spanning tree with the Kruskal method. This process continues until a feasible spanning tree is obtained where all nodes satisfy the degree condition. As it can be observed, Algorithm 2 mainly consists of penalizing iteratively the edges (arcs) which are incident to the vertices with degree violations until a feasible spanning tree solution for the problem is obtained.

4.3. VNS Algorithms. We now present our VNS algorithms with the classical near-far random local search strategy and then using an embedded Q-learning strategy.

4.3.1. Classical Random Local Search Strategy. Our VNS procedure is depicted in Algorithm 3. As it can be observed, the method requires an instance of the DCkMST problem and a particular degree with value $d \geq 3$. As in this paper, we only consider degree values of $d \in \{2, 3\}$, then we only solve instances for a degree value of $d = 3$ with this VNS algorithm. The method is simple and can be described as follows. First, it checks if k equals n; if this is the case, then we execute Algorithm 2 using parameters n, P, and d. For this particular case, it means the algorithm finds a feasible solution for the DCMST problem including all nodes of set V. On the opposite, if $k < n$, then we randomly choose an initial k-tuple of vertices from V in order to form the set As. Let Ac be the complement of As.

Next, we construct the submatrix $P\prime$ with the rows and columns of P corresponding to the nodes in As and execute Algorithm 2 using parameters k, $P\prime$, and d. This allows us to save an initial feasible solution together with its objective function value. Notice that the cardinality of set As is k, whilst the cardinality of set Ac is $n - k$. The current sets As and Ac are also saved in $AsOP$ and $AcOP$, respectively. Subsequently, we perform the following steps iteratively, while the CpuTime variable is less than or equal to the maximum CPU time allowed which is controlled with parameter maxTime. Inside this loop, we randomly interchange an element of As with an element of Ac a predefined number of times which is controlled with parameter ind$K$ that is initially set to a value of one. Then, we check if a better solution is obtained. If so, we save the new solution found, its objective function value, update sets $AsOP$ and $AcOP$, and reset the CpuTime variable to zero. Otherwise, we keep the previous best solution found and go back to the interchange step. Notice that parameter ind$K$ controls the number of swap moves the algorithm performs between sets As and Ac. It turns out that this step corresponds to a classical near-far random local search strategy related to VNS algorithm [28, 29]. The larger the number of swap moves, the wider the feasible space being explored. On the opposite, the smaller the number of moves, the closer the solutions which are being tested. In particular, in Algorithm 3, this is handled with ind$K$ variable which is increased by one unit each time a worse solution is obtained. This variable can be increased with up to a predefined value of ind$K$Max − 1. And we reset this parameter to a value of one each time a better solution is obtained in order to restart the search from a local to a wider feasible region. Finally, if ind$K$ equals ind$K$Max, then we also reset ind$K$ to a value of one.

4.3.2. Embedded Q-Learning Strategy. Our Q-learning strategy is simple and consists of embedding a reinforcement learning approach in Algorithm 3 instead of using classical random local search. Notice that the Q-learning approach was initially proposed in [31]. This approach mainly consists of an agent, a set of states S, and a set A of actions per state whereby performing an action $a \in A$; the agent moves from one state to another one according to a reward value, and the goal of the agent is to maximize its total future reward. The potential reward that an agent can obtain is computed as a weighted sum of the expected values of the rewards of all future steps starting from any of the current states. In order to embed this Q-learning approach in Algorithm 3, before entering the inner while loop of Algorithm 3, we need to initialize the matrix variable $Q = Q(\text{ind}KM, \text{ind}KM)$ with zero entries and also to declare the two scalar variables indKA and indicator.

In particular, the value of indicator variable, which can be true or false, is used to perform or not a piece of the code, whereas variables ind$K$ and indKA represent the number of swap moves from one to indKM to be performed between sets As and Ac. Thus, the underlying idea is to move from a

particular number of swap moves to another one by using the aforementioned reinforcement learning strategy. Under this setting, the values of indKA and ind$K$ represent two consecutive states where an agent can be in different moments. For example, consider the case when indKA = 2 and ind$K$ = 8, then we should read it as an agent that is moving from state 2 to state 8. In our problem, it means we are first interchanging 2 elements and then 8 elements between sets As and Ac. Consequently, we can replace the inner code of the **Else** statement corresponding to the **classical random local search strategy** step of Algorithm 3 with the code of Algorithm 4. By doing so, we provide learning capabilities to Algorithm 3 in order to perform the random local search. In Algorithm 4, the parameters λ, μ, and Prob $\in (0;1)$ are the learning rate, discount factor, and probability of moving randomly from one state to another one [31]. Finally, the function expressions Rand, Randint(indKM), and $\text{argmax}_{\{j\in\{1,\ldots,\text{ind}KM\}\}}(Q(\text{ind}KA, j))$ return a random fractional value between zero and one, a random integer value between one and indKM, and the jth column position with current maximum value in matrix Q while moving from state indKA, respectively.

4.4. ACO Algorithms. We now present the ACO algorithms. More precisely, first we present our classical ACO based strategy and then we present an adapted version of the generic ant-Q algorithm which is based on Q-learning strategy [26, 27]. Finally, we present a pure Q-learning-based algorithm. All these methods allow finding feasible solutions for the DCkMST problem while using a degree value of $d = 2$.

4.4.1. Classical Ant Colony Optimization Strategy. Our first procedure is depicted in Algorithm 5.

The method requires an instance of the DCkMST problem. In step one, it first initializes all required parameters and variables. In particular, ρ, α, β, and q denote the pheromone evaporation coefficient, the influence of the pheromone, influence of visibility matrix, and a constant real value, respectively. Each entry of the pheromone matrix $\tau = (\tau_{ij})$ for all $i, j \in V$ represents the amount of pheromone deposited by the ants in the edge $\{i, j\}$ for a state transition going from node i to j. Notice that each of these entries is initialized to a random value between zero and one, which is performed by using the Rand function. Similarly, the visibility matrix $\eta = (\eta_{ij})$ for all $i, j \in V$ represents the desirability value of edge $\{i, j\}$ while choosing the state transition from i to j. In particular, this value is computed by $\eta_{ij} = (1/P_{ij})$. The set of ants is denoted by Ants. The variables CpuTime, iter, LBest, and BestCPU denote the current CPU time, the number of iterations, and best current objective function value found so far and its exact CPU time in which this solution is obtained. Recall that, within ACO algorithms, each ant constructs a tour solution starting from a particular initial node while covering all nodes of the graph. For this purpose, each ant selects the next edge in its tour according to the following probability:

$$p_{ij}^{a} = \frac{(\tau_{ij})^{\alpha}(\eta_{ij})^{\beta}}{\sum_{z\in R_a}(\tau_{iz})^{\alpha}(\eta_{iz})^{\beta}}, \quad (37)$$

where R_a denotes the unvisited set of nodes of ant a. As it can be observed, this probability depends on both the pheromone and visibility values [26].

Subsequently, in step two of Algorithm 5, we enter into a while loop in which we perform the following steps. First, we initialize to a value of zero each entry of matrix $\Delta\tau = (\Delta\tau_{ij}^{a})$, $i, j \in V$, $a \in$ Ants, where each entry in this matrix represents the amount of pheromone deposited by ant $a \in$ Ants. Next, for each ant, we construct a tour T_a starting from a randomly assigned node and compute its length L_a. Then, for each arc in T_a, we deposit the amount of pheromone of ant a. This value is computed by $\Delta\tau_{ij}^{a} = (q/L_a)$. Then, we compute the shortest tour of length k according to T_a and save it as the best tour found so far if its objective function value is less than LBest. For this purpose, we denote by $L_a^k(t)$ the length of k consecutive nodes in T_a starting from node t. Notice that $t = \{1, \ldots, n-k+1\}$. Also notice that the variable CpuTime is reset to a value of zero each time a better solution is obtained. This allows Algorithm 5 to run for another maxTime unit of time with the hope of finding better solutions. Next, we update each arc of the pheromone matrix τ according to parameter ρ and matrix $\Delta\tau$. Finally, when the while loop is finished, we return the best feasible solution obtained together with its objective function value.

4.4.2. Ant-Q-Based Approach. Now, we present our adapted version of the generic Ant-Q algorithm which is based on Q-learning strategy [25, 27]. This procedure is depicted in Algorithm 6.

Similarly to Algorithm 5, in step one of the Ant-Q Algorithm 6, we first initialize the required parameters and variables. In this case, the parameters α and β allow us to weigh the relative importance of learned AQ and visibility values, respectively. Parameter q is a nonnegative constant value, whilst parameters Prob, λ, and μ represent a probability value, a learning step, and a discount factor, respectively. Variables CpuTime, iter, LBest, BestCPU are analogously defined as for Algorithm 5. In particular, matrices $AQ = (AQ(i, j))$ and $\Delta = (\Delta(i, j))$, for all $i, j \in V$, denote the Q-learning matrix and the delayed reinforcement matrices associated with the Ant-Q learning method [31].

In step two of Algorithm 6, we enter into a while loop and perform the following substeps. First, we construct a tour for each ant $a \in$ Ants. This is performed iteratively while choosing a new unvisited node j randomly or according to the following expression:

$$j = \underset{\{r \text{ allowed}\}}{\arg\max}\left\{AQ(T_a(i), r)^{\alpha}\left(\frac{1}{P(T_a(i), r)}\right)^{\beta}\right\}, \quad (38)$$

where $T_a(i)$ denotes the current ith position of ant a in its tour T_a. Then, we update the learning matrix AQ according to all the edges of each ant tour and save the best solution obtained with cardinality k. Next, we proceed with

reinforcement steps that are applied to all the edges belonging to each ant tour and best solution found so far [25, 27]. Finally, the algorithm returns the best feasible solution obtained and its objective function value.

4.5. A Pure Q-Learning-Based Algorithm. Now, we present a pure Q-Learning based approach that allows obtaining feasible solutions for the DCkMST problem while using $d = 2$. This method is depicted in Algorithm 7, and it is described as follows.

Similarly to Algorithm 6, the first step of Algorithm 7 consists of initializing parameters λ, μ, and Prob, which represent the learning rate, discount factor, and a given probability value, respectively. We also define and initialize the required variables CpuTime, iter, LBest, BestCPU, and $Q = (Q(i, j))$ for all $i, j \in V$. These variables allow us to handle the current CPU time, the number of iterations, the best objective function value obtained so far, CPU time of the best current solution found, and the Q-learning matrix, respectively. Next, in step two of Algorithm 7, we iteratively construct a unique tour T with cardinality n and evaluate each subtour composed of k consecutive nodes in T. These steps are performed, while the current CPU time value is lower than the maximum value allowed which is denoted by maxTime. In order to obtain the best subtour of cardinality k from each tour T generated, we let $L^k(t)$ denote the length of k consecutive nodes in T, starting from node t, and let $L^k = \min\{L^k(1), \ldots, L^k(n-k+1)\}$ be the minimum length value obtained. Notice that index $t = \{1, \ldots, n-k+1\}$; otherwise, no subtour of cardinality k can be obtained from T. If a better solution is obtained, we save its length value in LBest, the subtour solution in T^k, the number of iterations in which this new solution is obtained, and reset the current CPU time value to a value of zero. The latter allows the algorithm to search for another maxTime unit of time. Also, notice that the construction of tour T requires to add at each step a new unvisited node u randomly or with maximum value $\{Q(j, u)\}$ where j denotes the last node added to T. For this purpose, we remove node u from V at each step. The unvisited node u is chosen randomly if the Rand function generates a value in the interval $[0; 1]$ which is lower than Prob; otherwise, it is chosen according to the maximum value in the Q-learning matrix. Next, if set V is not empty, we update the corresponding entry in the Q-learning matrix as follows:

$$Q(j, u) = (1 - \lambda)Q(j, u) + \lambda\left(\frac{1}{P(j, u)} + \mu \max_{\{r \in V\}} Q(u, r)\right). \tag{39}$$

Notice that equation (39) is analog to the classical Q-learning algorithm [31]. In particular, the term $(1/P(j, u))$ represents the reward value obtained when going from node j to node $u \in V$. Finally, the algorithm returns the best feasible solution obtained and its objective function value.

5. Numerical Experiments

In this section, we conduct substantial numerical experiments in order to compare all the proposed models and algorithms. For this purpose, we implement Matlab programs using CPLEX 12.7 solver [54] to solve the MILP and LP models. The numerical experiments have been carried out on an Intel(R) 64 bits core (TM) with 3 GHz and 8 G of RAM under Windows 10. CPLEX solver is used with default options. We consider complete graph instances with random uniform and Euclidean distance costs. More precisely, we generate five instances with dimensions of $n = \{50, 100, 200, 300, 400\}$ nodes where each entry in the symmetric matrix $P = (P_{ij})$, $(i, j) \in A$ is randomly drawn from the interval $(0; 100)$. Then, we further consider four Euclidean benchmark instances from TSPLIB [34] referred to as "Berlin52," "gr96," "ch150," and "a280" with dimensions of $n = \{52, 96, 150, 280\}$ nodes, respectively. All these instances are solved for a different number of active nodes $k \leq n$ ranging from $k = 10$ and up to $k = 400$ nodes for the instances with random costs and up to $k = 280$ for the Euclidean ones.

5.1. Numerical Results for the MILP Models. In Tables 1–4, we present numerical results for P_1, P_1^h, P_2, and P_2^h. Notice that all these tables present the same column information. More precisely, column 1 shows the instance number, whilst columns 2 and 3 present the number of nodes of each graph instance and the value of k, respectively. Next, in columns 4–8 and 9–13, we present the optimal or best solution obtained with each model in at most 2 hours of CPU time, number of branch and bound nodes, CPU time in seconds, the optimal solution of each LP relaxation, and its CPU time in seconds. Finally, columns 14 and 15 present gaps that we compute by $[(\text{Opt} - \text{LP})/\text{Opt}]^*100$ where Opt and LP refer to the optimal solution found with the MILP and LP models, respectively. Also, notice that each row in Tables 1 and 3 and in Tables 2 and 4 corresponds to a same instance.

We limit CPLEX to 2 hours of CPU time for the random input graphs, whilst for the Euclidean ones, we limit CPLEX to 1 hour in order to avoid CPLEX shortage of memory events. Consequently, each reported solution is optimal if its CPU time is less than 2 hours. Otherwise, it corresponds to the best solution obtained with CPLEX in at most 2 hours. Subsequently, in Tables 5 and 6, we present numerical results for both P_3 and P_3^h. These two tables also present the same column information. In particular, columns 1–3 present the same information as in Tables 1–4. Next, in columns 4–8 and 9–13, we report the optimal or best solution obtained with Algorithm 1 in at most 1 hour of CPU time, number of branch and bound nodes, CPU time in seconds, number of cycles added to each model, and number of iterations as well. The number of branch and bound nodes, in this case, corresponds to the sum of all branched nodes within each iteration of Algorithm 1. Notice that Theorem 2 in [24] ensures that the solutions obtained within each iteration of Algorithm 1 are lower bounds for the optimal solution of the problem. Consequently, an optimal solution is obtained when the corresponding CPU time is less than 1 hour. Notice that obtaining tight lower bounds is of crucial importance when developing exact methods as it allows to speed up the process significantly. Further notice that

Data: a problem instance of P_3 (or P_3^h)
Result: the optimal solution of P_3 (or P_3^h)
Solve P_3 (resp., P_3^h) without using any constraint of the form (33)
Find cycles on the resulting digraph obtained by using a depth-first search algorithm
While (no cycles remain in the optimal solution of P_3 (resp., P_3^h)) **do**
  For each cycle found, write a new constraint of the form (33) and add it to the feasible set of P_3 (resp., P_3^h)
  Solve P_3 (resp., P_3^h)
Return: the optimal solution found and the objective function value

Algorithm 1: Iterative procedure for solving P_3 (or P_3^h)

Data: a problem instance of the DCMST problem.
Result: a feasible solution for the DCMST problem. Find a minimum spanning tree using the Kruskal method [53]
Set SW = true
While (SW) **do**
 SW = false
 ForEach ($j \in V$) **do**
  Compute the degree of node j and save this value in variable d_j
  If ($d_j > d$) **then**
   SW = true
   Update all incident arcs $(i, j) \in A$ for node j as follows:
   $P_{ij} = P_{ij} + \theta((P_{ij} - e\text{Min})/(e\text{Max} - e\text{Min}))e\text{Max}$
   $P_{ij} = P_{ji}$
 If (SW) **then**
  Find a minimum spanning tree using the Kruskal method [53]
Return feasible solution obtained and its objective function value

Algorithm 2: Modified penalty approach to obtain feasible degree-constrained spanning trees.

models P_1, P_1^h, P_2, and P_2^h obtain upper bounds for the problem if the optimal solution cannot be reached within 2 hours. Thus, we provide an interval where the optimal solution lies.

From Table 1, first we observe that both P_1 and P_1^h allow obtaining the optimal solution of the problem for most of the instances in less than 2 hours. In particular, for the degree value of $d = 3$, we see that P_1 and P_1^h cannot solve the instances #12 and #11-#12, respectively, whilst for $d = 2$, all the instances are solved to optimality. Regarding the CPU times, for $d = 3$, we see that P_1^h requires a higher amount of CPU time, whilst for $d = 2$, we obtain similar values with both models. Next, we observe that the number of branch and bound nodes is slightly lower for P_1 than for P_1^h. We also see that the LP bounds are near-optimal and exactly the same for both models which are confirmed by the gaps. Notice that all the LP relaxations are solved in less than 3 seconds. Finally, we observe that the CPU time values required to solve both models increase with k.

Regarding the Euclidean graph instances presented in Table 2, we observe that most of them cannot be solved to optimality in one hour. We mention that these instances proved to be very difficult to solve, and this is the main reason we limit CPLEX to one hour in order to avoid CPLEX shortages of memory. The difficulty in solving these instances is also reflected in the number of branch and bound nodes and in the gaps obtained which are significantly higher compared to Table 1. Notice that these gaps decrease with k in which evidences solving the DCkMST problem is harder to solve than the classical DCMST problem, at least for these instances. Next, we observe that the CPU time values of the LP relaxations are higher than those reported in Table 1 and that the degree values do not seem to affect the performance of each model. We further see that the optimal or best objective function values obtained for the instance "ch150" are significantly higher for $d = 2$. Finally, we observe that the objective function values obtained with P_1 for the instances #7–#10, #12 and #6–#9, and #11 are lower than those obtained with P_1^h for $d = 3$ and $d = 2$, respectively. However, the opposite occurs for the instances #6, #11 and #10, #12 while using $d = 3$ and $d = 2$, respectively.

From Table 3, we mainly observe that, for $d = 3$, only P_2 allows obtaining the optimal solution of the problem for all the instances in less than 2 hours. Notice that P_2^h only failed to find a feasible solution for the instance #10, whilst for $d = 2$, neither P_2 nor P_2^h can solve all the instances to optimality. However, P_2 can still solve more instances than P_2^h. Regarding the number of branch and bound nodes and LP times, we observe smaller and slightly higher values than in Table 1, respectively. Finally, we observe that the LP bounds are near-optimal which is confirmed by the gaps. In Table 4, we observe that P_2 and P_2^h can solve more instances to optimality than P_1 and P_1^h in Table 2, respectively. However, in this case, both P_2 and P_2^h cannot find a feasible solution for some of the instances. Next, we observe smaller values for the branch and bound nodes and CPU times with similar

```
Data: an instance of the DCkMST problem using degree d = 3.
Result: a feasible solution and its objective function value.
If (k = n) then
   Execute Algorithm 2 using parameters (n, P, d)
   Save feasible solution found and its objective function value
Else
   Classical random local search strategy:
   Generate an initial random k–tuple of vertices. Let As denote this set of vertices and Ac its complement, P' = P(As, As)
   Execute Algorithm 2 using parameters (k, P', d)
   Save the initial feasible solution found and its objective function value
  AsOP = As, AcOP = Ac
  indK = 1, cont = 0, CpuTime = 0
   While (CpuTime ≤ maxTime) do
     iter = iter + 1
      For i = 1 to indK do
         Interchange randomly an element of As with an element of Ac
     P' = P(As, As), execute Algorithm 2 using parameters (k, P', d)
      If (a better solution is obtained) then
         Save the new solution and set AsOP = As, AcOP = Ac
       iterOp = iter, cont = 0, indK = 1, Optime = Optime + CpuTime, CpuTime = 0
      Else
       cont = cont + 1, As = AsOP, Ac = AcOP
      If (cont ≥ 1) then
       cont = 0
        If (indK < indKMax) then
          indK = indK + 1
        Else
          indK = 1
Return best feasible solution obtained and its objective function value
```

ALGORITHM 3: VNS algorithm for the DCkMST problem using classical random local search strategy.

```
Generate an initial random k–tuple of vertices. Let As denote this set of vertices and Ac its complement P' = P(As, As)
Execute Algorithm 2 using parameters (k, P', d)
Save the initial feasible solution found and its objective function value
AsOP = As, AcOP = Ac
indK = 1, indKA = 1, Q = zeros(indKM, indKM), indicator = false, CpuTime = 0
While (CpuTime ≤ maxTime) do
   iter = iter + 1
   For i = 1 to indK do
      Interchange randomly an element of As with an element of Ac
   Set P' = P(As, As) and run Algorithm 2 using parameters (k, P', d)
   If (a better solution is obtained) then
      Save the new solution and set AsOP = As, AcOP = Ac, indicator = true
      iterOp = iter, Optime = Optime + CpuTime, CpuTime = 0
   Else
      As = AsOP, Ac = AcOP
   If (indicator) then
      Q(indKA, indK) = (1 − λ)Q(indKA, indK) + λ(Reward + μ max_{j∈{1,...,indKM}} (Q(indK, j)))
   indKA = indK
   If (Rand < Prob) then
      indK = Randint(indKM)
   Else
      indK = argmax_{j∈{1,...,indKM}} (Q(indKA, j))
   indicator = false
```

ALGORITHM 4: Embedded Q-learning code.

Data: an instance of the DCkMST problem using degree $d = 2$.
Result: a feasible solution and its objective function value.
Step 1
Initialize parameters ρ, α, β, q and the set of ants (Ants)
CpuTime = 0, iter = 0, LBest = ∞, BestCPU = 0
τ_{ij} = rand, $\forall i, j \in V$
Step 2
While (CpuTime $\leq$ maxTime) **do**
  iter = iter + 1
  $\Delta\tau_{ij}^a = 0, \forall i, j \in V, a \in$ Ants
  ForEach $a \in$ Ants **do**
    Construction of each ant tour
    Randomly assign an initial position $j \in V$ for the ant a according to graph G and construct a tour T_a using all remaining nodes of G.
    Find best solution
    Compute the length L_a of T_a
    Denote by $L_a^k(t)$ the length of k consecutive nodes in T_a starting from node t
    Compute $L_a^k = \min\{L_a^k(1), \ldots, L_a^k(n-k+1)\}$
    If (LBest $> L_a^k$) **then**
      Set LBest = L_a^k and save the best tour of length k in T^k
      iterOp = iter, BestCPU = BestCPU + CpuTime, CpuTime = 0
    Update accumulated pheromone matrix
    ForEach $(i, j) \in T_a$ **do**
      $\Delta\tau_{ij}^a = (q/L_a)$, $\Delta\tau_{ji}^a = (q/L_a)$
  Update pheromone matrix
  ForEach $(i, j) \in A$ **do**
    $\tau_{i,j} = (1-\rho)\tau_{i,j} + \sum_{a\in \text{Ants}} \Delta\tau_{ij}^a$
Return best feasible solution found and its objective function value

Algorithm 5: Classical ACO algorithm for the DC kMST problem.

orders of magnitude for the LP relaxations when compared to Table 2. Finally, we observe that the LP bounds are not tight which is again confirmed by the gap columns.

From Table 5, we mainly observe that models P_3 and P_3^h, which are both solved with Algorithm 1, allow to solve to optimality the instances #1–#9 and #1–#12 for $d = 3$ and $d = 2$, respectively. In particular, we see that, for $d = 3$, it is harder to solve these instances than for $d = 2$. This fact is reflected in the number of branch and bound nodes, CPU times of the LP relaxations, number of iterations, and number of cycles added to each exponential model which are clearly lower when $d = 2$. Notice that, for the instances #10–#12 and degree value of $d = 3$, we only report the best objective function values obtained which are in fact lower bounds. Regarding the Euclidean instances reported in Table 6, we observe that the instances #1–#3, #12 and #1–#8, #11-#12 are all solved to optimality for $d = 3$ and $d = 2$, respectively. We also see in Table 6 that it is harder to solve the instances for $d = 3$ than for $d = 2$. Notice that, for $d = 2$, we almost solve all the instances to optimality with the exception of instances #9 and #10 for which we obtain lower bounds. Again, this observation can be verified by looking at the number of branch and bound nodes, CPU times of the LP relaxations, number of iterations, and number of cycles added to each proposed model which are smaller when $d = 2$. Finally, from the numerical results presented in Tables 1–6, we can conclude that model P_1 outperforms the other ones as it allows to obtain either an optimal or a best upper bound for most of the tested instances. Notice that the flow and exponential models also have good performance in terms of optimality, but for the large-size instances, the flow models deteriorate rapidly, whilst the exponential ones cannot be handled efficiently in terms of CPU times. Consequently, in Tables 7 and 8, we present numerical results for P_1 for large random and Euclidean input graph instances with up to 400 and 280 nodes, respectively. In particular, in Table 7, these numerical results are reported for $d = 3$, whereas in Table 8, these numerical results are obtained while using a degree value of $d = 2$.

From Tables 7 and 8, we observe similar trends as for the above Tables 1–4. More precisely, we observe that P_1 can solve almost all random input graph instances to optimality in less than 2 hours using both degree values. Only the instance number #8 could not be solved to optimality although an upper bound is reported for this particular instance. In contrast, none of the Euclidean instances can be solved to optimality in less than 1 hour of CPU time. For these instances, only upper bounds are reported too. The difficulty in solving the Euclidean instances can also be observed in the gap columns which report significantly higher values as a consequence of the LP bounds obtained. On the opposite, the LP bounds obtained for the random graph instances are near-optimal. Finally, we observe from both Tables 7 and 8 that the number of branch and bound nodes is significantly smaller for $d = 2$ than for $d = 3$ and that the LP times are slightly higher for $d = 2$.

5.2. Numerical Results for VNS Algorithms. In Tables 9 and 10, we present numerical results obtained with the proposed

Data: an instance of the DCkMST problem using degree $d = 2$.
Result: a feasible solution and its objective function value.
Step 1
Initialize parameters α, β, q, Prob, λ, μ, and the set of ants (Ants)
CpuTime = 0, iter = 0, LBest = ∞, BestCPU = 0
$AQ(i,j) = 0$, $\Delta(i,j) = 0$, $\forall i, j \in V$
Step 2
While (CpuTime $\leq$ maxTime) **do**
  iter = iter + 1
  Construct ant tours
  ForEach $a \in$ Ants **do**
    Randomly choose a node $j \in V$
    $T_a = \emptyset$, $T_a = T_a \cup \{j\}$
  For $i = 1$ to $n-1$ **do**
    ForEach $a \in$ Ants **do**
      If (Rand < Prob) **then**
        Randomly choose an unvisited node of the ant and add it to T_a.
      Else
        $j = \text{argmax}_{\{r \text{ allowed}\}} \{AQ(T_a(i), r)^{\alpha} (1/P(T_a(i), r))^{\beta}\}$
        $T_a = T_a \cup \{j\}$
      Update AQ-values
      $AQ(T_a(i), T_a(i+1)) = (1-\lambda)AQ(T_a(i), T_a(i+1)) + \lambda\mu\max_{\{j \text{ allowed}\}}(AQ(T_a(i+1), j))$
      $AQ(T_a(i+1), T_a(i)) = AQ(T_a(i), T_a(i+1))$
  Find best solution
  ForEach $a \in$ Ants **do**
    Denote by $L_a^k(t)$ the length of k consecutive nodes in T_a starting from node t
    Compute $L_a^k = \min\{L_a^k(1), \ldots, L_a^k(n-k+1)\}$
    If (LBest $> L_a^k$) **then**
      Set LBest $= L_a^k$ and save the best tour of length k in T^k
      iterOp = iter, BestCPU = BestCPU + CpuTime, CpuTime = 0
    Reinforcement of ant tours
    ForEach $(i,j) \in T_a$ **do**
      $\Delta(i,j) = (q/L_a^k)$, $\Delta(j,i) = (q/L_a^k)$
  Reinforcement of best solution
  ForEach $(i,j) \in T^k$ **do**
    $\Delta(i,j) = (q/L\text{Best})$, $\Delta(j,i) = (q/L\text{Best})$
  Delayed reinforcement of AQ values
  ForEach $a \in$ Ants **do**
    $V = \{1, \ldots, n\}$
    For $i = 1$ to $n-1$ **do**
      $V = V - \{T_a(i)\}$, $V = V - \{T_a(i+1)\}$
      If $V \neq \emptyset$ **then**
        $AQ(T_a(i), T_a(i+1)) = (1-\lambda)AQ(T_a(i), T_a(i+1)) + \lambda(\Delta(T_a(i), T_a(i+1)) + \mu\max\{AQ(T_a(i+1), V)\})$
        $AQ(T_a(i+1), T_a(i)) = AQ(T_a(i), T_a(i+1))$
Return best feasible solution found and its objective function value

Algorithm 6: Ant-Q algorithm for the DCkMST problem.

VNS algorithms for both random and Euclidean input graph instances while using a degree value of $d = 3$. Recall that our second VNS approach consists of replacing the code lines of Algorithm 3 corresponding to the random local search strategy with the code of Algorithm 4 which represents the embedded Q-learning strategy. Hereafter, we denote by VNS_R and VNS_Q these two VNS approaches. In Algorithms 3 and 4, we arbitrarily set the maximum CPU time allowed to a value of maxTime = 250 seconds, whereas the input parameters ind$K$Max and indKM are set to a value of 10. Similarly, the required parameters of Algorithm 4 were calibrated to $\lambda = 0.25$, $\mu = 0.25$, and Prob = 0.15. Finally, the reward value and parameter θ of Algorithm 2 were set to Reward = 1 and $\theta = 0.1$, respectively.

In particular, in Table 9, we report numerical results for the instances presented in Tables 1–6, whereas in Table 10, we report numerical results for the large-size instances presented in Table 7. In both tables, columns 4–15 and 2–13 contain exactly the same column information, respectively. More precisely, in Table 10, columns 1 to 3 present the instance number, number of nodes, and the value of k. Next, in columns 4 and 5, we report the minimum objective function and CPU time values reported for each instance in Tables 1–4. Subsequently, in columns 6–10 and 11–15, we

Data: an instance of the DCkMST problem using degree $d = 2$.
Result: a feasible solution and its objective function value.
Step 1
Initialize parameters λ, μ, Prob
CpuTime = 0, iter = 0, LBest = ∞, BestCPU = 0, $Q(i, j) = 0$, $\forall i, j \in V$
Step 2
While (CpuTime $\leq$ maxTime) **do**
  iter = iter + 1
  Construct a unique tour
  Randomly choose a node $j \in V$
  $T = \emptyset$, $T = T \cup \{j\}$, $V = V - \{j\}$
  For $i = 1$ to $n - 1$ **do**
    If(Rand < Prob)**then**
      Randomly choose a node $u \in V$
    Else
      $u = \text{argmax}_{\{r \in V\}}\{Q(j, r)\}$
    $T = T \cup \{u\}$, $V = V - \{u\}$
    If $V \neq \emptyset$ **then**
      $Q(j, u) = (1 - \lambda)Q(j, u) + \lambda((1/P(j, u)) + \mu\max_{\{r \in V\}}Q(u, r))$
    $j = u$
  Find best solution
  Denote by $L^k(t)$ the length of k consecutive nodes in T starting from node t
  Compute $L^k = \min\{L^k(1), \ldots, L^k(n - k + 1)\}$
  If (LBest > L^k) **then**
    Set LBest = L^k and save the best tour of length k in T^k
    iterOp = iter, BestCPU = BestCPU + CpuTime, CpuTime = 0
Return best feasible solution found and its objective function value

ALGORITHM 7: Pure Q-learning-based approach for the DCkMST problem.

TABLE 1: Numerical results obtained with P_1 and P_1^h for random graphs using $d = 3$ and $d = 2$.

#	n	k	P_1					P_1^h					Gaps	
			Opt	$B\&Bn$	Time	LP	Time	Opt	$B\&Bn$	Time	LP	Time	Gap_1 %	Gap_1^h %
							$d = 3$							
1	50	10	46.86	75	0.80	43.38	0.17	46.86	109	1.06	43.38	0.19	7.43	7.43
2		20	113.97	498	0.75	109.46	0.20	113.97	362	0.73	109.46	0.19	3.96	3.96
3		30	204.35	1470	2.79	199.05	0.19	204.35	1911	3.48	199.05	0.17	2.59	2.59
4		50	459.09	1005	1.76	453.49	0.19	459.09	2901	5.74	453.49	0.17	1.22	1.22
5	100	25	115.44	23	1.95	114.63	0.48	115.44	25	1.79	114.63	0.42	0.69	0.69
6		50	263.95	2501	8.30	262.04	0.47	263.95	1872	6.43	262.04	0.42	0.73	0.73
7		75	454.44	3186	5.04	451.58	0.42	454.44	1715	24.23	451.58	0.42	0.63	0.63
8		100	749.75	36160	202.21	745.09	0.45	749.75	65193	338.15	745.09	0.44	0.62	0.62
9	200	50	162.55	1035	10.39	160.88	1.73	162.55	1462	10.86	160.88	1.64	1.03	1.03
10		100	378.43	80814	506.06	374.87	1.67	378.43	94564	518.29	374.87	1.65	0.94	0.94
11		150	652.86	2944093	6417.84	648.89	1.73	652.86	2635764	7200	648.89	1.95	0.61	0.61
12		200	1049.25	403853	7200	1044.74	2.26	1049.25	215415	7200	1044.74	1.78	0.43	0.43
							$d = 2$							
1	50	10	52.22	49	1.58	46.30	0.17	52.22	129	0.61	46.30	0.17	11.34	11.34
2		20	134.08	237	2.06	129.71	0.22	134.08	904	2.57	129.71	0.22	3.26	3.26
3		30	240.30	45	1.26	239.32	0.20	240.30	9	1.26	239.32	0.20	0.41	0.41
4		50	564.37	2948	7.38	557.08	0.19	564.37	1953	7.47	557.08	0.17	1.29	1.29
5	100	25	123.71	404	4.63	121.54	0.48	123.71	148	3.99	121.54	0.45	1.75	1.75
6		50	287.78	1093	37.88	285.31	0.61	287.78	1875	7.75	285.31	0.50	0.86	0.86
7		75	529.56	0	1.92	529.31	0.50	529.56	1	3.99	529.31	0.48	0.05	0.05
8		100	915.29	683	9.75	914.63	0.48	915.29	77	8.14	914.63	0.45	0.07	0.07
9	200	50	184.29	0	12.28	184.29	2.11	184.29	0	7.02	184.29	2.03	0	0
10		100	437.08	6424	99.08	435.45	2.50	437.08	2568	93.76	435.45	2.28	0.37	0.37
11		150	775.89	275	94.08	774.76	2.40	775.89	90	42.20	774.76	2.20	0.15	0.15
12		200	1285.32	190	67.75	1284.58	2.23	1285.32	459	98.28	1284.58	1.89	0.06	0.06

Table 2: Numerical results obtained with P_1 and P_1^h for Euclidean graphs using $d = 3$ and $d = 2$.

#	n	k	P_1					P_1^h					Gaps	
			Opt	*B&Bn*	Time	LP	Time	Opt	*B&Bn*	Time	LP	Time	Gap_1 %	Gap_1^h %
							$d = 3$							
1	Berlin52	10	274.46	10	2.25	253.21	0.34	274.46	60	0.94	253.21	0.28	7.74	7.74
2		20	926.77	543991	218.23	801.28	0.20	926.77	591996	300.65	801.28	0.19	13.54	13.54
3		30	1935.71	4246933	3600	1666.19	0.20	1935.71	5292832	3600	1666.19	0.19	13.92	13.92
4		52	6081.63	2024616	3600	5567.13	0.20	6081.63	1688601	3600	5567.13	0.19	8.46	8.46
5	gr96	25	74.14	488206	3600	40.18	0.41	74.14	452709	3600	40.18	0.39	45.81	45.81
6		50	163.11	513293	3600	117.66	0.55	162.80	462902	3600	117.66	0.42	27.86	27.73
7		75	269.15	593506	3600	222.47	0.48	269.23	672775	3600	222.47	0.41	17.34	17.37
8		96	436.65	1192367	3600	384.51	0.80	436.69	1168784	3600	384.51	0.70	11.94	11.95
9	ch150	50	1607.24	243342	3600	1258.47	0.95	1610.93	157353	3600	1258.47	0.94	21.70	21.88
10		80	2657.21	186474	3600	2303.05	1.01	2694.88	154768	3600	2303.05	0.90	13.33	14.54
11		100	3457.86	199914	3600	3058.71	0.97	3441.59	129318	3600	3058.71	0.92	11.54	11.13
12		150	5914.15	215787	3600	5522.97	0.95	5923.78	234887	3600	5522.97	0.89	6.61	6.77
							$d = 2$							
1	Berlin52	10	279.23	0	0.52	261.79	0.30	279.23	0	0.53	261.79	0.22	6.25	6.25
2		20	1015.39	2253	7.08	862.04	0.25	1015.39	8735	15.91	862.04	0.22	15.10	15.10
3		30	2065.12	9446	12.21	1871.76	0.28	2065.12	543	9.36	1871.76	0.27	9.36	9.36
4		52	6968.77	1699	8.24	6545.08	0.27	6968.77	1244	7.44	6545.08	0.23	6.08	6.08
5	gr96	25	74.59	380670	3600	45.19	0.77	74.59	324849	3600	45.19	2.39	39.42	39.42
6		50	168.53	270415	3600	128.73	0.92	174.25	194752	3600	128.73	8.45	23.62	26.12
7		75	283.14	154838	3600	248.44	13.60	288.49	310429	3600	248.44	0.58	12.25	13.88
8		96	474.95	168539	3600	452.14	9.66	476.08	397540	3600	452.14	0.72	4.80	5.03
9	ch150	50	1753.83	98742	3600	1356.91	1.04	1807.83	86310	3600	1356.91	9.91	22.63	24.94
10		80	2917.72	73829	3600	2489.25	2.29	2881.72	60210	3600	2489.25	10.64	14.69	13.62
11		100	3744.47	66710	3600	3302.54	15.88	4013.58	54518	3600	3302.54	14.05	11.80	17.72
12		150	6411.73	62743	3600	6158.67	1.15	6397.12	70574	3600	6158.67	1.01	3.95	3.73

Table 3: Numerical results obtained with P_2 and P_2^h for random graphs using $d = 3$ and $d = 2$.

#	n	k	P_2					P_2^h					Gaps	
			Opt	*B&Bn*	Time	LP	Time	Opt	*B&Bn*	Time	LP	Time	Gap_2 %	Gap_2^h %
							$d = 3$							
1	50	10	46.86	42	2.29	43.38	0.23	46.86	340	1.25	43.38	0.20	7.43	7.43
2		20	113.97	2161	16.65	109.46	0.25	113.97	999	11.42	109.46	0.27	3.96	3.96
3		30	204.35	116	1.67	197.61	0.19	204.35	2024	14.65	197.61	0.19	3.30	3.30
4		50	459.09	123	1.00	444.77	0.25	459.09	629	3.87	447.87	0.20	3.12	2.44
5	100	25	115.44	358	46.71	114.54	0.53	115.44	97	54.60	114.54	0.70	0.78	0.78
6		50	263.95	1243	73.63	261.02	0.51	263.95	668	77.50	261.02	0.75	1.11	1.11
7		75	454.44	459	143.16	449.51	0.55	454.44	525	111.53	449.51	0.67	1.08	1.08
8		100	749.75	24	22.56	728.70	2.29	749.75	0	40.36	734.84	0.59	2.81	1.99
9	200	50	162.55	479	3995.62	160.36	2.21	162.55	920	2835.21	160.36	3.29	1.35	1.35
10		100	378.43	479	2671.16	373.28	3.81	—	—	7200	373.28	3.68	1.36	—
11		150	652.86	479	2097.25	642.84	4.71	652.86	465	1663.29	642.84	3.49	1.53	1.53
12		200	1049.25	607	505.27	1030.04	30.11	1049.25	502	1490.46	1034.23	3.21	1.83	1.43
							$d = 2$							
1	50	10	52.22	5	1.01	46.42	0.25	52.22	190	1.73	46.30	0.20	11.12	11.34
2		20	134.08	379	3.43	129.90	0.33	134.08	1515	21.95	129.71	0.20	3.12	3.26
3		30	240.30	235	7.35	239.32	0.28	240.30	736	19.08	239.32	0.20	0.41	0.41
4		50	564.37	488	20.20	557.48	0.28	564.37	1647	42.00	557.08	0.19	1.22	1.29
5	100	25	123.71	1067	248.12	121.55	1.50	123.71	1933	237.10	121.54	0.70	1.74	1.75
6		50	287.78	1676	632.83	285.35	2.84	287.78	857	356.21	285.31	0.81	0.84	0.86
7		75	529.56	0	8.88	529.31	3.23	529.56	1050	639.40	529.31	0.76	0.05	0.05
8		100	—	—	7200	914.60	4.13	—	—	7200	914.63	0.89	—	—
9	200	50	184.29	10	228.02	184.29	13.13	—	—	7200	184.29	3.81	0	—
10		100	—	—	7200	435.45	26.49	—	—	7200	435.45	6.61	—	—
11		150	—	—	7200	774.76	55.04	—	—	7200	774.76	6.62	—	—
12		200	1285.32	1045	1408.71	1284.58	94.68	—	—	7200	1284.58	4.15	0.06	—

—: no solution found.

Table 4: Numerical results obtained with P_2 and P_2^h for Euclidean graphs using $d = 3$ and $d = 2$.

#	n	k	P_2 Opt	P_2 B&Bn	P_2 Time	P_2 LP	P_2 Time	P_2^h Opt	P_2^h B&Bn	P_2^h Time	P_2^h LP	P_2^h Time	Gaps Gap_2 %	Gaps Gap_2^h %
							$d = 3$							
1	Berlin52	10	274.46	0	0.97	253.21	0.21	274.46	0	0.82	253.21	0.21	7.74	7.74
2		20	926.77	0	2.25	697.51	0.25	926.77	0	2.15	697.51	0.21	24.74	24.74
3		30	1935.71	175	2.31	1414.09	0.22	1935.71	427	6.12	1414.09	0.20	26.95	26.95
4		52	6081.63	556	2.00	4822.43	0.28	6081.63	95	2.20	4923.77	0.19	20.71	19.04
5	gr96	25	72.40	2518	268.87	38.47	0.51	72.40	913	301.42	38.47	0.61	46.87	46.87
6		50	162.80	3274	316.75	110.48	0.56	162.80	1816	250.78	110.46	0.60	32.14	32.15
7		75	267.57	591	109.28	210.19	1.86	267.57	615	157.00	210.18	0.68	21.44	21.45
8		96	436.23	480	70.35	354.60	3.72	436.23	985	42.30	357.83	0.60	18.71	17.97
9	ch150	50	1567.77	1734	921.27	1219.69	1.21	—	—	3600	1219.69	1.46	22.20	—
10		80	—	—	3600	2237.22	1.25	—	—	3600	2237.22	1.85	—	—
11		100	—	—	3600	2974.08	1.60	3399.17	1037	1009.76	2974.08	1.97	—	12.51
12		150	5882.45	982	212.27	5339.18	27.69	5882.45	974	245.39	5353.54	1.61	9.24	8.99
							$d = 2$							
1	Berlin52	10	279.23	0	2.95	262.40	0.25	279.23	0	0.67	261.79	0.20	6.03	6.25
2		20	1015.39	1487	21.75	868.22	0.37	1015.39	84	3.21	862.04	0.19	14.49	15.10
3		30	2065.12	0	4.55	1874.56	0.34	2065.12	0	2.87	1871.76	0.22	9.23	9.36
4		52	6968.77	972	31.65	6554.14	0.28	6968.77	0	2.89	6545.08	0.20	5.95	6.08
5	gr96	25	73.56	1592	845.82	45.41	0.77	73.56	2525	621.64	45.19	0.62	38.27	38.57
6		50	168.53	862	299.41	129.00	2.81	168.53	601	350.55	128.73	0.67	23.46	23.62
7		75	283.14	1235	408.99	248.89	4.12	283.14	1672	447.78	248.44	0.64	12.10	12.25
8		96	474.95	2874	329.41	456.19	2.50	474.95	2482	315.00	452.14	0.56	3.95	4.80
9	ch150	50	—	—	3600	1358.73	16.52	—	—	3600	1356.91	2.46	—	—
10		80	3167.66	1241	3600	2491.52	22.79	—	—	3600	2489.25	2.34	21.35	—
11		100	5425.41	1224	3600	3306.74	20.47	—	—	3600	3302.54	2.79	39.05	—
12		150	7176.67	3698	3600	6166.49	29.53	—	—	3600	6158.67	1.78	14.08	—

—: no solution found.

report for each VNS approach, the initial and minimum objective function values, CPU time in seconds required by VNS, number of iterations, and gaps, respectively. The gaps are computed by [(VNS – Best)/Best]*100.

From both Tables 9 and 10, first we observe that the objective function values of the initial solutions are significantly higher than the best ones. The latter clearly evidences the effectiveness of both VNS approaches. Notice that when $k = n$, the problem reduces to the classical DCMST problem. Consequently, for these instances, the solutions are obtained only with Algorithm 2. Next, we observe that the CPU times required by both VNS methods are larger for the random instances than for the Euclidean ones. From Table 9, next, we observe that the solutions obtained with VNS_Q outperform those obtained with VNS_R for both random and Euclidean instances. Although, in general, we see that both VNS procedures allow obtaining near-optimal solutions which is confirmed by the gap columns. On the opposite, we see that the objective function values reported in Table 10 are not tight for the random instances when compared to those obtained by the MILP models. But still, in this case, the gaps reported for VNS_Q are significantly better than for VNS_R. However, for the Euclidean instances reported in Table 10, we see that both VNS approaches allow us to obtain better solutions than the MILP models. The latter can be verified by the negative gaps which show that the objective function values obtained with VNS algorithms are significantly lower than those obtained with the MILP models.

5.3. Numerical Results for ACO Algorithms. Now, we report numerical results obtained with Algorithms 5 and 6 for random and Euclidean instances while using a degree value of $d = 2$. More precisely, in Table 11, we report numerical results for the instances presented in Tables 1–6. However, in Table 12, we report numerical results for the large-size instances presented in Table 8. Hereafter, we denote by ACO_R and ACO_Q the two ant colony optimization approaches presented in Algorithms 5 and 6, respectively. Recall that the first one is constructed based on the classical ACO metaheuristic [26]. However, the latter is based on the Ant-Q algorithm proposed in [25, 27]. In Algorithm 5, we calibrated the input parameters to $\alpha = 0.5$, $\beta = 2$, $\rho = 0.9$, and $q = 1$. However, in Algorithm 6, we calibrated the input parameters to $\alpha = 1$, $\beta = 4$, $q = 1$, Prob = 0.1, $\lambda = 0.15$, and $\mu = 0.15$. Finally, in both Algorithms 5 and 6, we arbitrarily set the maximum CPU time and number of ants to maxTime = 250 seconds and |Ants| = 20, respectively.

The legend of Table 11 is as follows. From columns 1 to 3, we present the instance number, the number of nodes of the input graph, and the value of k. Next, in columns 4 and 5, we present the minimum objective function and CPU time values reported in Tables 1–6. We repeat this information for the sake of clarity. Next, in columns 6–10 and 11–15, we report the objective function values of the initial solutions, the objective function values of the best solutions, CPU time in seconds, number of iterations, and gaps which are computed by [(ACO – Best)/Best]*100, respectively.

Table 5: Lower bounds obtained with P_3 and P_3^h for random graphs using $d = 3$ and $d = 2$.

#	n	k	P_3 Opt	B&Bn	Time	#Cycles	#Iter	P_3^h Opt	B&Bn	Time	#Cycles	#Iter
						$d = 3$						
1	50	10	46.86	53	1.75	3	4	46.86	36	2.15	3	4
2		20	113.97	0	1.33	2	3	113.97	2	1.01	2	3
3		30	204.35	1021	5.62	14	14	204.35	755	5.63	14	14
4		50	459.09	955	6.40	23	22	459.09	611	5.99	23	22
5	100	25	115.44	53	4.01	2	3	115.44	22	2.90	2	3
6		50	263.95	3373	24.62	19	20	263.95	3394	26.10	19	20
7		75	454.44	7154	58.03	43	44	454.44	7558	59.75	43	44
8		100	749.75	214640	441.97	211	211	749.75	157118	446.80	211	211
9	200	50	162.55	8	23.45	4	5	162.55	15	23.78	4	5
10		100	378.24	1438706	3604.81	252	252	378.20	1241277	3624.11	237	237
11		150	651.49	1509749	3626.19	263	263	651.43	1214061	3614.95	239	239
12		200	1046.96	852808	3602.98	335	334	1046.95	1036504	3615.80	320	319
						$d = 2$						
1	50	10	52.22	6	0.97	1	2	52.22	0	0.87	1	2
2		20	134.08	11	1.61	4	4	134.08	17	1.56	4	4
3		30	240.30	26	1.31	1	2	240.30	10	1.08	1	2
4		50	564.37	0	0.52	3	2	564.37	0	0.47	3	2
5	100	25	123.71	166	3.56	1	2	123.71	36	2.95	1	2
6		50	287.78	499	7.19	6	5	287.78	222	6.04	6	5
7		75	529.56	0	0.90	0	1	529.56	0	0.84	0	1
8		100	915.29	0	4.07	4	5	915.29	0	3.34	4	5
9	200	50	184.29	0	4.32	0	1	184.29	0	4.04	0	1
10		100	437.08	3274	31.73	4	4	437.08	1075	30.84	4	4
11		150	775.89	0	18.53	1	2	775.89	0	19.20	1	2
12		200	1285.32	0	7.05	1	2	1285.32	0	6.43	1	2

Similarly, the first three columns of Table 12 report the instance number and the best objective function values and CPU times reported in Table 8. Again, this information is repeated for comparison purposes. Finally, the legends of columns 4–8 and 9–13 are exactly the same as for the columns 6–10 and 11–15 in Table 11.

From Table 11, we observe that the best solutions obtained with both ACO approaches are near-optimal and far from the initial solutions obtained. These facts prove the effectiveness of the proposed methods. Next, we see that the CPU times are lower for ACO_R than for ACO_Q for most of the instances and in particular for the larger ones. Regarding the number of iterations, in general, we observe similar orders of magnitude for both methods. Finally, we observe that the gaps obtained with ACO_R are tighter than those obtained with ACO_Q for both the random and Euclidean instances. In particular, for large instances, these values are significantly better. Finally, notice that we obtain negative gaps for some of the large Euclidean instances. This clearly evidences that the solutions obtained with the ACO methods outperform significantly the solutions obtained with the MILP models.

From Table 12, we mainly observe that the solutions obtained with both ACO algorithms are significantly worse than those reported in Table 11 for random instances. On the opposite, for all the Euclidean instances, the solutions obtained with the ACO approaches are significantly better than those obtained with P_1 which is again confirmed by the negative gaps obtained.

5.4. Numerical Results for QL Algorithm. In Tables 13 and 14, we present numerical results obtained with Algorithm 7 (denoted as QL) for both random and Euclidean instances while using a degree value of $d = 2$. In Tables 13 and 14, we report numerical results for the same instances presented in Tables 1–6 and for the instances of Table 8, respectively. The legend of Table 13 is as follows: columns 1–3 present the instance number, number of nodes (or name of the Euclidean instance), and the value of k, respectively. Notice that the name of each Euclidean instance indicates at the end the number of nodes it contains. For example, the instance name "Berlin52" has 52 nodes. Next, columns 4 and 5 report the best solution and minimum CPU time in seconds obtained with the MILP models. Finally, columns 6–10 report the initial solution obtained with Algorithm 7, its best solution found, CPU time in seconds, number of iterations, and gaps obtained which are computed by $[(\text{QL} - \text{Best})/\text{Best}]^{*}100$, respectively. The legend of Table 14 is exactly the same as for Table 13 and is obtained by removing columns 2 and 3 from Table 13.

From Table 13, first we observe that the initial solutions obtained with Algorithm 7 are considerably worse than those obtained with the ACO ones. However, we also see that the best solutions obtained with it are near-optimal and competitive with the ACO methods. Notice that this fact is relevant as it clearly shows the effectiveness of Algorithm 7 which is mainly based on its learning capability and simplicity. Concerning the gaps reported in Table 13, we observe that, for some random and Euclidean instances, Algorithm 7

Table 6: Lower bounds obtained with P_3 and P_3^h for Euclidean graphs using $d = 3$ and $d = 2$.

#	n	k	P_3 Opt	B&Bn	Time	#Cycles	#Iter	P_3^h Opt	B&Bn	Time	#Cycles	#Iter
						$d = 3$						
1	Berlin52	10	274.46	0	1.44	4	4	274.46	2	1.12	4	4
2		20	926.77	28584	50.95	81	80	926.77	42931	62.09	81	80
3		30	1935.71	432870	650.35	298	294	1935.71	384635	710.19	298	294
4		52	6081.63	1522843	4720.06	937	932	6076.57	1261869	3611.00	860	854
5	gr96	25	68.14	2782639	3619.00	169	92	68.07	2507913	3621.48	161	84
6		50	154.06	1878243	3716.32	196	65	153.89	1740625	3616.42	190	62
7		75	265.11	1188745	3614.37	297	152	264.86	1113079	3626.85	274	129
8		96	427.94	434160	3602.41	734	638	427.94	380249	3606.05	733	637
9	ch150	50	1545.75	1487346	3642.21	172	79	1545.23	1307916	3697.27	167	75
10		80	2599.80	1451730	3659.13	212	77	2598.40	1327931	3776.47	202	72
11		100	3388.09	859090	3682.26	226	108	3387.08	797571	3617.01	216	98
12		150	5882.45	133462	2909.89	499	419	5882.45	127032	2874.88	499	419
						$d = 2$						
1	Berlin52	10	279.23	0	0.51	2	2	279.23	0	0.67	2	2
2		20	1015.39	64	2.47	15	5	1015.39	62	2.09	15	5
3		30	2065.12	134	2.23	15	4	2065.12	63	2.42	15	4
4		52	6968.77	0	0.73	10	3	6968.77	0	0.75	10	3
5	gr96	25	73.56	689177	1525.83	125	51	73.56	764555	1832.40	125	51
6		50	168.53	189277	981.32	109	36	168.53	303696	1584.56	109	36
7		75	283.14	24803	100.75	63	18	283.14	15880	96.28	63	18
8		96	474.95	11951	23.93	34	10	474.95	4802	17.58	34	10
9	ch150	50	1637.17	904211	3620.84	112	35	1636.05	989940	4001.45	106	31
10		80	2788.87	564918	3790.60	127	32	2786.68	539699	3697.03	123	30
11		100	3627.93	28619	279.88	66	14	3627.93	36800	341.06	66	14
12		150	6368.79	8152	43.71	38	10	6368.79	9046	45.69	38	10

Table 7: Numerical results obtained with P_1 for large input graphs using $d = 3$.

#	n	k	P_1 Opt	B&Bn	Time	LP	Time	Gap_1%
				Random graph instances				
1	300	50	118.82	120	40.30	117.39	4.40	1.20
2		100	283.63	3719	64.73	281.70	4.20	0.68
3		200	703.42	994913	2022.93	701.15	4.21	0.32
4		300	1334.61	177538	7200	1332.44	5.01	0.16
5	400	50	110.20	1651	193.66	108.48	8.47	1.56
6		100	236.59	10088	2316.85	234.51	9.48	0.88
7		200	522.61	13577	259.07	520.66	8.24	0.37
8		400	1459.53	31763	7200	1456.34	10.84	0.22
				Euclidean graph instances				
1	a280	50	480.35	23222	3600	384.00	4.21	20.06
2		100	1011.07	42957	3600	784.00	4.52	22.46
3		150	1316.17	28761	3600	1184.16	4.57	10.03
4		280	2493.68	46232	3600	2385.76	3.85	4.33

allows obtaining tighter gaps than ACO_Q in less CPU time. Finally, we observe that the number of iterations required by Algorithm 7 is significantly larger than the ACO methods. This can be explained by the fact that each iteration of Algorithm 7 requires a considerable less computational effort.

From Table 14, we observe that the distance between the initial and best solutions obtained with Algorithm 7 is even larger than for the instances presented in Table 13. This confirms again that the learning capability of Algorithm 7 is effective. Next, we see that the CPU time required by Algorithm 7 is significantly lower than the amount required by ACO_Q for random instances. We also see that the gaps reported in Table 14 are tighter than those reported in Table 12 for the ACO_Q approach for all random instances. In particular, we see that, for the random instances #2 and #3 in

Table 8: Numerical results obtained with P_1 for large input graphs using $d = 2$.

#	n	k	P_1 Opt	*B&Bn*	Time	LP	Time	Gap_1%
Random graph instances								
1	300	50	132.92	0	34.03	132.87	6.91	0.04
2		100	321.64	454	309.74	320.82	8.30	0.26
3		200	820.70	272	520.03	820.02	12.60	0.08
4		300	1623.85	1250	1301.40	1623.16	11.51	0.04
5	400	50	123.02	413	2003.07	120.05	19.05	2.42
6		100	265.68	30587	3073.49	262.44	19.19	1.22
7		200	598.20	2527	2448.10	595.08	34.01	0.52
8		400	1818.03	999	7200	1797.76	16.21	1.12
Euclidean graph instances								
1	a280	50	534.97	13685	3600	384.00	5.48	28.22
2		100	1133.49	23212	3600	784.00	4.93	30.83
3		150	1417.20	11405	3600	1186.71	6.16	16.26
4		*200*	2991.96	11036	3600	2513.31	4.66	16.00

Table 9: Numerical results obtained with VNS algorithms for random and Euclidean instances using $d = 3$.

#	n	k	MILP		VNS_R					VNS_Q				
			Best	Time (s)	VNS_{ini}	VNS	Time (s)	#Iter	Gap_R	VNS_{ini}	VNS	Time (s)	#Iter	Gap_Q
Input graphs with random uniform costs														
1	50	10	46.86	0.80	254.76	48.27	6.06	13842	3.00	208.73	46.86	0.62	1620	0
2		20	113.97	0.73	266.68	118.53	22.80	7213	4.00	353.48	116.07	8.29	2796	1.84
3		30	204.35	1.67	338.03	206.35	46.51	3674	0.97	326.75	205.43	30.49	2503	0.52
4		50	459.09	1.00	463.14	463.14	0.08	1	0.88	463.14	463.14	0.08	1	0.88
5	100	25	115.44	1.79	352.14	115.44	115.10	23826	0	413.40	118.77	18.59	8854	2.88
6		50	263.95	6.43	564.30	269.18	61.89	2962	1.98	517.86	265.42	86.41	7382	0.55
7		75	454.44	5.04	600.65	465.48	197.61	2681	2.42	659.23	456.59	215.46	4098	0.47
8		100	749.75	22.56	749.87	749.87	0.25	1	0.01	749.87	749.87	0.24	1	0.01
9	200	50	162.55	10.39	555.42	179.98	1857.83	63640	10.72	463.99	169.38	477.84	18728	4.20
10		100	378.43	506.06	725.35	395.98	2928.53	12324	4.63	763.28	386.12	1657.52	9194	2.03
11		150	652.86	1663.29	883.28	721.07	1250.91	912	10.44	926.60	670.36	1130.82	872	2.68
12		200	1049.25	505.27	1056.34	1056.34	3.98	1	0.67	1056.34	1056.34	3.90	1	0.67
Input graphs with Euclidean distance costs														
1	Berlin52	10	274.46	0.82	1976.48	274.46	1.41	5959	0	1645.70	274.46	0.09	470	0
2		20	926.77	2.15	3923.04	926.77	2.69	7382	0	3161.01	926.77	0.36	820	0
3		30	1935.71	2.31	4961.30	2104.32	5.29	5133	8.71	4484.23	2071.97	178.34	193048	7.03
4		52	6081.63	2	6081.63	6081.63	0.01	1	0	6081.63	6081.63	0.00	1	0
5	gr96	25	72.40	268.87	204.00	79.28	122.54	230336	9.50	247.07	74.14	271.89	508583	2.40
6		50	162.80	250.78	307.92	179.50	74.78	34094	10.25	260.74	163.70	2.74	1163	0.55
7		75	267.57	109.28	388.13	269.88	20.97	3049	0.86	372.46	267.57	141.16	21176	0
8		96	436.23	42.30	436.23	436.23	0.02	1	0	436.23	436.23	0.01	1	0
9	ch150	50	1567.77	921.27	3441.77	1716.09	348.67	154734	9.46	3290.91	1696.06	37.20	16107	8.18
10		80	2657.21	3600	4327.97	2792.69	115.65	13889	5.09	4480.37	2771.08	223.80	21675	4.28
11		100	3399.17	1009.76	4722.64	3527.70	408.36	15366	3.78	4745.73	3499.01	288.25	13858	2.93
12		150	5882.45	212.27	5882.45	5882.45	0.12	1	0	5882.45	5882.45	0.12	1	0

Table 14, the gaps obtained by Algorithm 7 are smaller than those reported for both ACO methods in Table 12. Regarding the Euclidean instances, we observe that the gaps obtained are still competitive when compared to those reported for the ACO methods. In fact, we also obtain negative gaps which means we outperform the best solutions obtained with the MILP models in less than 1 h. Finally, from Table 14, we observe that the gap obtained for the Euclidean instance #4 outperforms both gaps reported for the ACO methods in Table 12.

Table 10: Numerical results obtained with VNS algorithms for the instances of Table 7 using $d = 3$.

#	MILP		VNS_R					VNS_Q				
	Best	Time (s)	VNS_{ini}	VNS	Time (s)	#Iter	Gap_R	VNS_{ini}	VNS	Time (s)	#Iter	Gap_Q
Input graphs with random uniform costs												
1	118.82	40.30	534.32	139.36	1330.83	47429	17.28	510.03	127.26	494.26	36291	7.10
2	283.63	64.73	762.97	338.44	2549.43	7665	19.32	775.47	298.83	1893.42	8481	5.35
3	703.42	2022.93	1088.49	850.68	1381.55	326	20.93	1000.36	750.05	3851.54	966	6.62
4	1334.61	7200	1349.38	1349.38	32.68	1	1.10	1349.38	1349.38	32.70	1	1.10
5	110.20	193.66	492.26	126.27	1376.58	52156	14.58	526.08	124.87	1015.61	63720	13.31
6	236.59	2316.85	783.62	306.03	1465.56	8242	29.35	677.53	257.46	3628.70	20296	8.82
7	522.61	259.07	1129.36	696.20	2587.27	664	33.21	1058.37	635.21	4130.04	1045	21.54
8	1459.53	7200	1485.59	1485.59	83.45	1	1.78	1485.59	1485.59	83.13	1	1.78
Input graphs with Euclidean distance costs												
1	480.35	3600	1095.00	446.91	314.41	96332	−6.96	1076.54	440.38	74.39	31647	−8.32
2	1011.07	3600	1385.72	897.55	425.80	11551	−11.22	1447.17	890.15	195.38	8251	−11.95
3	1316.17	3600	1774.88	1304.78	786.14	8339	−0.86	1689.48	1303.48	570.31	5417	−0.96
4	2493.68	3600	2440.94	2440.94	7.94	1	−2.11	2440.94	2440.94	7.94	1	−2.11

Table 11: Numerical results obtained with ACO algorithms for random and Euclidean instances using $d = 2$.

#	n	k	MILP		ACO_R					ACO_Q				
			Best	Time (s)	ACO_{ini}	ACO	Time (s)	#Iter	Gap_R	ACO_{ini}	ACO	Time (s)	#Iter	Gap_Q
Input graphs with random uniform costs														
1	50	10	52.22	0.61	93.69	52.22	54.78	5576	0	259.26	52.22	89.89	2083	0
2		20	134.08	1.61	175.53	135.75	24.58	2551	1.24	955.85	135.75	126.17	2111	1.24
3		30	240.30	1.08	370.28	244.31	26.01	2648	1.66	435.68	245.24	234.25	4437	2.05
4		50	564.37	0.47	684.50	584.18	5.53	561	3.51	2486.65	588.31	48.69	803	4.24
5	100	25	123.71	2.95	163.55	123.71	2.06	67	0	1231.26	123.71	192.91	995	0
6		50	287.78	6.04	337.69	293.90	29.20	975	2.12	2546.91	298.88	339.66	1731	3.85
7		75	529.56	0.84	650.54	550.41	46.00	1533	3.93	3804.55	563.89	220.28	1164	6.48
8		100	915.29	3.34	1093.20	970.43	50.66	1699	6.02	5188.97	1021.63	239.28	1230	11.61
9	200	50	184.29	4.04	279.89	193.34	168.47	1626	4.91	2495.55	198.36	698.71	1093	7.63
10		100	437.08	30.84	546.76	467.76	68.13	657	7.01	4856.80	492.90	258.63	408	12.77
11		150	775.89	18.53	941.43	836.89	9.54	90	7.86	7444.72	870.56	273.53	460	12.20
12		200	1285.32	6.43	1734.02	1480.76	15.91	156	15.20	10277.04	1614.61	321.30	529	25.61
Input graphs with Euclidean distance costs														
1	Berlin52	10	279.23	0.51	2195.31	279.23	389.57	42096	0	4581.09	279.23	21.44	351	0
2		20	1015.39	2.09	1270.76	1015.39	2.01	201	0	9757.74	1020.02	72.24	1189	0.45
3		30	2065.12	2.23	2757.23	2065.12	58.28	6258	0	12888.36	2069.90	32.44	535	0.23
4		52	6968.77	0.73	9738.24	7156.65	17.71	1895	2.69	22432.48	7234.27	6.47	106	3.80
5	gr96	25	73.56	621.64	112.78	74.59	144.67	5647	1.40	144.72	74.80	7.65	45	1.68
6		50	168.53	299.41	219.39	172.38	97.18	3811	2.28	546.20	180.29	161.80	984	6.97
7		75	283.14	96.28	379.18	292.20	24.97	966	3.19	609.61	302.66	50.22	302	6.89
8		96	474.95	17.58	619.16	497.06	66.02	2578	4.65	944.82	534.30	320.22	1956	12.49
9	ch150	50	1753.83	3600	2050.39	1678.93	38.58	672	−4.27	18522.68	1722.83	222.95	613	−1.76
10		80	2881.72	3600	3667.84	2843.87	169.47	2972	−1.31	27659.44	2929.44	141.14	394	1.65
11		100	3627.93	279.88	4833.86	3788.26	98.18	1718	4.41	35594.85	3773.76	325.01	908	4.01
12		150	6368.79	43.71	7566.16	6720.31	42.55	742	5.51	52740.44	7005.90	113.50	315	10.00

5.5. Average Numerical Results for the Proposed Algorithms. In order to give more insights with respect to the behavior of the proposed algorithms, in Figures 2 and 3, we report average upper bounds and CPU times in seconds obtained with both VNS_R and VNS_Q for different values of k. More precisely, we randomly generate 20 large-size instances using $n = 300$ nodes for each value of k in each figure. In particular, in Figure 2, these averages are reported for instances

Table 12: Numerical results obtained with ACO algorithms for random and Euclidean instances of Table 8 using $d = 2$.

#	MILP		ACO_R					ACO_Q				
	Best	Time (s)	ACO_{ini}	ACO	Time (s)	#Iter	Gap_R	ACO_{ini}	ACO	Time (s)	#Iter	Gap_Q
Input graphs with random uniform costs												
1	132.92	34.03	193.95	146.53	26.59	98	10.23	2228.39	158.67	1443.44	1091	19.37
2	321.64	309.74	419.46	367.16	43.30	165	14.15	4620.75	396.94	589.52	468	23.41
3	820.70	520.03	980.56	898.25	64.56	237	9.44	9596.31	923.88	918.36	703	12.57
4	1623.85	1301.40	2122.15	1933.14	128.82	495	19.04	14537.58	2107.74	1130.55	865	29.79
5	123.02	2003.07	172.02	136.33	98.30	210	10.81	2581.64	143.35	2080.93	917	16.52
6	265.68	3073.49	328.39	298.32	155.47	338	12.28	5243.13	316.62	1187.53	530	19.17
7	598.20	2448.10	754.03	683.36	6.10	13	14.23	10356.11	724.67	1247.05	563	21.14
8	1818.03	7200	2370.22	2224.40	5.61	11	22.35	20229.56	2631.20	1151.90	521	44.72
Input graphs with Euclidean distance costs												
1	534.97	3600	426.30	401.87	113.06	632	−24.87	675.36	405.20	227.45	196	−24.25
2	1133.49	3600	990.04	846.67	96.56	539	−25.30	2944.45	919.30	3.48	3	−18.89
3	1417.20	3600	1554.94	1311.62	42.31	236	−7.44	2909.14	1358.90	312.40	270	−4.11
4	2991.96	3600	3477.83	2983.92	2.18	11	−0.26	6137.08	3239.96	292.15	255	8.28

Table 13: Numerical results obtained with Q-learning algorithm for random and Euclidean instances using $d = 2$.

#	n	k	MILP		QL				
			Best	Time (s)	QL_{ini}	QL	Time (s)	#Iter	Gap
Input graphs with random uniform costs									
1	50	10	52.22	0.61	427.26	52.22	9.03	42359	0
2		20	134.08	1.61	794.00	138.00	8.76	40622	2.92
3		30	240.30	1.08	1473.09	245.62	2.25	10079	2.21
4		50	564.37	0.47	2567.98	601.37	15.38	70922	6.55
5	100	25	123.71	2.95	1207.44	123.71	4.75	9890	0
6		50	287.78	6.04	2440.56	291.89	98.12	211891	1.42
7		75	529.56	0.84	3473.78	565.78	487.90	1067899	6.83
8		100	915.29	3.34	4975.56	1024.77	30.10	64345	11.96
9	200	50	184.29	4.04	2543.82	199.27	130.44	122671	8.12
10		100	437.08	30.84	5027.12	476.04	123.87	118105	8.91
11		150	775.89	18.53	7342.28	843.60	38.57	36512	8.72
12		200	1285.32	6.43	9799.77	1546.62	227.21	214107	20.32
Input graphs with Euclidean distance costs									
1	Berlin52	10	279.23	0.51	2692.71	279.23	39.93	200322	0
2		20	1015.39	2.09	12651.13	1020.02	151.28	774386	0.45
3		30	2065.12	2.23	16299.43	2070.65	45.42	228363	0.26
4		52	6968.77	0.73	30795.84	7234.27	85.36	433901	3.80
5	gr96	25	73.56	621.64	858.80	74.59	78.73	199419	1.40
6		50	168.53	299.41	1774.49	180.05	89.15	225194	6.83
7		75	283.14	96.28	2500.70	303.35	260.64	648302	7.13
8		96	474.95	17.58	3352.78	552.06	191.68	477430	16.23
9	ch150	50	1753.83	3600	18911.28	1711.33	125.94	186763	−2.42
10		80	2881.72	3600	28695.92	2949.71	202.88	306120	2.35
11		100	3627.93	279.88	37580.42	3953.79	107.61	160624	8.98
12		150	6368.79	43.71	54468.42	7122.77	565.44	844020	11.83

using random costs, whereas in Figure 3, these average values are reported for Euclidean instances.

From Figures 2 and 3, we mainly observe that the average upper bounds obtained with VNS_Q are significantly smaller than those obtained with VNS_R. This fact clearly shows that the embedded Q-learning strategy of VNS_Q outperforms the classical near-far local search approach. Next, we further notice that the average CPU times are significantly smaller for VNS_Q than for VNS_R when solving the Euclidean instances. On the opposite, the VNS_Q approach requires a significantly

Table 14: Numerical results obtained with Q-learning algorithm for the random and Euclidean instances of Table 8 using $d = 2$.

#	MILP		QL				
	Best	Time (s)	QL_{ini}	QL	Time (s)	#Iter	Gap
Input graphs with random uniform costs							
1	132.92	34.03	2455.40	147.03	116.82	72924	10.61
2	321.64	309.74	5203.22	347.51	101.90	63021	8.04
3	820.70	520.03	10356.83	897.52	160.04	99831	9.36
4	1623.85	1301.40	15547.51	1990.10	441.38	263922	22.55
5	123.02	2003.07	2173.05	137.42	155.63	64597	11.70
6	265.68	3073.49	5346.13	307.22	442.60	183130	15.63
7	598.20	2448.10	10535.56	691.97	286.76	115717	15.67
8	1818.03	7200	20088.17	2287.40	305.80	126631	25.81
Input graphs with Euclidean distance costs							
1	534.97	3600	6193.93	426.46	170.69	119614	−20.28
2	1133.49	3600	11619.78	909.03	82.05	57482	−19.80
3	1417.20	3600	18065.40	1441.78	275.34	192437	1.73
4	2991.96	3600	33966.22	2800.73	130.98	91758	−6.39

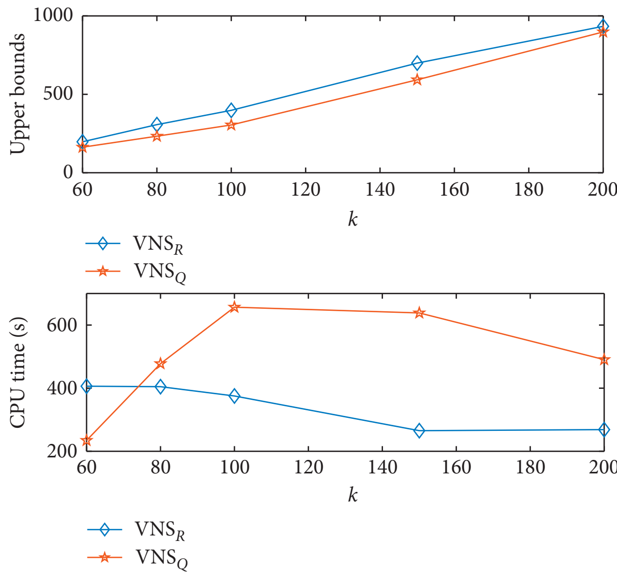

Figure 2: Average upper bounds and CPU times in seconds obtained with VNS_R and VNS_Q while varying k for random complete graph instances of size $n = 300$ using $d = 3$.

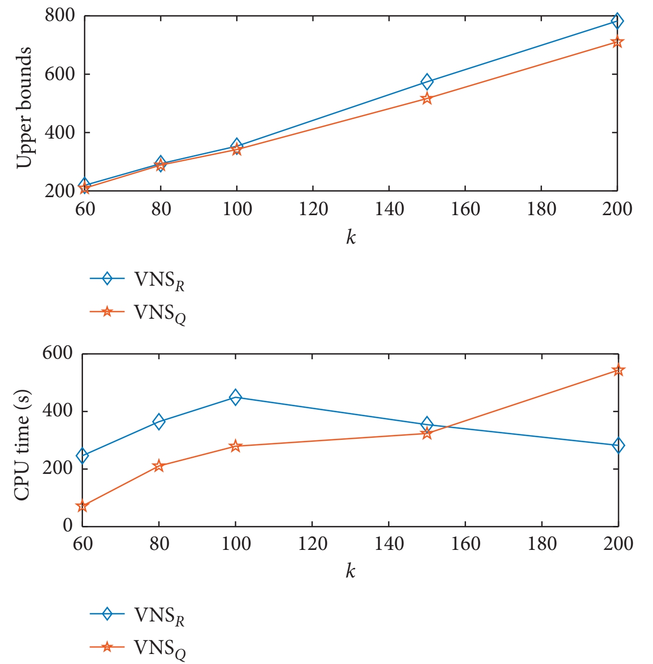

Figure 3: Average upper bounds and CPU times in seconds obtained with VNS_R and VNS_Q while varying k for Euclidean complete graph instances of size $n = 300$ using $d = 3$.

higher CPU time effort when solving instances with random costs. Similarly, in Figures 4 and 5, we report average upper bounds and CPU times in seconds for ACO_R, ACO_Q, and QL algorithms while varying k. For this purpose, again, we randomly generate 20 large-size instances of $n = 300$ nodes for each value of k. In particular, in Figure 4, these averages are reported for instances with random costs. However, in Figure 5, these values are reported for Euclidean instances.

From Figures 4 and 5, we mainly observe that the average upper bounds obtained with ACO_Q are slightly larger than those obtained by QL. In turn, the average upper bounds obtained with QL are larger than those obtained with ACO_R. We further notice that this trend is more evident when solving graph instances with Euclidean distance costs. On the opposite, these bounds seem to be closer for the instances with random costs. Finally, we observe that the average CPU time values are significantly smaller for ACO_R than for QL and ACO_Q. Similarly, QL requires less CPU time than ACO_Q. In conclusion, we observe that ACO_R outperforms both QL and ACO_Q. However, the QL method outperforms the ACO_Q approach.

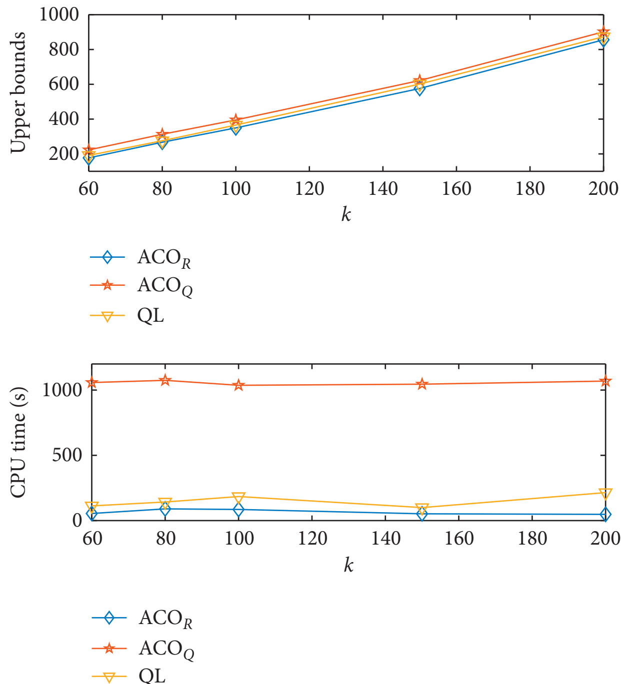

FIGURE 4: Average upper bounds and CPU times in seconds obtained with ACO$_R$, ACO$_Q$, and QL while varying k for random complete graph instances of size $n = 300$ using $d = 2$.

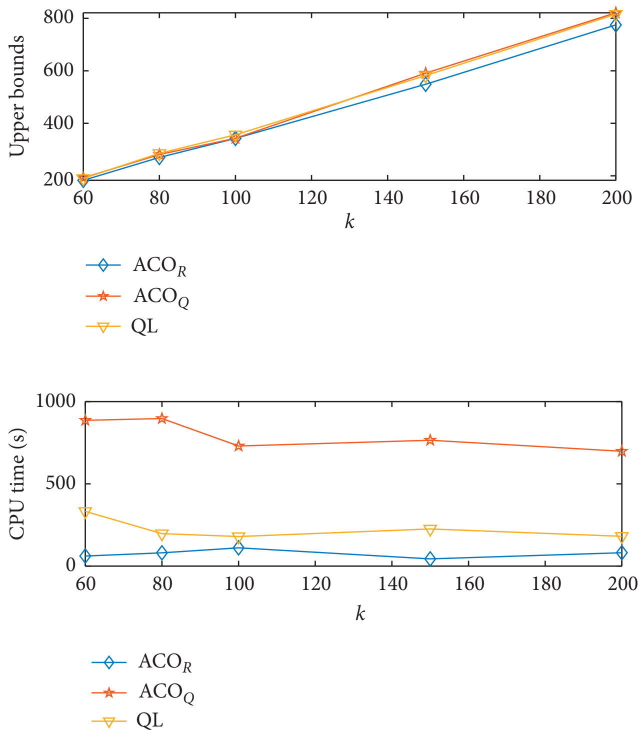

FIGURE 5: Average upper bounds and CPU times in seconds obtained with ACO$_R$, ACO$_Q$, and QL while varying k for Euclidean complete graph instances of size $n = 300$ using $d = 2$.

6. Conclusions

In this paper, we consider the degree-constrained k-cardinality minimum spanning tree problem which emerges as a combination of two classical combinatorial optimization problems, namely, the degree-constrained and k-minimum spanning tree problems. One can see from the literature that this problem has not been studied in depth yet. This leads us to propose three mixed-integer linear programming models for which we derive equivalent formulations by using the handshaking lemma. In order to obtain near-optimal solutions, we further propose ant colony optimization, variable neighborhood search, and a pure Q-learning-based algorithm. In particular, for each proposed metaheuristic, we obtain new algorithms while embedding a Q-learning strategy. We conduct substantial numerical experiments using benchmark input graph instances from TSPLIB and randomly generated ones with random uniform and Euclidean distance costs with up to 400 nodes. From the numerical results obtained, our main conclusions can be listed as follows:

(1) We observe that, in general, the proposed models which are constructed based on Miller–Tucker–Zemlin-constrained approach are more robust than the flow ones as they allow to obtain optimal or best upper bounds for all the instances. In particular, we see that the flow models allow us to solve to optimality Euclidean instances with up to 100 nodes, which is not possible to achieve with the other models. However, the flow models cannot provide an upper bound for some of the instances with higher dimensions. Similarly, we observe that our proposed exponential models do also show good performance in terms of optimality, but again they cannot be handled efficiently when solving large-size instances. However, they provide an interval where the optimal solution lies. We further conclude that it is not evident to decide whether the performance of the proposed models improves or deteriorates while using the handshaking lemma. Finally, we observe that the Miller–Tucker–Zemlin-based models allow us to obtain optimal solutions for instances with up to 400 nodes while using random costs and degree values of $d \in \{2, 3\}$. On the opposite, they cannot solve all the Euclidean instances to optimality.

(2) Concerning the VNS algorithms, we observe that the objective function values obtained with both VNS algorithms are significantly lower when compared to their initial solutions obtained. This clearly evidences the effectiveness of our VNS approaches. In general, the two VNS procedures allow obtaining near-optimal solutions and even better solutions than CPLEX for the large-size instances. Next, we see that the CPU times required by these algorithms are larger for the random instances than for the Euclidean ones. Notice that Euclidean instances are

significantly harder to solve by the MILP models. We can also conclude that the embedded Q-learning strategy in our VNS algorithm allows us to obtain better solutions than the classical near-far random local search strategy. This suggests that the construction of optimization methods with learning capabilities in order to make them robust, self-adaptive, and independent from decision-makers is worth to be further investigated.

(3) Regarding the ACO algorithms, we observe that both methods allow obtaining near-optimal solutions which certainly prove their effectiveness. In general, we obtain better solutions for the Euclidean instances than for the random ones. In particular, the quality of the Euclidean solutions obtained improves significantly for large-size instances of the problem for which both ACO methods obtain better solutions than CPLEX. Next, we observe that the solutions obtained with ACO_R are better in terms of quality when compared to the solutions obtained with ACO_Q for both random and Euclidean instances of the problem. Concerning our pure Q-learning approach proposed to solve instances with degree $d = 2$, we observe that the solutions obtained for random and Euclidean instances are competitive with those obtained with the ACO methods. In particular, we see that the CPU time required by this method is significantly lower than the amount required by ACO_Q. Then, we further see that, for some of the instances, the gaps obtained by the pure Q-learning approach are tighter than those obtained with ACO_Q. We also obtain better solutions than CPLEX solver which is used to solve the MILP model. Finally, notice that this pure Q-learning method is simple and versatile, and then, it can be adapted to any other combinatorial optimization problem in a straightforward manner.

As future research, we plan to propose new formulations related to the degree-constrained k-minimum spanning tree problem with applications on network design problems.

Data Availability

The data used to support the findings of the study are available upon request to the corresponding author.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

The authors acknowledge the financial support from FONDECYT (nos. 11180107 and 3190147).

References

[1] P. Adasme, "Optimal sub-tree scheduling for wireless sensor networks with partial coverage," *Computer Standards & Interfaces*, vol. 61, pp. 20–35, 2019.

[2] P. Adasme, "*p*-median based formulations with backbone facility locations," *Applied Soft Computing*, vol. 67, pp. 261–275, 2018.

[3] P. Adasme, R. Andrade, J. Leung, and A. Lisser, "Improved solution strategies for dominating trees," *Expert Systems with Applications*, vol. 100, pp. 30–40, 2018.

[4] P. Adasme and A. Dehghan Firoozabadi, "Facility location with tree topology and radial distance constraints," *Complexity*, vol. 2019, Article ID 9723718, 29 pages, 2019.

[5] B. Ahlgren, M. Hidell, and E. C.-H. Ngai, "Internet of things for smart cities: interoperability and open data," *IEEE Internet Computing*, vol. 20, no. 6, pp. 52–56, 2016.

[6] R. Al-Zaidi, J. C. Woods, M. Al-Khalidi, and H. Hu, "Building novel VHF-based wireless sensor networks for the internet of marine things," *IEEE Sensors Journal*, vol. 18, no. 5, pp. 2131–2144, 2018.

[7] Z. Chu, F. Zhou, Z. Zhu, R. Q. Hu, and P. Xiao, "Wireless powered sensor networks for internet of things: maximum throughput and optimal power allocation," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 310–321, 2018.

[8] R. B. Dial, "An efficient algorithm for building min-path trees for all origins in a multi-class network," *Transportation Research Part B: Methodological*, vol. 40, no. 10, pp. 851–856, 2006.

[9] H. Hua, L. Hovestadt, P. Tang, and B. Li, "Integer programming for urban design," *European Journal of Operational Research*, vol. 274, no. 3, pp. 1125–1137, 2019.

[10] I. Khan, F. Belqasmi, R. Glitho, N. Crespi, M. Morrow, and P. Polakos, "Wireless sensor network virtualization: a survey," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 553–576, 2016.

[11] E. Bulut and I. Korpeoglu, "Sleep scheduling with expected common coverage in wireless sensor networks," *Wireless Networks*, vol. 17, no. 1, pp. 19–40, 2011.

[12] T. Yardibi and E. Karasan, "A distributed activity scheduling algorithm for wireless sensor networks with partial coverage," *Wireless Networks*, vol. 16, no. 1, pp. 213–225, 2010.

[13] P. Adasme, R. Andrade, and A. Lisser, "Minimum cost dominating tree sensor networks under probabilistic constraints," *Computer Networks*, vol. 112, pp. 208–222, 2017.

[14] M. Cardei, D. MacCallum, and X. Cheng, "Wireless sensor networks with energy efficient organization," *Journal of Interconnection Networks*, vol. 3, pp. 3-4, 2002.

[15] H. Gupta, Z. Zhou, S. R. Das, and Q. Gu, "Connected sensor cover: self-organization of sensor networks for efficient query execution," *IEEE/ACM Transactions on Networking*, vol. 14, no. 1, pp. 55–67, 2006.

[16] L. Wang and Y. Xiao, "A survey of energy-efficient scheduling mechanisms in sensor networks," *Mobile Networks and Applications*, vol. 11, no. 5, pp. 723–740, 2006.

[17] J. Thenepalle and P. Singamsetty, "The degree constrained *k*-cardinality minimum spanning tree problem: a lexisearch algorithm," *Decision Science Letters*, vol. 7, pp. 301–310, 2018.

[18] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, New York, NY, USA, 1979.

[19] D. Lozovanu and A. Zelikovsky, "Minimal and bounded tree problems," in *Tezele Congresului XVIII Al Academiei Romano-Americane*Kishniev, Chişinău, Moldova, 1996.

[20] R. Ravi, R. Sundaram, M. V. Marathe, D. J. Rosenkrantz, and S. S. Ravi, "Spanning trees-short or small," *SIAM Journal on Discrete Mathematics*, vol. 9, no. 2, pp. 178–200, 1996.

[21] P. Adasme, I. Soto, and F. Seguel, "Finding degree constrained k-cardinality minimum spanning trees for wireless sensor

networks," in *Mobile Web and Intelligent Information Systems*, M. Younas, I. Awan, G. Ghinea, and M. Catalan Cid, Eds., vol. 10995, pp. 51–62, Lecture Notes in Computer Science, Springer, Cham, Switzerland, 2018.

[22] L. Caccetta and S. P. Hill, "A branch and cut method for the degree-constrained minimum spanning tree problem," *Networks*, vol. 37, no. 2, pp. 74–83, 2001.

[23] L. Euler, "Solutio problematis ad geometriam situs pertinentis," *Commentarii Academiae Scientiarum Imperialis Petropolitanae*, vol. 8, pp. 128–140, 1976.

[24] P. Adasme, R. Andrade, M. Letournel, and A. Lisser, "Stochastic maximum weight forest problem," *Networks*, vol. 65, no. 4, pp. 289–305, 2015.

[25] M. Dorigo and L. M. Gambardella, "A study of some properties of ant-Q," in *Prcoeedings of the 1996 Parallel Problem Solving from Nature—PPSN IV*, Berlin, Germany, 1996.

[26] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system: optimization by a colony of cooperating agents," *IEEE Transactions on Systems, Man and Cybernetics, Part B (Cybernetics)*, vol. 26, no. 1, pp. 29–41, 1996.

[27] L. Gambardella and M. Dorigo, "Ant-Q: a reinforcement learning approach to the traveling salesman problem," in *Proceedings of the 12th International Conference on Machine Learning*, Tahoe City, CA, USA, July 1995.

[28] P. Hansen and N. Mladenović, "Variable neighborhood search: principles and applications," *European Journal of Operational Research*, vol. 130, no. 3, pp. 449–467, 2001.

[29] N. Mladenovic and P. Hansen, "Variable neighborhood search," *Computers & OR*, vol. 24, no. 11, pp. 1097–1100, 1997.

[30] L. Caccetta and Wamiliana, "Heuristics approach for the degree constrained minimum spanning tree," in *Proceedings of the 2001 International Modeling and Simulations*, pp. 2161–2166, Canberra, Australia, 2001.

[31] C. Watkins, *Learning with delayed rewards*, Ph.D. dissertation, Psychology Department, University of Cambridge, Cambridge, UK, 1989.

[32] Y. Bengio, A. Lodi, and A. Prouvost, "Machine learning for combinatorial optimization: a methodological tour d'horizon," 2018, https://arxiv.org/abs/1811.06128.

[33] J. P. Queiroz dos Santos, J. D. de Melo, A. D. Duarte Neto, and D. Aloise, "Reactive search strategies using reinforcement learning, local search algorithms and variable neighborhood search," *Expert Systems with Applications*, vol. 41, no. 10, pp. 4939–4949, 2014.

[34] https://github.com/pdrozdowski/TSPLib.Net/tree/master/TSPLIB95/tspTSBLIB,.

[35] R. Andrade, A. Lucena, and N. Maculan, "Using Lagrangian dual information to generate degree constrained spanning trees," *Discrete Applied Mathematics*, vol. 154, no. 5, pp. 703–717, 2006.

[36] M. Chlebík and J. Chlebíková, "The Steiner tree problem on graphs: inapproximability results," *Theoretical Computer Science*, vol. 406, no. 3, pp. 207–214, 2008.

[37] G. Naveen, "Saving an epsilon: a 2-approximation for the k-MST problem in graphs," in *Proceedings of the 37th Annual ACM Symposium on Theory of Computing*, Baltimore, MD, USA, 2005.

[38] H. Katagiri and Q. Guo, "A hybrid-heuristics algorithm for *k*-minimum spanning tree problems," in *IAENG Transactions on Engineering Technologies*, pp. 167–180, Springer, Dordrecht, Netherlands, 2013.

[39] H. Katagiri, T. Hayashida, I. Nishizaki, and Q. Guo, "A hybrid algorithm based on tabu search and ant colony optimization for *k*-minimum spanning tree problems," *Expert Systems with Applications*, vol. 39, no. 5, pp. 5681–5686, 2012.

[40] M. Doan, "An effective ant-based algorithm for the degree-constrained minimum spanning tree problem," in *Prcoeedings of the Evolutionary Computation IEEE CEC 2007*, Singapore, September 2007.

[41] L. Hanr and Y. Wang, "A novel genetic algorithm for degree-constrained minimum spanning tree problem," *International Journal of Computer Science and Network Security*, vol. 6, no. 7A, pp. 50–57, 2006.

[42] M. Krishnamoorthy, A. T. Ernst, and Y. M. Sharaiha, "Comparison of algorithms for the degree constrained minimum spanning tree," *Journal of Heuristics*, vol. 7, no. 6, pp. 587–611, 2001.

[43] S. C. Narula and C. A. Ho, "Degree-constrained minimum spanning tree," *Computers & Operations Research*, vol. 7, no. 4, pp. 239–249, 1980.

[44] A. Volgenant, "A Lagrangean approach to the degree-constrained minimum spanning tree problem," *European Journal of Operational Research*, vol. 39, no. 3, pp. 325–331, 1989.

[45] M. C. de Souza and P. Martins, "Skewed VNS enclosing second order algorithm for the degree constrained minimum spanning tree problem," *European Journal of Operational Research*, vol. 191, no. 3, pp. 677–690, 2008.

[46] R. J. Chagas, C. A. Valle, and A. S. da Cunha, "Exact solution approaches for the Multi-period degree constrained minimum spanning tree problem," *European Journal of Operational Research*, vol. 271, no. 1, pp. 57–71, 2018.

[47] K. Singh and S. Sundar, "A hybrid steady-state genetic algorithm for the min-degree constrained minimum spanning tree problem," *European Journal of Operational Research*, vol. 276, no. 1, pp. 88–105, 2019.

[48] Q. Wang, H. E. Psillakis, and C. Sun, "Cooperative control of multiple agents with unknown high-frequency gain signs under unbalanced and switching topologies," *IEEE Transactions on Automatic Control*, vol. 64, no. 6, pp. 2495–2501, 2019.

[49] Q. Wang and C. Sun, "Adaptive consensus of multiagent systems with unknown high-frequency gain signs under directed graphs," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 50, no. 6, pp. 2181–2186, 2020.

[50] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical Society*, vol. 7, no. 1, p. 48, 1956.

[51] R. C. Prim, "Shortest connection networks and some generalizations," *Bell System Technical Journal*, vol. 36, no. 6, pp. 1389–1401, 1957.

[52] M. Desrochers and G. Laporte, "Improvements and extensions to the Miller-Tucker-Zemlin subtour elimination constraints," *Operations Research Letters*, vol. 10, no. 1, pp. 27–36, 1991.

[53] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press and McGraw-Hill, Cumberland, RI, USA, 2009.

[54] IBM ILOG, "CPLEX high-performance mathematical programming engine," 2016, http://www.ibm.com/software/integration/optimization/cplex/.