Less is more: general variable neighborhood search for the capacitated modular hub location problem

Marija Mikić, Raca Todosijević, Dragan Urošević

Computers and Operations Research 110 (2019) 101–115

Less is more: General variable neighborhood search for the capacitated modular hub location problem

Marija Miki ć [a], Raca Todosijevi ć [b, c], Dragan Uroševi ć [b, *]

[a] Faculty of Mathematics, University of Belgrade, Belgrade, Serbia
[b] Mathematical Institute SANU, Belgrade, Serbia
[c] Univ. Polytechnique Hauts-de-France, CNRS, UMR 8201 - LAMIH - Laboratoire d'Automatique, de Mécanique et d'Informatique industrielles et Humaines, F-59313 Valenciennes, France

article info

Keywords:
Hub location problem
General variable neighborhood search
Basic sequential variable neighborhood descent
Heuristic

abstract

In this paper, we study the capacitated modular hub location problem. The problem belongs to the class of the single assignment hub location problems, where a terminal can be assigned to only one hub. In addition, the problem imposes capacity constraints, on both hubs and edges that connect them. The observed problem is directly related to the real problem. Namely, in air traffic, the number of flights between two cities directly determines the conditions of the capacity. In order to tackle the problem we propose a general variable neighborhood search (GVNS) based heuristic. We have performed exhaustive testing that led to the conclusion that the GVNS method gave superior results in comparison to the previous methods. This is especially reflected in the number of best solutions that were obtained in a much shorter time. Additionally, we applied statistical tests which showed that GVNS is undoubtedly superior with respect to the previously observed methods.

1. Introduction

Hub location problems belong to a very important sub-field of location science. As such they represent a fertile area for interdisciplinary researchers from operations research, transportation, geography, network design, telecommunications, regional science, economics, etc. The term hub refers to the special facility which serves as switching, transshipment and sorting point in a distribution system (see Alumur and Kara (2008)). Existence of such facilities enables the flow to be redirected through them instead of directly linking each origin and each destination. Main features of hub locations problems as indicated in Campbell (1994) and Campbell and O'Kelly (2012) are:

- Demand is associated with flows between origin-destination (OD) pairs;
- Flows are allowed to go through hub facilities;
- Hubs are facilities to be located;
- There is a benefit of routing flows via hubs (or a requirement to route flows via hubs);
- There is an objective that depends on the locations of hub facilities and the routing of the flows.

In addition, most hub location problems impose two additional assumptions:

- Paths between OD pairs visit at most two hubs;
- Direct OD flows are not allowed.

Papers that relax the last two assumptions are, for example, Brimberg et al. (2019); Contreras et al. (2010, 2017); Mahmutogullari and Kara (2015); Taherkhani and Alumur (2018). In Mahmutogullari and Kara (2015) and Taherkhani and Alumur (2018) the direct flows are allowed, while in papers Brimberg et al. (2019); Contreras et al. (2010) and Contreras et al. (2017) paths between OD pairs may contain more than two hubs. Contreras et al. (2010) and Contreras et al. (2017) consider structured non-fully connected backbone networks (e.g., tree-of-hubs and the cycle-of-hubs), while Brimberg et al. (2019) relax the assumption that triangle inequality holds for costs.

The main applications of hub location problems have been in transportation, telecommunications and postal services. For example, in air transport it is unreasonable to establish direct flights between each pair of cities due to extremely high prices for doing so. For this reason, a different approach is used. Namely, passengers in air traffic are redirected through intermediate airports. Such airports are called hubs. On the other hand, hub location problems in telecommunications include the location of hubs (i.e.,

DOI : 10.1016/j.cor.2019.05.020

hardware such as switches, routers, and concentrators) through which the flow between origin and destination is directed. Apart from this example, which has its everyday application, hub location problems are present in the maritime industry, freight transport companies, public transit and message delivery networks (for more details see Farahani et al. (2013)). For surveys on hub location problems we refer the reader to Campbell and O'Kelly (2012) and Alumur and Kara (2008).

In this paper, we study the capacitated single assignment hub location problem with modular link capacities (CSHLPMLC). The CSHLPMLC is defined on a backbone/tributary network. In this kind of network, each node is defined either as a terminal or as a hub. *Terminal nodes* are nodes that represent origins and destinations. As the flow must be routed from origins to destinations, and direct origin–destination connections are economically unprofitable, as has been previously noticed, the flow is routed through nodes called *hubs*. The network that connects the terminal nodes to the hubs (to which they are assigned) is called a *tributary network*. The network that connects the hubs is called the *backbone network*. The CSHLPMLC considers a fully connected backbone network and each terminal is assigned to exactly one hub. Also, it is not allowed to directly connect the terminals. Therefore, every flow between each two terminals must be done through at least one hub. In addition, the CSHLPMLC assumes that at most two hubs are used in each path to route the flow. Therefore, there are two options. The first option is that the flow from one terminal to another is routed through the path Terminal1–Hub–Terminal2 (if both terminals are assigned to the same hub). Alternatively, if the terminals do not share a common hub, the flow is routed through the path Terminal1–Hub1–Hub2–Terminal2.

We consider fixed costs of installing hubs and fixed costs of installing the needed capacity on each edge. The capacity needed to route the flow on an edge is provided by the installation of an integer number of *links* of fixed capacity. The link capacity can be different for the backbone and for the tributary edges. Given a flow matrix, which represents the flow between terminal nodes, the CSHLPMLC problem consists of determining the set of nodes to serve as hubs, assigning each terminal node to exactly one hub and installing capacities on edges, so that all the flow is routed respecting the capacity constraints. The aim is to minimize the total cost of establishing a network, which is the sum of hub installation costs and link costs.

In the work by Yaman and Carello (2005) the CSHLPMLC problem was introduced for the first time. In this paper a branch-and-cut algorithm and a tabu search metaheuristic were presented. To solve larger instances, Corberán et al. (2016) used a heuristic method based on the strategic oscillation (SO). Their method is based on constructive and destructive algorithms which work together with associated local search procedures, to balance diversification and intensification. The initial solution is constructed in a greedy manner. The destructive procedure works on a complete solution, from which it removes a fixed number of hubs and terminals. On the other hand, the constructive procedure repairs a partial solution created by the destructive procedure in order to have a complete solution which is used as the initial solution for the local search procedures. Local search procedures are based on two neighborhoods: one which exchanges terminal-hub assignments of two terminals assigned to two different hubs (the first terminal is reassigned to the hub associated with the second terminal and vice versa) and another that changes hub-terminal assignment of single terminal by assigning it to another hub. Hoff et al. (2017) proposed a new heuristic, named adaptive memory procedure (AMP), to solve the CSHLPMLC. Their method is an iterative procedure that employs the constructive procedure and the local search procedure. The constructive procedure uses adaptive memory structures which means that instead of randomization, it keeps track of past hub appearances to discourage their selection in future constructions. The constructive procedure provides an initial solution for the local search procedure. The local search procedure, in addition to the neighborhood structures proposed in Corberán et al. (2016), exploits neighborhood structures that open or close a hub. Also, it uses the neighborhood structure in which a terminal becomes a hub, while the hub it is assigned to is being closed. In this case, all terminals that were assigned to the closed hub, as well as the closed hub itself, are being assigned to the newly opened hub. At the end of the iterative procedure, the path relinking procedure is launched on the elite solutions encountered during the run of the iterative procedure. The problems related to the CSHLPMLC have been studied recently by Rastani et al. (2016) and Tanash et al. (2017). Rastani et al. (2016) considered a hub problem with multi-level capacities where hubs and hub links take capacity from a given set of capacities. In the paper by Tanash et al. (2017) a hub location problem is considered where flow-dependent transportation costs are modeled using modular link costs. The authors presented two mixed integer programming formulations and propose a Lagrangean relaxation for one of them. Based on this relaxation, and on a heuristic algorithm, they developed a branch and bound producing good computational results on benchmark instances with up to 75 nodes. For a more detailed survey on problems related to the CSHLPMLC, we refer the reader to Hoff et al. (2017), Mohri et al. (2018) and Momayezi et al. (2018).

In this paper, we propose a heuristic based on variable neighborhood search metaheuristic. It examines just two neighborhood structures. One neighborhood structure is already used in Hoff et al. (2017), while another neighborhood structure is an extension of a neighborhood structure from Hoff et al. (2017) and Corberán et al. (2016). This extension allows that a hub is closed during the intensification phase or opened during the shaking (diversification) phase. In Hoff et al. (2017), the authors considered neighborhood structures that open and close a hub as well. In their implementation, when a hub is opened, a new assignment of terminals to hubs is made as well. On the other hand, when a hub is closed, all terminals assigned to it are reassigned to the remaining hubs. The difference in our implementation is that we do not consider these two neighborhood structures at all. We rather allow that when we change a terminal-hub assignment we may open/close a hub. More precisely, if only a single terminal is assigned to an individual hub (the hub itself), it is allowed to be reassigned to another hub, therefore enabling it to be closed. Similarly, we allow a non-hub node to become a hub assigned to itself. In this way, we accomplish the opening/closing of the hub, simply, within the neighborhood that changes terminal-hub assignments. Although, this neighborhood structure has been considered in Hoff et al. (2017) and Corberán et al. (2016), in these papers it was not allowed to change a hub set while exploring this neighborhood, i.e., while changing terminal-hub assignment. For this reason, in Hoff et al. (2017) the authors introduce two more complex neighborhood structures in order to possibly improve the current solution by changing the cardinality of a hub set. Beside this, we introduce auxiliary data structures that speed up the exploration of the used neighborhoods. The proposed heuristic is tested on benchmark instances from the literature. The obtained results demonstrate its superiority over existing state-of-the-art heuristics. In sum, the contributions of the paper are three-fold: the extension of existing neighborhood structures used for the CSHLPMLC; an efficient way of exploring neighborhood structures using auxiliary data structures; and establishment of new state-of-the-art results.

It should be noted that our method uses a smaller number of neighborhood structures than the current state-of-the-art adaptive memory based procedure (AMP) proposed by Hoff et al. (2017). In

addition, it uses a very simple procedure to construct an initial solution. Consequently, the proposed approach may be classified as a less is more approach (LIMA). Less is more approach is a recent approach in solving optimization problems (see Mladenović et al. (2016), Brimberg et al. (2017) and Costa et al. (2017)). LIMAs main idea is to find the minimum number of search ingredients in solving some particular optimization problem, that makes some heuristic more efficient than the ones currently considered to be the best in literature. In other words, the goal is to make heuristic as simple as possible, but at the same time, more effective and efficient than the current state-of-the-art heuristic.

The rest of this paper is organized as follows. In Section 2, we give the mathematical formulation of problem. In Section 3, we introduce our approach. We give details of proposed Basic Variable Neighborhood Descent (BVND), general variable neighborhood search (GVNS) and shaking procedure. In Section 4, we present the computational results and we compare them with previous results. In Section 5, we give conclusion remarks and outline some possible research directions.

2. Problem formulation

In this paper, we consider the single assignment hub location problem. The problem considered here consists in selecting a subset of nodes to be hubs, and assigning the rest of the nodes to them, in such a way that the transportation cost is minimized while satisfying the capacity constraints.

Let $G = (V, E)$ be a complete graph, where $V = \{1, 2, \ldots, n\}$ represents a set of nodes (terminals), while E denotes a set of edges. Each edge $\{i, j\} \in E$, has two arcs associated, i.e., arcs (i, j) and (j, i). Hence, the set of all arcs is defined as $A = \{(i, j) : i, j \in V\}$.

In the CSHLPMLC, the number of hubs is not imposed. The cost of opening a hub at the node k is fixed and it is given as c_{kk}, while the cost of assigning node i to hub k is given as c_{ik}.

For any pair of nodes $i, j \in V$, t_{ij} denotes the flow to be transported from node i to node j.

The flow is routed through hubs in a way that passes through at least one and at most two hubs. For each hub opened at node k there is a limit, Q_k^h, on the amount of flow it can accommodate. This limitation directly affects the number of terminals that can be allocated to the observed hub.

We will distinguish two types of edges which connect nodes: access edges and backbone edges. Terminals are connected to hubs using the access edges, while hubs are connected using backbone edges. Each backbone edge has a maximum flow capacity (in each direction), and it is given as Q^b.

On an edge $\{k, l\}$ we may install several backbone edges. The number of installed edges was denoted by w_{kl} and the cost of installing one backbone edge on edge $\{k, l\}$ by r_{kl}. Then, the maximum flow that we can transport on edge $\{k, l\}$, in each direction, is $Q^b w_{kl}$ and the associated cost of establishing backbone edges is $w_{kl} r_{kl}$. Since each backbone edge connects two hub nodes, the maximum number of links that may be installed on certain edge connecting hubs h_1 and h_2 is given as $\lceil \frac{max\{Q^h_{h_1}, Q^h_{h_2}\}}{Q^b} \rceil$.

In order to formulate CSHLPMLC as a mixed integer nonlinear program the following variables are used (see Yaman and Carello (2005)):

- the variable x_{ii} takes the value of 1 if node i receives a hub, and 0 otherwise. The variable x_{ik} is equal to 1 if node i is allocated to hub k, and 0 otherwise;
- the non-negative variable z_{kl}, which denotes flow on an arc $(k, l) \in A$;
- the variable w_{kl}, which denotes the number of links on the edge $\{k, l\} \in E$.

The formulation is, then:

$$\min \sum_{i \in V} \sum_{k \in V} c_{ik} x_{ik} + \sum_{\{k,l\} \in E} r_{kl} w_{kl} \tag{1}$$

Subject to:

$$\sum_{k \in V} x_{ik} = 1, \qquad \forall i \in V \tag{2}$$

$$x_{ik} \leq x_{kk}, \quad \forall i \in V, \qquad \forall k \in V \backslash \{i\} \tag{3}$$

$$\sum_{i \in V} \sum_{j \in V} (t_{ij} + t_{ji}) x_{ik} - \sum_{i \in V} \sum_{j \in V} t_{ij} x_{ik} x_{jk} \leq Q_k^h x_{kk}, \qquad \forall k \in V \tag{4}$$

$$z_{kl} \geq \sum_{i \in V} \sum_{j \in V} t_{ij} x_{ik} x_{jl}, \qquad \forall (k, l) \in A \tag{5}$$

$$Q^b w_{kl} \geq z_{kl}, \; Q^b w_{kl} \geq z_{lk}, \qquad \forall \{k, l\} \in E \tag{6}$$

$$x_{ik} \in \{0, 1\}, \qquad \forall i \in V, \forall k \in V \tag{7}$$

$$w_{kl} \geq 0 \text{ and integer}, \qquad \forall \{k, l\} \in E. \tag{8}$$

The objective function in (1) calculates the cost of establishing the network. Constraints (2) require that exactly one hub is assigned to each node, while constraints (3) ensure that node k must be a hub if node i is assigned to a node k. Constraints (4) do not allow that the total flow which goes through hub k to be greater than its capacity Q_k^h. In this case, subtracting the second member on the left-hand side of the inequality ensures that there is no double-counting between the pairs of nodes assigned to the same hub. Constraints (5) regulate flow through the observed arc (k, l), while constraints (6) determine the number of links needed on each backbone edge. Finally, constraints (7) and (8) define the integrality properties of the variables and their domains.

3. Solution method

In this section, we give a thorough description of the proposed approach. First, we describe how the solution of the problem is represented in our approach and which data structures are used to accomplish this. After that, we present neighborhood structures following the introduced solution representation. Finally, we describe how these neighborhood structures are explored within a variable neighborhood descent scheme as well as within a variable neighborhood search scheme.

3.1. Solution representation

A solution of capacitated modular hub location problem is determined by a set of hubs, the node-to-hubs assignments, and the travel paths for each pair of nodes. More precisely, we represent a solution as $S = (H, A)$, where H is a set containing hubs, while A is a series of assignments, i.e., $A[i]$ represents hub assigned to node i. For example, if node 12 is assigned to hub 1, then $A[12] = 1$.

Each hub together with its assigned terminals form a set (*cluster*) in the network. The size of the cluster CL_k that corresponds to the hub k is the number of terminals assigned to hub k (the hub is assigned to itself and thus it is counted as well). We observe different types of flow in the network. *Node to Cluster Flow* (NCF) represents sent and received flow from a node to a cluster (for the cluster with a node i as a hub, flow t_{ii} is not taken into account), *Cluster to Node Flow* (CNF) represents sent and received flow from a cluster to a node (for the cluster with a node i as a hub, flow t_{ii} is not taken into account again), and *Cluster to Cluster Flow* (CCF) represents the flow sent between two clusters. These additional data structures will be used to speed up the exploration of the solution space as it will be explained subsequently. In Corberán et al. (2016), the authors used just data structure CCF.

3.2. Initial solution

The initial solution of the CSHLPMLC problem is constructed as follows. First, each node is designed to be a hub. In other words, the set H contains all nodes. After the initial set of hubs H is determined, each node (hub) is assigned to itself. This means that each cluster contains only one element, i.e., a hub. Finally, we calculate matrices NCF, CNF, CCF, and the objective function value of the initial solution.

3.3. Neighborhood structures

In this article, we consider two neighborhood structures in order to improve a solution. Neighborhood structures are defined by operators that change a set of chosen hubs or node to hub assignments. More precisely, for a given solution $S = (H, A)$, we define the following neighborhood structures:

- *Replacement of the Terminal and Hub within the Cluster (RTHC(S))* contains all solutions that may be obtained by changing of a hub within a cluster, i.e., the hub in a cluster changes its status to become a terminal and one of the terminals in this cluster is the new hub of the cluster, if its capacity allows this. The rest of the terminals in the cluster remain the same but are now assigned to the new hub. For example, if node 2 is a hub in the cluster, and nodes 3,13 and 15 are terminals in the same cluster, then $A[3]$, $A[13]$ and $A[15]$ are equal 2. After replacement, if node 13 becomes a hub and node 2 becomes a terminal, then $A[2] = 13$. Also, for all unchanged terminals $j \in \{3, 15\}$ in that cluster, we have $A[j] = 13$.
  As RTHC limits the exploration to changes within a cluster, there is no need to recalculate the number of links on the backbone edges to compute the value of the new solution. Hence, in order to calculate the value of some new solution $S' = (H', A')$ generated by such a replacement, it suffices to update the cost of node to hub assignments. This may be accomplished by recalculating the assignment cost of each node in the cluster whose corresponding hub has been changed.
  This neighborhood structure has been used in Hoff et al. (2017) as well as in Ilic et al. (2010).
- *Moving the Terminal from one Cluster to another Cluster (MTCC(S))*, as the name says, contains all solutions that may be obtained by moving the terminal from one cluster to another, i.e., the terminal from one cluster becomes a terminal in another cluster if its capacity allows this. Note that the size of the cluster from which the terminal is ejected decreases by one, while the size of the cluster to which the terminal is moved increases by one. For example, if node 2 was a terminal assigned to hub 1 in some cluster, then $A[2]$ was equal 1. After replacement, if node 2 becomes a terminal assigned to hub 11 in another cluster, then $A[2] = 11$. Note that if the cluster contains only one terminal (i.e., hub itself), then we can close this hub as well as the corresponding cluster by moving this terminal (hub) to another cluster.
  In order to update the value of some new solution $S' = (H', A')$ generated by such a move, we first have to update the cost of node to hub assignment for a terminal that has been moved to another cluster. In addition, the number of established backbone edges has to be recalculated between the cluster whose configuration has been changed and each other cluster. More precisely, let us assume that terminal n is moved from cluster i to cluster j. Then the number of required backbone edges between a cluster j and cluster k is given as:

  $$\lceil \max\{CCF[j][k] + NCF[n][k], CCF[k][j] + CNF[k][n]\}/Q_b \rceil.$$

  Similarly, the number of required backbone edges between a cluster i and cluster k is given as:

  $$\lceil \max\{CCF[i][k] - NCF[n][k], CCF[k][i] - CNF[k][n]\}/Q_b \rceil.$$

  Finally, the number of required backbone edges between clusters i and j is given as:

  $$\lceil \max\{CCF[i][j] - NCF[n][j] + CNF[i][n], CCF[j][i] - CNF[j][n] + NCF[n][i]\}/Q_b \rceil.$$

  The next two properties give information about time complexity needed to evaluate a neighboring solution as well as time complexity to update data structures after accepting a solution from the neighborhood $MTCC(S)$ to be new incumbent solution.

Property 3.1. *Evaluating each solution in the neighborhood $MTCC(S)$ requires $O(|H|)$ operations.*

Property 3.2. *Updating data structures NCF and CNF after accepting a solution from the neighborhood $MTCC(S)$ to be the new incumbent solution, requires $O(|V|)$ operations, while updating data structure CCF requires $O(|H|)$ operations.*

Note that the straightforward computation in the change of required backbone edges requires $O(\sum_{k\in H, k\neq i,j} |CL_k|(|CL_i| - 1) + \sum_{k\in H, k\neq i,j} |CL_k|(|CL_j| + 1) + (|CL_i| - 1)(|CL_j| + 1))$ operations. It may be verified that $O(\sum_{k\in H, k\neq i,j} |CL_k|(|CL_i| - 1) + \sum_{k\in H, k\neq i,j} |CL_k|(|CL_j| + 1) + (|CL_i| - 1)(|CL_j| + 1)) \geq O(\sum_{k\in H} CL_k) = O(|V|)$. The equality in the last inequality holds when, for example, $|CL_i| = |CL_j| = 1$. Since it usually holds that $|H| \ll |V|$ significant savings may occur by using the auxiliary data structure. For example, if we assume that we have only three hubs ($|H| = 3$) and each cluster contains $|V|/3$ nodes, than the straightforward computation in the change of required backbone edges has complexity $O(|V|^2)$, while using auxiliary data structures just $O(|H|)$ operations is needed to do so. In addition, updating auxiliary data structures, which does not occur so often, requires $O(|V|)$ which is again less than $O(V^2)$.

The following two propositions provide information on the cardinalities of the neighborhood structures $RTHC(S)$ and $MTCC(S)$.

Property 3.3. *The cardinality of neighborhood $RTHC(S)$ is $O(\sum_{i\in H} CL_i) = O(|V|)$.*

Property 3.4. *The cardinality of neighborhood $MTCC(S)$ is $O(|V||H|)$.*

3.4. Basic sequential variable neighborhood descent (BVND)

The variable neighborhood descent (VND) passes iteratively through several neighborhood structures in order to improve the current solution if it is possible. In the paper authored by Hansen et al. (2017), it is underlined that the main principle of VND stems from the fact that it is much more likely that the global minimum will be achieved if the solution is a local minimum with respect to several neighborhood structures, instead of just one.

In this paper, we used the basic sequential VND (BVND) procedure in the following way. Namely, starting from the given solution S, BVND procedure iteratively goes through the neighborhood structures $RTHC(S)$ and $MTCC(S)$, one after another. As soon as an improved solution is reached in some neighborhood structure, the BVND procedure continues search in the first neighborhood structure with a new, currently best-performing solution. The whole process ends if the current solution can not be improved in any of the considered neighborhood structures.

The steps of the basic sequential VND using the best improvement search strategy are given in Algorithm 1.

Algorithm 1: Basic Sequential Variable Neighborhood Descent.

Function BVND(S);
$k \leftarrow 1$;
while $k \leq 2$ **do**
  if $k = 1$ **then**
    $k \leftarrow 2$;
    $S' \leftarrow$ LocalSearch(S, N_{MTCC});
  else
    $k \leftarrow 3$;
    $S' \leftarrow$ LocalSearch(S, N_{RTHC});
  end
  if S' *better than* S **then**
    $S \leftarrow S'$;
    $k \leftarrow 1$;
  end
end
return S'

3.5. Shaking procedure

In order to diversify the search and avoid optima traps, we use the shaking procedure which requires solution S and parameter k at the input. The parameter k represents the number of iterations of the shaking procedure. Within each of the iterations, the current solution is replaced by a randomly chosen solution from the $MTCC(S)$ neighborhood structure.

More precisely, by utilising the shaking procedure, we select a cluster in an arbitrary way and select a terminal inside that cluster in an arbitrary way, too. Then, we arbitrarily choose the second cluster where we move the selected terminal from the first cluster (respecting the capacity of the second cluster). Note that we allow that the terminal is moved to either an existing cluster or it is designated to be a hub, i.e., a new cluster is formed containing just this terminal (which is at the same time a hub).

The output of the shaking procedure is the solution obtained in the k-th iteration. In Algorithm 2, key steps of the shaking procedure are shown.

Algorithm 2: Shaking procedure.

Function shaking(S, k);
1 **for** $i = 1$ *to* k **do**
2 Select S'' in $MTCC(S)$ at random;
3 $S \leftarrow S''$;
  end
4 **return** S;

3.6. General variable neighborhood search (GVNS)

In this paper, we propose a heuristic based on *variable neighborhood search* (VNS) that uses *basic sequential variable neighborhood descent* (BVND). Variable neighborhood search (VNS) is a metaheuristic suggested by Mladenović and Hansen (1997). It contains two main phases: improvement phase, used to possibly get an improved solution, and the so-called shaking phase, used to hopefully avoid local optima traps generated in the improvement phase. Within the variable neighborhood search scheme, the improvement and shaking phases are executed alternately until the pre-set stopping criteria are achieved.

The most common and widely used VNS variant is General VNS (GVNS). Unlike basic VNS (BVNS) which uses simple local search in the improvement, GVNS uses a more advanced improvement procedure that examines several neighborhood structures. The most common improvement methods used in GVNS are variable neighborhood descent variants (VND), such as sequential VND, nested VND, cyclic VND, etc.

For more information about VND and VNS we refer to Hansen et al. (2017).

Algorithm 3: General VNS.

Function GVNS(S, $k_{\max}$, $T_{\max}$);
1 $S' \leftarrow$ initial solution();
2 $S \leftarrow$ BVND (S') ;
3 **repeat**
4 $k \leftarrow 1$;
5 **while** $k \leq k_{\max}$ **do**
6 $S' \leftarrow$ shaking (S, k) ;
7 $S'' \leftarrow$ BVND (S') ;
8 $k \leftarrow k + 1$;
9 **if** S'' *is better then* S **then**
10 $S \leftarrow S''$; $k \leftarrow 1$;
  end
  end
11 $T \leftarrow$ CpuTime();
  until $T > T_{\max}$;
12 **Return** S;

In this paper, we develop a GVNS heuristic (Algorithm 3) which uses the BVND and the previously described shaking procedure. Our GVNS heuristic has two parameters: the first one denoted by $T_{\max}$ represents maximum CPU time allowed to be consumed by GVNS, while the second one, named $k_{\max}$, represents the maximum number of iterations that can be executed within the shaking procedure. In this case, we have taken that $k_{\max} = \frac{n}{8}$ and $T_{\max} = n$ seconds (see Section 4.1).

Note that the feasibility of solutions is maintained throughout the entire solution process. Our GVNS starts from a feasible solution and accepts only feasible solutions as new incumbent solutions within both the BVND and the shaking procedure. Moreover, the neighborhood structures considered in the BVND and the shaking procedure contain only the feasible solutions.

4. Computational results

In this section, we present the results of the computational experiments that we have performed with the aim of examining the effectiveness and efficiency of the proposed approach.

The proposed GVNS algorithm is coded in C and executed on an Intel(R) Core(TM) i5-3470 with CPU 3.20GHz and 16GB RAM.

For testing purposes, we have used 170 instances generated by Corberán et al. (2016) which have been considered as benchmark instances in previous publications on the CSHLPMLC. These instances were derived from three standard hub problem sets (i.e., CAB (O'Kelly (1987)), AP (Ernst and Krishnamoorth (1996)) and USA423 (Peiró et al. (2014))) by imposing associated modular capacities as well as costs for installing hubs and edges. Accordingly, the examined instances are classified in the following three groups:

- The **CAB** data set is derived from the Civil Aeronautics Board survey of 1970 of passenger data in the United States of America and has 23 instances. The number of nodes in these instances ranges from 10 to 25.
- The **AP** data set is based on real data from the Australian Postal service and has 70 instances. The number of nodes in these instances ranges from 10 to 200.

Table 1
Summary results for GVNS: different values of k_{max} parameter.

	k_{max}									
	$\lceil n/2 \rceil$		$\lceil n/4 \rceil$		$\lceil n/8 \rceil$		$\lceil n/16 \rceil$		$\lceil n/32 \rceil$	
Size	Value	CPU	Value	CPU	Value	CPU	Value	CPU	Value	CPU
Small	288954.39	11.69	**288821.20**	11.75	289206.30	8.67	291686.98	9.85	306274.08	0.21
Medium	556104.09	55.04	553399.55	50.58	**550214.43**	53.48	554480.38	52.57	579025.39	34.39
Large	913636.96	106.86	906940.43	101.45	**902068.65**	97.36	903647.91	97.05	936216.14	96.64
Extra Large	739253.38	137.68	736167.85	139.97	**734874.66**	128.77	735598.41	127.74	744492.26	129.91
Huge	620177.04	168.99	613322.69	164.57	610267.93	168.90	**606278.31**	163.01	610050.89	154.25
Average	623625.17	96.05	619730.34	93.67	**617326.39**	91.43	618338.40	90.05	635211.75	83.08

Table 2
Summary results for GVNS: different values of T_{max} parameter.

			T_{max}									
			n		$\lceil n/2 \rceil$		$\lceil n/4 \rceil$		$\lceil n/8 \rceil$		$\lceil n/16 \rceil$	
Size	SO	AMP	Value	CPU	Value	CPU	Value	CPU	Value	CPU	Value	CPU
Small	323124.50	307267.25	289206.30	8.67	289625.58	5.99	290106.55	3.11	290872.56	1.58	291427.83	0.80
Medium	631817.75	588495.75	550214.43	53.48	557188.48	25.99	560163.76	14.53	568210.50	6.88	578611.88	3.46
Large	1115154.75	991018.25	902068.65	97.36	920480.03	54.52	939337.38	27.43	1002182.91	14.44	1081973.59	7.02
Extra Large	936864.25	783344.25	734874.66	128.77	739376.88	70.72	748266.05	37.53	809699.78	19.89	1081820.83	10.22
Huge	789900.50	700929.50	610267.93	168.90	620214.00	86.34	626632.81	44.39	636339.80	22.29	653375.48	11.41
Average	759372.35	674211.00	617326.39	91.43	625376.99	48.71	632901.31	25.39	661461.11	13.01	737441.92	6.58

- The **USA423** data set is based on a data file concerning 423 cities in the United States of America and has 77 instances. The number of nodes in these instances ranges from 20 to 250.

Each of the used instances includes the matrices (t_{ij}), (c_{ij}), (r_{kl}) as well as the capacities for each backbone edge and each hubs. Entire set of instances can be found at the following address: www.optsicom.es. The first number in the instance name indicates the number of nodes in the instance, while the letters indicate whether the instance is derived from the CAB, AP or USA423 data set.

The instances have been divided into five groups according to their size:

- small ($10 \leq n \leq 50$);
- medium ($55 \leq n \leq 100$);
- large ($110 \leq n \leq 150$);
- extra-large ($155 \leq n \leq 200$);
- huge ($205 \leq n \leq 250$).

In the rest of the section, we first perform preliminary tests to determine the best parameter setting for our GVNS. After that, extensive testing is conducted on benchmark instances and the results are compared with the state-of-the-art ones.

4.1. Parameter calibration

In order to tune our GVNS heuristic we perform tests on the subset of 20 instances. Since the instances are divided in five groups according to their size, from each group we choose one smallest and one largest AP instance, as well as one smallest and one largest USA instance. CAB instances were neglected because these instances belong just to the set of small instances and the largest instance has only 20 nodes. The chosen instances are:

- small: 10_700_50_60_8_1_60_AP; 20_700_50_60_8_1_60_USA; 50_700_80_50_8_1_69_AP; 50_700_80_50_8_1_69_USA;
- medium: 55_500_60_69_60_1_50_AP; 55_500_60_69_60_1_50_USA; 95_600_60_69_60_1_69_AP; 100_500_60_69_60_1_50_USA;
- large: 110_500_60_69_60_1_50_AP; 110_700_80_60_89_1_60_USA; 150_800_69_50_80_1_60_AP; 150_900_69_60_80_1_89_USA;
- extra-large: 155_500_60_69_60_1_50_AP; 155_1000_69_60_80_1_69_USA; 195_900_89_89_89_1_69_AP; 190_700_89_69_89_1_89_USA;
- huge: 200_500_60_69_60_1_50_AP; 200_700_80_60_89_1_60_USA; 200_800_69_50_80_1_60_AP;; 250_900_69_60_80_1_89_USA.

4.1.1. Evaluating performance of GVNS with different values of k_{max}

In order to tune the k_{max} parameter, we executed our GVNS with 5 different choices of k_{max}: $\lceil n/2 \rceil$, $\lceil n/4 \rceil$, $\lceil n/8 \rceil$, $\lceil n/16 \rceil$, and $\lceil n/32 \rceil$. For each choice of k_{max}, GVNS was executed 20 times on each instance, each time using a different random seed. In each run, the time parameter t_{max} was set to n seconds. The summary results are provided in Table 1. For each parameter setting, we report the average solution value and the average time-to-target. The averages are calculated over 4 instances from the same set according to the size. The time-to-target refers to the CPU time spent before reaching the final solution returned by our GVNS for the first time.

From Table 1, we may infer that under all settings, GVNS requires a similar time-to-target. In addition, we infer that the for $k_{max} = \lceil n/32 \rceil$ and $k_{max} = \lceil n/2 \rceil$, we have the worst results on the average. These two cases may be considered as the ones with the lowest level of the diversification ($k_{max} = \lceil n/32 \rceil$) and the highest level of the diversification ($k_{max} = \lceil n/2 \rceil$). On the other hand, much better results are obtained at the moderate levels of the diversification ($k_{max} = \lceil n/4 \rceil$, $\lceil n/8 \rceil$ and $\lceil n/16 \rceil$). In these three cases the average solution values and average times-to-target are very similar (e.g., values differ by less than 0.5%). However, the best average value on medium, large and extra large instances as well as the best overall average is attained under the setting $k_{max} = \lceil n/8 \rceil$. Hence, in the rest of testing we set k_{max} to $\lceil n/8 \rceil$.

4.1.2. Evaluating performance of GVNS with different values of T_{max}

In this section, we perform tests with the time limit of GVNS, T_{max}, set to the one of the values from the set $\{n, \lceil n/2 \rceil \lceil n/4 \rceil \lceil n/8 \rceil, \lceil n/16 \rceil\}$. All values are expressed in seconds. For each choice of T_{max}, GVNS was executed 20 times on each instance, each time using a different random seed. In all tests, parameter k_{max} was set to $\lceil n/8 \rceil$ due to findings presented in the precedent section. The summary results are presented in Table 2 and for each parameter

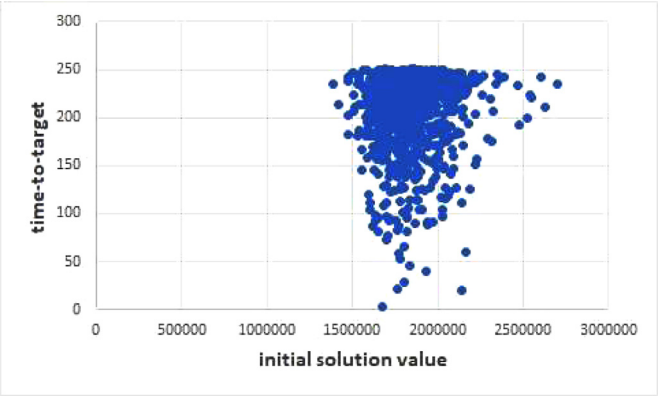

Fig. 1. Impact of initial solution quality on the running time of GVNS.

setting the average solution values and the average times-to-target are reported. Again, the averages are calculated over 4 instances from the same size set. In addition, we report the results found by the state-of-the-art methods, SO and AMP, on the considered instances. According to the reported results, as expected, the more CPU time was allowed to GVNS, the better was the final result obtained. If we compare the average values obtained for $T_{\max} = n$ and $T_{\max} = \lceil n/2 \rceil$, we see that the deviation between these values is about 1% (compare 617326.39 and 625376.99). This is also true if we compare the average values for $T_{\max} = \lceil n/2 \rceil$ and $T_{\max} = \lceil n/4 \rceil$. However, for the smaller values of $T_{\max}$, the average solution values are worse. If we compare times-to-target, we may conclude that for each choice of $T_{\max}$, GVNS consumed a similar fraction of $T_{\max}$ to find the reported solution for the first time. More precisely, as we double $T_{\max}$, the time-to-target usually doubles as well. If we compare the results reported by SO, AMP and GVNS, it follows that for $T_{\max}$ greater or equal to $\lceil n/4 \rceil$, GVNS finds better solutions for the considered instances than SO and AMP. However, since the hub location decisions are decisions at a strategic level, not solving the problem almost instantly may not be critical. Hence, we decide to set $T_{\max}$ to n seconds in all subsequent testings. This means that our GVNS requires 250 sec at most to solve an instance (i.e., this is time needed to solve the largest instance with 250 nodes).

4.2. Evaluating the impact of initial solution quality on the running time of GVNS

The initial solution which is constructed by opening all nodes as hubs (Section 3.2), may be far from the optimal one in most cases. For this reason, such solution is improved by BVND before starting the main GVNS loop. Thus, BVND may be considered here as an improving constructive procedure which returns a solution of good quality. The similar approach was used for example in Gelareh et al. (2019).

In order to check how the quality of initial solution impacts the running time of GVNS, we take the largest instance 250_900_69_60_80_1_89_USA and solve it by GVNS 1000 times. In each run GVNS is fed with a different initial solution. The pool of 1000 initial solutions is constructed by applying the shaking procedure on the initial solution constructed by BVND 1000 times. The shaking parameter k is set to $n/2$. In order to ensure diversity among the solutions, different random seeds are used within the shaking procedure to generate solutions. Fig. 1 depicts the outcome of the experiment. The x-axis provides the values of initial solutions, while the y-axis provides the times-to-target. As we can observe from the figure, sometimes a high quality solution may lead to high time-to-target and the opposite, a low quality solution may lead to low time-to-target. Similar observations are made in the earlier empirical studies conducted on VNS (see e.g., Hansen and Mladenović (2006); Mjirda et al. (2017)). This once again confirms that for VNS, and heuristic in general, a high quality solution may stick a search in the deep local optimum valley, which further implies that huge CPU time investment is required in order to resolve this local optimum trap. On the other hand, if we have a worse solution, such local optimum traps may be luckily avoided and the search may be ended with a much better solution.

4.3. Comparison with the state-of-the-art methods

We compare the proposed GVNS with the method SO based on Strategic Oscillation (Corberán et al. (2016)), the method based on adaptive memory denoted by AMP (Hoff et al. (2017)) and with solutions found by CPLEX MIQCP 12.8 solver used to solve mixed integer quadratically constrained program (MIQCP) presented in Section 2. The time limit for CPLEX MIQCP 12.8 solver was set to 8 h. The computation experiments using CPLEX MIQCP 12.8 solver were performed using CPLEX Callable Library. We have used the following metrics to measure the performance of our procedure:

- **Value**: Average objective value of the best solutions obtained with the algorithm on the instances considered in the experiment;
- **Best**: The number of instances for which a procedure is able to find the best-known solution;
- **CPU**: Average computing time in seconds employed by the algorithm;
- **Dev_Avg_SO**: Average percentage deviation of the value returned by GVNS from the SO value. It is calculated as

  $100 \times (SO\_value - GVNS\_value)/SO\_value$;
- **Dev_Avg_AMP**: Average percentage deviation of the value returned by GVNS from the AMP value. It is calculated as

  $100 \times (AMP\_value - GVNS\_value)/AMP\_value$;
- **Dev_Best_SO**: Percentage deviation of the best solution value returned by the considered method from the SO value. It is calculated as

  $100 \times (SO\_best\_value - GVNS\_best\_value)/SO\_value$;
- **Dev_Best_AMP**: Percentage deviation of the best solution value returned by the considered method from the AMP value. It is calculated as

  $100 \times (AMP\_best\_value - GVNS\_best\_value)/AMP\_value$;
- **Dev_SO_CPLEX**: The percentage deviation of the SO value from the CPLEX value. It is calculated as

  $100 \times (CPLEX\_value - SO\_value)/CPLEX\_value$;
- **Dev_AMP_CPLEX**: The percentage deviation of the AMP value returned by GVNS from the CPLEX value. It is calculated as

  $100 \times (CPLEX\_value - AMP\_value)/CPLEX\_value$;
- **Dev_GVNS_CPLEX**: The percentage deviation of the best solution value returned by GVNS from the CPLEX value. It is calculated as

  $100 \times (CPLEX\_value - GVNS\_best\_value)/CPLEX\_value$.

Now, we compare our procedure, denoted GVNS, with the best previous methods. GVNS is executed 20 times and the obtained results are shown in Tables 3–9.

Table 3 reports the summary results of the comparison between the our procedure and the Strategic Oscillation method, SO, by Corberán et al. (2016), and between the our procedure and AMP_50 (50 corresponds to a termination criteria of 50 executions), by Hoff et al. (2017). For each of the above methods and

Table 3
Comparison with best previous methods.

		Dev (%)		CPU			Best		
Size	Num. Instance	Dev_Avg_SO	Dev_Avg_AMP	SO	AMP	GVNS	SO	AMP	GVNS
Small	45	6.66	3.93	2.96	1.02	5.77	0	5	45
Medium	36	13.10	7.12	37.81	37.24	53.06	0	0	36
Large	27	18.46	7.68	310.36	359.64	108.06	0	0	27
Extra-large	37	18.45	6.24	1606.35	772.55	137.45	0	0	37
Huge	25	21.36	9.77	4916.33	2866.40	186.70	0	0	25
Summary	170	14.63	6.56	1130.69	654.95	87.30	0	5	170

Table 4
Comparison with CPLEX.

	CPLEX		SO			AMP			GVNS		
Instance	Value	CPU	Value	CPU	Dev_SO_CPLEX	Value	CPU	Dev_AMP_CPLEX	Best Value	CPU	Dev_GVNS_CPLEX
10_600_89_60_40_1_60_CAB	**234243**	19.65	237701	0.22	-1.48	234243	0.01	0.00	234243	0.00	0.00
10_700_50_60_8_1_60_AP	**242031**	10.94	273801	0.23	-13.13	249318	0.00	-3.01	242031	0.00	0.00
10_700_50_60_8_1_60_CAB	**243854**	20.83	253255	0.22	-3.86	243854	0.01	0.00	243854	0.00	0.00
10_700_69_40_8_1_50_CAB	**246610**	18.37	257691	0.22	-4.49	246610	0.01	0.00	246610	0.00	0.00
10_800_60_60_6_1_69_AP	**213201**	9.29	245513	0.23	-15.16	232104	0.01	-8.87	213201	0.00	0.00
10_800_60_60_6_1_69_CAB	**238144**	16.75	244417	0.58	-2.63	238144	0.00	0.00	238144	0.00	0.00
10_800_60_80_8_1_80_CAB	**240872**	18.5	247078	0.22	-2.58	240872	0.00	0.00	240872	0.00	0.00
15_500_50_60_40_1_60_CAB	**264000**	7639.54	289339	0.36	-9.60	286027	0.03	-8.34	264000	0.82	0.00
15_600_80_89_6_1_69_CAB	**268618**	19423.08	293075	0.44	-9.10	289839	0.03	-7.90	268618	1.41	0.00
15_600_80_89_8_1_60_CAB	**276975**	28802.29	308168	0.41	-11.26	300251	0.03	-8.40	276975	0.71	0.00
15_700_89_60_40_1_60_CAB	**264299**	6758.23	289953	0.39	-9.71	285965	0.03	-8.20	264299	0.69	0.00
15_800_50_60_40_1_60_CAB	**264449**	13372.56	290026	0.39	-9.67	285274	0.03	-7.87	264449	1.19	0.00
15_900_80_89_40_1_60_CAB	**264596**	7450.27	290134	0.41	-9.65	285390	0.03	-7.86	264596	1.12	0.00
20_700_50_60_8_1_60_AP	386296	28804.29	406266	0.98	-5.17	389589	0.08	-0.85	340118	0.01	11.95
20_700_50_60_8_1_60_CAB	183115	28802.1	176142	1.03	3.81	174048	0.09	4.95	173194	0.30	5.42
20_700_50_60_8_1_60_USA	**117896**	13551.51	127058	0.92	-7.77	117986	0.02	-0.08	117896	0.01	0.00
20_700_69_40_8_1_50_AP	405389	28805.18	408538	1.03	-0.78	408293	0.10	-0.72	355910	0.00	12.21
20_700_69_40_8_1_50_CAB	186146	28802.19	187305	0.84	-0.62	178275	0.09	4.23	177419	0.39	4.69
20_800_60_60_6_1_69_CAB	168675	28802.98	164351	0.73	2.56	164568	0.10	2.43	164270	5.02	2.61
20_800_60_80_8_1_80_AP	357944	28803.56	363247	1.19	-1.48	363247	0.09	-1.48	322126	2.47	10.01
20_800_60_80_8_1_80_CAB	171822	28802.49	170069	0.75	1.02	168204	0.11	2.11	167005	3.29	2.80
20_800_60_80_8_1_80_USA	**115154**	13270.68	121071	0.87	-5.14	115370	0.02	-0.19	115154	0.00	0.00
20_900_80_89_40_1_80_CAB	151650	28803.58	151342	0.80	0.20	151324	0.10	0.21	150191	0.02	0.96
25_600_80_60_6_1_40_CAB	230498	28803.69	210578	1.70	8.64	208444	0.26	9.57	207136	2.84	10.14
25_600_80_89_6_1_69_CAB	206423	28801.13	192213	1.39	6.88	192900	0.27	6.55	190297	0.44	7.81
25_600_80_89_8_1_60_CAB	223489	28803.49	214944	1.64	3.82	206386	0.25	7.65	204157	0.79	8.65
25_650_69_69_6_1_50_CAB	217497	28804.56	210278	1.84	3.32	201993	0.24	7.13	200351	4.66	7.88
25_650_69_69_6_1_70_CAB	174138	28804.4	162862	2.14	6.48	161398	0.32	7.32	156302	5.42	10.24
25_800_89_60_40_1_80_CAB	184882	28804.11	179893	1.86	2.70	179948	0.28	2.67	179817	0.45	2.74
25_900_80_89_40_1_80_CAB	188774	28804.18	181207	1.61	4.01	180600	0.30	4.33	179969	1.44	4.66

for each observed set of instances the average percentage deviation (Column `'Dev'`), the running time in seconds (Column `'CPU'`) (time when the solution reported as final has been found for the first time) and the number of best solutions (Column `'Best'`) are reported.

The last row provides summary results: the total number of instances, average CPU times and percentage deviations, and the total number of best-known solutions found by each method. The detailed comparison of these methods is presented in Tables 5–9 (see Appendix).

Table 3 shows that our procedure gives far better results in comparison with SO and AMP methods.

Our method found 170 best-known solutions of which 165 solutions are newly established by our heuristic. Moreover, our method gives significantly lower solution values, which makes this method more favorable compared to the previous ones. The average improvements over SO and AMP are 14.63% and 6.56%, respectively. These improvements are in some instances greater than 20%. The superiority of our method over the previous methods has been confirmed by a Wilcoxon test. More precisely, we have performed a Wilcoxon test on SO values and ours ($p = 1.206708e^{-29}$), as well as the AMP values and our own ($p = 7.987178e^{-29}$). The resulting p-value < 0.0001 in both cases confirms the superiority of our GVNS.

Our method shows an average computing time of 87.30 sec. An average computing time obtained by SO is 1130.69 sec, while that of AMP is 654.95 sec. It should be emphasized here that SO and AMP heuristics were tested on faster CPUs than ours. SO was executed on an Intel(R) Core(TM) i7-3770 at 3.40 GHz, AMP on an Intel Xeon E3-1270 at 3.40 GHz, while our method was executed on an Intel(R) Core(TM) i5-3470 at 3.20GHz. According tothe Passmark CPU scores available at www.cpubenchmark.net, our CPU has the lowest score, meaning that it is the slowest (the Passmark CPU scores of Intel(R) Core(TM) i7-3770 at 3.40 GHz, Intel Xeon E3-1270 at 3.40 GHz and Intel(R) Core(TM) i5-3470 at 3.20GHz are 9283, 8238 and 6712, respectively).

As we can see, our method gives us results for large instances (large, extra-large, huge) much faster than the previous methods. So, this fact demonstrates the usefulness of auxiliary data structures introduced to explore neighborhood structures.

Moreover, the superiority in terms of computing time of our GVNS over SO and AMP heuristics, has been confirmed by the Wilcoxon test. In both cases, comparing GVNS and SO as well as comparing GVNS and AMP, the Wilcoxon test returned p-values less than 0.0001, which confirms the superiority of our procedure.

It should be noted that our method uses a smaller number of neighborhood structures than the current state-of-the-art AMP

method. Hence, less may yield more (see Mladenović et al. (2016), Brimberg et al. (2017) and Costa et al. (2017)).

Table 4 reports the results comparison among SO, AMP, GVNS and CPLEX MIQCP 12.8 solver used to solve the mathematical formulation presented in Section 2. In particular, the table reports the values found by each method and the computing time used by each method. In addition, the percentage deviations of values provided by SO, AMP and GVNS from the corresponding values found by CPLEX are reported. The results are reported for 30 small instances. For larger instances results are not reported because CPLEX failed to provide a feasible solution either due to the imposed time limit or memory requirements. The optimal solutions values found by CPLEX are boldfaced.

From results in Table 4, it follows that GVNS was able to find optimal solutions on fifteen instances CPLEX does, and in addition, it obtained better solutions for the remaining instances in very short computing times. On the other hand, AMP and SO, exhibited worse performances, failing to reach equal or better solutions than CPLEX on several instances. Moreover, AMP failed to reach 10 known optimal solutions, while SO found none.

5. Conclusions

Hub location problems are optimization problems that greatly attract the attention of the researchers.

In this paper, we have studied a capacitated modular hub location problem. To solve this problem, a heuristic method based on the general variable neighborhood search (GVNS) framework is proposed. A powerful local search is applied within the heuristic that uses a sequential structure with two neighborhoods. One of the neighborhoods is considered in this paper for the first time. In addition, auxiliary data structures are used to explore neighborhoods in an efficient way. We test the heuristic on benchmark instances from the literature. Our experiments show that GVNS is capable of exploring the solution space efficiently and effectively, outperforming existing approaches. The results obtained demonstrate the superiority of the GVNS heuristic over the best previous methods in terms of both solution quality and CPU run time.

As future research direction, we propose to examine other search strategies within GVNS framework such as nested, mixed-nested strategies etc. (see Hansen et al. (2017)). Future research may also apply the GVNS framework to other types of hub location problems. In addition, the problem studied here assumes that the flow between each origin–destination pair is routed via at most two hubs. This assumption may be relaxed in the future work by considering structured non-fully connected backbone networks as the tree-of-hubs or the cycle-of-hubs location problem. Hence, future work may include extensions of the studied problem which would consider different backbone network types and development of corresponding mathematical models and solution approaches.

Acknowledgments

The authors would like to thank three anonymous referees for their careful reading of the manuscript and for their valuable comments and suggestions that have contributed significantly towards improving the article. This research was partially supported by Serbian Ministry of Education, Science and Technological Development under the grants no. 174010, 044006 and 174001. This work also has been supported by ELSAT project, which is co-financed by the European Union with the European Regional Development Fund, the French state and the Hauts de France Region Council.

Appendix

The detailed comparisons of the SO, AMP and GVNS is given in Tables 5–9.

Table 5
Results and comparisons of the SO, AMP and GVNS on small size instances.

	SO		AMP		GVNS						
Instance	Value	CPU	Value	CPU	Best Value	Value	CPU	Dev_Best_SO	Dev_Best_AMP	Dev_Avg_SO	Dev_Avg_AMP
10_600_89_60_40_1_60_CAB	237701	0.22	234243	0.01	234243	234243.00	0.00	1.45	0.00	1.45	0.00
10_700_50_60_8_1_60_AP	273801	0.23	249318	0.00	242031	242031.00	0.00	11.60	2.92	11.60	2.92
10_700_50_60_8_1_60_CAB	253255	0.22	243854	0.01	243854	243854.00	0.00	3.71	0.00	3.71	0.00
10_700_69_40_8_1_50_CAB	257691	0.22	246610	0.01	246610	246610.00	0.00	4.30	0.00	4.30	0.00
10_800_60_60_6_1_69_AP	245513	0.23	232104	0.01	213201	213201.00	0.00	13.16	8.14	13.16	8.14
10_800_60_60_6_1_69_CAB	244417	0.58	238144	0.00	238144	238144.00	0.00	2.57	0.00	2.57	0.00
10_800_60_80_8_1_80_CAB	247078	0.22	240872	0.00	240872	240872.00	0.00	2.51	0.00	2.51	0.00
15_500_50_60_40_1_60_CAB	289339	0.36	286027	0.03	264000	264000.00	0.82	8.76	7.70	8.76	7.70
15_600_80_89_6_1_69_CAB	293075	0.44	289839	0.03	268618	268618.00	1.41	8.34	7.32	8.34	7.32
15_600_80_89_8_1_60_CAB	308168	0.41	300251	0.03	276975	276975.00	0.71	10.12	7.75	10.12	7.75
15_700_89_60_40_1_60_CAB	289953	0.39	285965	0.03	264299	264299.00	0.69	8.85	7.58	8.85	7.58
15_800_50_60_40_1_60_CAB	290026	0.39	285274	0.03	264449	264449.00	1.19	8.82	7.30	8.82	7.30
15_900_80_89_40_1_60_CAB	290134	0.41	285390	0.03	264596	264596.00	1.12	8.80	7.29	8.80	7.29
20_700_50_60_8_1_60_AP	406266	0.98	389589	0.08	340118	340118.00	0.01	16.28	12.70	16.28	12.70
20_700_50_60_8_1_60_CAB	176142	1.03	174048	0.09	173194	173194.00	0.30	1.67	0.49	1.67	0.49
20_700_50_60_8_1_60_USA	127058	0.92	117986	0.02	117896	117896.00	0.01	7.21	0.08	7.21	0.08
20_700_69_40_8_1_50_AP	408538	1.03	408293	0.10	355910	355910.00	0.00	12.88	12.83	12.88	12.83
20_700_69_40_8_1_50_CAB	187305	0.84	178275	0.09	177419	177419.00	0.39	5.28	0.48	5.28	0.48
20_800_60_60_6_1_69_CAB	164351	0.73	164568	0.10	164270	164270.00	5.02	0.05	0.18	0.05	0.18
20_800_60_80_8_1_80_AP	363247	1.19	363247	0.09	322126	324569.95	2.47	11.32	11.32	10.65	10.65
20_800_60_80_8_1_80_CAB	170069	0.75	168204	0.11	167005	167089.00	3.29	1.80	0.71	1.75	0.66
20_800_60_80_8_1_80_USA	121071	0.87	115370	0.02	115154	115154.00	0.00	4.89	0.19	4.89	0.19
20_900_80_89_40_1_80_CAB	151342	0.80	151324	0.10	150191	150191.00	0.02	0.76	0.75	0.76	0.75
25_600_80_60_6_1_40_CAB	210578	1.70	208444	0.26	207136	207550.60	2.84	1.63	0.63	1.44	0.43
25_600_80_89_6_1_69_CAB	192213	1.39	192900	0.27	190297	190297.00	0.44	1.00	1.35	1.00	1.35
25_600_80_89_8_1_60_CAB	214944	1.64	206386	0.25	204157	204360.50	0.79	5.02	1.08	4.92	0.98
25_650_69_69_6_1_50_CAB	210278	1.84	201993	0.24	200351	200744.05	4.66	4.72	0.81	4.53	0.62
25_650_69_69_6_1_70_CAB	162862	2.14	161398	0.32	156302	156323.55	5.42	4.03	3.16	4.01	3.14
25_800_89_60_40_1_80_CAB	179893	1.86	179948	0.28	179817	179830.05	0.45	0.04	0.07	0.03	0.07
25_900_80_89_40_1_80_CAB	181207	1.61	180600	0.30	179969	179969.00	1.44	0.68	0.35	0.68	0.35
30_600_80_89_8_1_60_AP	383188	3.46	351284	0.19	339114	339114.00	2.94	11.50	3.46	11.50	3.46
30_700_69_40_8_1_50_USA	211640	2.93	211382	0.42	187883	188111.55	11.48	11.23	11.12	11.12	11.01
35_600_80_89_8_1_60_AP	448064	3.98	438722	1.07	411669	418633.50	4.97	8.12	6.17	6.57	4.58
35_600_80_89_8_1_60_USA	206472	4.40	202473	0.60	201456	201466.35	2.38	2.43	0.50	2.42	0.50
35_700_80_50_8_1_69_AP	436550	4.23	432959	1.06	414636	415244.00	4.92	5.02	4.23	4.88	4.09
40_600_80_89_8_1_60_USA	288503	6.19	276510	2.92	268109	268239.95	14.28	7.07	3.04	7.02	2.99
40_700_80_50_8_1_69_AP	478470	6.43	458982	1.86	446397	447314.95	13.36	6.70	2.74	6.51	2.54
40_700_80_50_8_1_69_USA	282629	6.01	270601	2.83	263301	263479.05	16.64	6.84	2.70	6.78	2.63
45_600_80_89_8_1_60_AP	521958	8.44	515630	2.69	475399	479020.60	17.69	8.92	7.80	8.23	7.10
45_700_69_40_8_1_50_AP	589832	8.63	586276	2.70	527410	533589.00	16.88	10.58	10.04	9.54	8.99
45_700_69_40_8_1_50_USA	345335	6.83	301809	4.23	294768	295998.25	20.85	14.64	2.33	14.29	1.93
50_600_80_89_8_1_60_USA	347675	12.34	318016	6.24	311022	312238.10	19.99	10.54	2.20	10.19	1.82
50_700_69_40_8_1_50_AP	598636	11.50	599205	4.93	535571	541255.95	30.49	10.53	10.62	9.59	9.67
50_700_80_50_8_1_69_AP	562786	11.23	545111	4.61	487366	491959.55	27.57	13.40	10.59	12.58	9.75
50_700_80_50_8_1_69_USA	328853	10.87	316654	6.64	303791	304886.15	21.56	7.62	4.06	7.28	3.72

Table 6
Results and comparisons of the SO, AMP and GVNS on medium size instances.

	SO		AMP		GVNS						
Instance	Value	CPU	Value	CPU	Best Value	Value	CPU	Dev_Best_SO	Dev_Best_AMP	Dev_Avg_SO	Dev_Avg_AMP
55_500_60_69_60_1_50_AP	551996	12.28	528434	7.32	462324	474258.85	36.42	16.25	12.51	14.08	10.25
55_500_60_69_60_1_50_USA	356419	11.51	313285	8.42	308568	309088.45	23.53	13.43	1.51	13.28	1.34
55_800_69_50_80_1_60_AP	627685	13.65	623632	6.71	558601	564359.20	32.01	11.01	10.43	10.09	9.50
55_800_69_50_80_1_60_USA	356039	10.31	324208	8.80	317505	319791.85	27.43	10.82	2.07	10.18	1.36
60_500_60_69_60_1_50_AP	597551	14.23	556315	9.56	508142	512160.95	34.53	14.96	8.66	14.29	7.94
60_600_60_69_60_1_69_USA	324245	14.34	310881	12.01	299143	301254.85	35.21	7.74	3.78	7.09	3.10
60_800_69_50_80_1_60_AP	664374	13.32	643873	10.31	592124	597716.15	39.30	10.87	8.04	10.03	7.17
60_800_69_50_80_1_60_USA	375981	14.96	337606	11.04	325076	327100.60	39.09	13.54	3.71	13.00	3.11
65_500_60_69_60_1_50_AP	594968	19.49	591705	13.08	540231	548160.45	45.41	9.20	8.70	7.87	7.36
65_600_60_69_60_1_69_AP	596768	19.06	572237	12.69	528391	537145.25	37.67	11.46	7.66	9.99	6.13
65_600_60_69_60_1_69_USA	366133	21.09	330006	17.42	305931	319389.10	38.34	16.44	7.30	12.77	3.22
65_800_69_50_80_1_60_USA	405756	21.08	366629	16.35	329712	343814.05	44.68	18.74	10.07	15.27	6.22
70_500_60_69_60_1_50_AP	664745	24.56	613848	16.81	586095	592651.90	48.85	11.83	4.52	10.85	3.45
70_600_60_69_60_1_69_AP	615656	22.25	600082	18.08	567843	577114.15	41.02	7.77	5.37	6.26	3.83
70_600_60_69_60_1_69_USA	383492	29.03	345887	21.84	335079	347997.65	35.68	12.62	3.12	9.26	-0.61
70_800_69_50_80_1_60_AP	769108	27.27	740056	18.55	683197	692843.15	50.41	11.17	7.68	9.92	6.38
75_500_60_69_60_1_50_USA	552080	29.14	524031	31.68	413458	417824.50	49.62	25.11	21.10	24.32	20.27
75_600_60_69_60_1_69_AP	665111	35.23	650554	22.03	601062	611731.05	57.12	9.63	7.61	8.03	5.97
75_600_60_69_60_1_69_USA	519474	31.44	475886	30.33	394072	400542.85	53.26	24.14	17.19	22.89	15.83
75_800_69_50_80_1_60_AP	828245	32.87	787150	21.92	715753	731646.75	55.69	13.58	9.07	11.66	7.05
80_500_60_69_60_1_50_AP	722632	40.25	692408	29.76	655138	664824.55	59.25	9.34	5.38	8.00	3.98
80_500_60_69_60_1_50_USA	632672	35.09	586997	45.36	483450	485892.50	63.70	23.59	17.64	23.20	17.22
80_800_69_50_80_1_60_AP	837210	39.53	815360	31.50	770193	779047.85	55.08	8.00	5.54	6.95	4.45
80_800_69_50_80_1_60_USA	673561	36.57	615377	46.32	498337	503758.80	62.24	26.01	19.02	25.21	18.14
85_500_60_69_60_1_50_AP	762920	62.71	760980	38.37	698844	714903.25	57.44	8.40	8.17	6.29	6.05
85_500_60_69_60_1_50_USA	777825	53.17	690770	74.37	585420	590433.70	71.63	24.74	15.25	24.09	14.53
85_800_69_50_80_1_60_AP	903683	60.42	871710	38.25	821306	832180.30	66.44	9.12	5.78	7.91	4.53
90_500_60_69_60_1_50_USA	804494	53.98	695573	80.35	594821	600080.15	76.92	26.06	14.48	25.41	13.73
90_600_60_69_60_1_69_AP	759377	70.56	736086	48.27	696307	717533.65	62.43	8.31	5.40	5.51	2.52
90_600_60_69_60_1_69_USA	708086	54.34	642502	72.99	564666	570261.45	67.51	20.25	12.11	19.46	11.24
90_800_69_50_80_1_60_AP	966669	70.72	903969	51.35	864325	876440.35	69.54	10.59	4.39	9.33	3.05
95_500_60_69_60_1_50_AP	905808	75.52	815931	56.49	786329	798399.90	69.95	13.19	3.63	11.86	2.15
95_500_60_69_60_1_50_USA	780646	64.88	691424	103.14	616060	632274.05	71.12	21.08	10.90	19.01	8.55
95_600_60_69_60_1_69_AP	826451	82.84	798566	56.01	754236	758446.45	77.61	8.74	5.55	8.22	5.02
95_600_60_69_60_1_69_USA	714260	65.63	651071	102.89	584572	599009.20	74.12	18.16	10.21	16.14	8.00
100_500_60_69_60_1_50_USA	792405	77.82	713698	150.43	644955	659063.95	79.86	18.61	9.63	16.82	7.65

Table 7
Results and comparisons of the SO, AMP and GVNS on large size instances.

	SO		AMP		GVNS						
Instance	Value	CPU	Value	CPU	Best Value	Value	CPU	Dev_Best_SO	Dev_Best_AMP	Dev_Avg_SO	Dev_Avg_AMP
110_500_60_69_60_1_50_AP	996975	150.65	954372	108.51	885138	899690.75	89.11	11.22	7.25	9.75	5.73
110_600_60_69_60_1_69_AP	934826	142.15	906009	118.71	844228	858263.20	96.13	9.69	6.82	8.19	5.27
110_700_80_60_89_1_60_USA	1122255	105.91	915625	265.08	797473	814999.65	92.67	28.94	12.90	27.37	10.99
110_800_69_50_80_1_60_USA	1039753	117.52	903544	228.90	783675	806214.15	91.82	24.63	13.27	22.46	10.77
110_900_69_50_89_1_60_USA	1105052	105.43	938808	258.70	814244	837426.50	102.88	26.32	13.27	24.22	10.80
120_500_60_69_60_1_50_AP	1123004	206.13	1031226	164.92	960001	972060.25	94.33	14.51	6.91	13.44	5.74
120_500_60_69_60_1_50_USA	1117652	151.31	948970	328.15	828556	856817.10	105.47	25.87	12.69	23.34	9.71
120_700_80_60_89_1_60_USA	1219108	163.57	1010735	389.88	884353	930874.85	106.90	27.46	12.50	23.64	7.90
120_900_69_50_89_1_60_USA	1175745	146.52	1067450	415.56	890011	924849.60	102.46	24.30	16.62	21.34	13.36
125_500_60_69_60_1_50_AP	1145428	192.06	1085187	192.29	993929	1003098.50	105.91	13.23	8.41	12.43	7.56
125_800_69_50_80_1_60_AP	1371476	208.84	1244985	207.28	1170346	1192964.65	101.86	14.67	6.00	13.02	4.18
130_600_60_69_60_1_69_AP	1158759	291.91	1068564	262.15	996212	1018072.40	104.86	14.03	6.77	12.14	4.73
130_600_60_69_60_1_69_USA	1040651	180.23	949318	446.82	843063	906759.25	116.96	18.99	11.19	12.87	4.48
130_800_69_50_80_1_60_USA	1264063	227.08	1061879	481.97	923684	959534.85	108.91	26.93	13.01	24.09	9.64
135_600_60_69_60_1_69_AP	1229722	335.92	1119159	316.02	1036416	1051339.75	109.55	15.72	7.39	14.51	6.06
135_800_69_50_80_1_60_USA	1405752	251.49	1066218	672.35	952019	971985.90	119.43	32.28	10.71	30.86	8.84
140_500_60_69_60_1_50_AP	1412597	357.75	1252721	370.81	1141082	1159504.00	107.75	19.22	8.91	17.92	7.44
140_700_80_60_89_1_60_USA	1490007	286.76	1199339	335.99	1012166	1087732.10	130.50	32.07	15.61	27.00	9.31
140_800_69_50_80_1_60_AP	1570932	377.44	1400828	395.23	1299407	1325545.00	119.15	17.28	7.24	15.62	5.37
140_900_69_50_89_1_60_USA	1444192	275.01	1213396	350.83	1030811	1077125.15	128.20	28.62	15.05	25.42	11.23
145_600_80_69_60_1_50_AP	1462902	352.66	1343050	452.93	1218987	1245029.20	106.46	16.67	9.24	14.89	7.30
145_600_80_69_60_1_50_USA	596573	710.75	492827	430.37	461934	480880.55	93.48	22.57	6.27	19.39	2.42
145_800_69_50_80_1_60_AP	1603039	357.12	1480588	456.14	1340027	1381779.80	116.82	16.41	9.49	13.80	6.67
145_800_69_50_80_1_60_USA	654681	647.73	587954	412.61	527006	541510.95	101.32	19.50	10.37	17.29	7.90
150_1000_69_60_80_1_69_USA	612031	837.07	553365	499.78	500649	511838.20	116.87	18.20	9.53	16.37	7.50
150_800_69_50_80_1_60_AP	1734477	424.82	1563951	646.73	1414743	1425748.1	127.50	18.43	9.54	17.80	8.84
150_900_69_60_80_1_89_USA	606912	775.96	530125	501.58	460547	467836.1	120.19	24.12	13.12	22.93	11.76

Table 8
Results and comparisons of the SO, AMP and GVNS on extra-large size instances.

	SO		AMP		GVNS						
Instance	Value	CPU	Value	CPU	Best Value	Value	CPU	Dev_Best_SO	Dev_Best_AMP	Dev_Avg_SO	Dev_Avg_AMP
155_1000_69_60_80_1_69_USA	650419	919.46	563085	551.45	508757	518557.25	123.29	21.78	9.65	20.27	7.90
155_500_60_69_60_1_50_AP	1427231	635.50	1206831	557.88	1134570	1137122.7	125.05	20.51	5.99	19.91	5.78
155_800_69_50_80_1_60_AP	1602365	604.55	1371004	558.30	1330656	1354263.75	126.22	16.96	2.94	15.48	1.22
160_600_60_69_60_1_69_AP	685604	950.80	595141	373.73	548103	550911.75	118.04	20.06	7.90	19.65	7.43
160_700_80_60_89_1_60_USA	704872	1010.66	534598	758.45	504782	519238.55	140.94	28.39	5.58	26.34	2.87
160_800_69_50_80_1_60_AP	826772	1042.87	739445	383.80	692752	698629.20	109.08	16.21	6.31	15.50	5.52
160_900_80_50_60_1_69_AP	775178	1008.39	733757	398.09	672197	674808.30	114.69	13.28	8.39	12.95	8.03
160_900_89_50_60_1_69_USA	530376	1048.41	483740	663.18	428126	444753.25	132.82	19.28	11.50	16.14	8.06
165_1000_69_60_80_1_69_USA	665409	1302.26	589474	734.50	537354	551992.25	145.63	19.24	8.84	17.04	6.36
165_800_69_50_80_1_60_AP	876244	1109.94	725874	405.87	666657	671511.30	115.08	23.92	8.16	23.36	7.49
165_800_69_50_80_1_60_USA	704389	1302.59	643904	742.05	565765	579758.10	122.97	19.68	12.14	17.69	9.96
170_500_60_69_60_1_50_AP	714822	1152.12	550621	504.30	526730	531015.35	130.47	26.31	4.34	25.71	3.56
170_600_89_60_69_1_80_USA	551600	1304.46	495349	772.88	435001	464120.55	135.22	21.14	12.18	15.86	6.30
170_700_80_60_89_1_60_USA	609581	1361.95	561585	968.87	491041	522690.60	145.19	19.45	12.56	14.25	6.93
170_900_69_60_80_1_89_USA	613150	1403.26	578471	735.98	509648	532364.95	137.01	16.88	11.90	13.18	7.97
170_900_80_50_60_1_69_AP	754262	1251.03	718370	562.75	659823	662272.55	132.09	12.52	8.15	12.20	7.81
175_500_60_69_60_1_50_AP	649148	1427.86	598241	554.70	538843	544367.75	140.13	16.99	9.93	16.14	9.01
175_600_60_69_60_1_69_AP	648587	1664.50	611906	591.87	543091	553413.95	138.84	16.27	11.25	14.67	9.56
175_800_69_50_80_1_60_USA	716778	1625.83	653231	915.04	592725	606469.75	143.54	17.31	9.26	15.39	7.16
175_900_69_60_80_1_89_USA	602257	1434.86	560301	919.48	513347	537165.55	151.18	14.76	8.38	10.81	4.13
180_1000_69_60_80_1_69_USA	740453	1970.72	639398	938.22	571404	593325.85	138.02	22.83	10.63	19.87	7.21
180_600_60_69_60_1_69_AP	746918	1804.62	613648	673.60	565706	571364.20	150.57	24.26	7.81	23.50	6.89
180_600_89_60_69_1_80_USA	558479	1739.88	495902	1122.06	457943	479568.35	149.75	18.00	7.65	14.13	3.29
180_800_89_69_89_1_89_AP	797621	1921.67	739750	695.90	670205	677483.15	151.46	15.97	9.40	15.06	8.42
185_500_60_69_60_1_50_AP	858506	1872.10	614628	691.73	577981	583323.20	142.41	32.68	5.96	32.05	5.09
185_600_80_89_89_1_89_AP	700282	2029.67	651382	672.15	604816	613279.70	141.59	13.63	7.15	12.42	5.85
185_600_89_60_69_1_80_USA	530783	1880.00	488142	1202.84	438655	460694.50	161.08	17.36	10.14	13.20	5.62
185_800_69_50_80_1_60_AP	890070	1971.64	791304	688.01	734318	742650.75	144.01	17.50	7.20	16.56	6.15
185_900_69_60_80_1_89_USA	621650	2158.96	555466	1004.08	515851	529610.10	148.65	17.02	7.13	14.81	4.65
190_600_60_69_60_1_69_AP	773840	2200.58	633895	792.25	588436	598046.45	143.67	23.96	7.17	22.72	5.66
190_600_80_89_89_1_89_AP	800719	2288.98	654744	812.07	607448	618548.35	142.25	24.14	7.22	22.75	5.53
190_600_89_60_69_1_80_USA	522508	2190.87	485190	1515.13	444360	464953.95	151.26	14.96	8.42	11.01	4.17
190_700_89_69_89_1_89_USA	575933	2177.05	511453	1490.72	475010	483766.20	160.62	17.52	7.13	16.00	5.40
190_800_69_50_80_1_60_AP	1031614	2298.88	798755	800.36	742804	749382.55	145.88	28.00	7.00	27.36	6.18
195_600_60_69_60_1_69_AP	774165	2487.11	645712	941.72	594906	605348.00	123.83	23.16	7.87	21.81	6.25
195_800_69_50_80_1_60_AP	1107739	2431.53	822538	943.84	750916	759244.25	137.45	32.21	8.71	31.46	7.69
195_900_89_89_89_1_69_AP	1093874	2449.22	852008	946.40	794826	800052.50	125.54	27.34	6.71	26.86	6.10

Table 9
Results and comparisons of the SO, AMP and GVNS on huge size instances.

	SO		AMP		GVNS						
Instance	Value	CPU	Value	CPU	Best Value	Value	CPU	Dev_Best_SO	Dev_Best_AMP	Dev_Avg_SO	Dev_Avg_AMP
200_500_60_69_60_1_50_AP	661633	2686.72	635108	1045.68	507726	510934.95	146.27	23.26	20.06	22.77	19.55
200_700_80_60_89_1_60_USA	743444	2976.42	610310	1927.87	589535	601545.25	157.30	20.70	3.40	19.30	1.43
200_700_89_69_89_1_89_USA	608272	2985.89	539340	1852.44	524124	545375.55	167.20	13.83	2.82	10.34	-1.12
200_800_69_50_80_1_60_AP	763358	2731.83	738765	1068.13	576823	582509.05	150.17	24.44	21.92	23.69	21.15
200_800_89_69_89_1_89_USA	645449	2995.71	555040	1895.10	527897	554984.15	168.25	18.21	4.89	14.02	0.01
205_800_69_50_80_1_60_USA	800686	3086.01	693477	1681.57	595820	615530.60	173.19	25.59	14.08	23.12	11.24
205_900_69_60_80_1_89_USA	670343	3570.73	608148	1705.52	521221	544901.55	171.67	22.25	14.29	18.71	10.40
210_800_69_50_80_1_60_USA	811601	3832.57	730273	1790.77	596435	627439.65	161.73	26.51	18.33	22.69	14.08
210_900_69_60_80_1_89_USA	699431	3743.74	593529	1743.47	528108	546343.45	164.18	24.49	11.02	21.89	7.95
215_800_69_50_80_1_60_USA	897348	3927.14	758429	2085.28	632980	665461.20	184.61	29.46	16.54	25.84	12.26
215_900_69_60_80_1_89_USA	736413	3653.79	619427	1928.71	545942	571192.40	191.88	25.86	11.86	22.44	7.79
220_800_69_50_80_1_60_USA	872528	4275.09	723734	2473.09	628389	647428.90	195.77	27.98	13.17	25.80	10.54
220_900_69_60_80_1_89_USA	725843	4485.24	647915	2328.85	557527	573798.50	187.12	23.19	13.95	20.95	11.44
225_800_69_50_80_1_60_USA	978703	4993.80	868963	2775.11	706599	739865.85	190.97	27.80	18.68	24.40	14.86
225_900_69_60_80_1_89_USA	812905	4668.05	732658	2681.13	631859	657980.25	196.22	22.27	13.76	19.06	10.19
230_800_69_50_80_1_60_USA	994501	6096.23	858609	3213.37	715126	748980.55	197.94	28.09	16.71	24.69	12.77
230_900_69_60_80_1_89_USA	859790	5934.23	726502	3065.26	633090	660826.60	206.59	26.37	12.86	23.14	9.04
235_800_69_50_80_1_60_USA	1067897	6223.85	967074	3838.36	788643	836837.25	193.99	26.15	18.45	21.64	13.47
235_900_69_60_80_1_89_USA	967918	6017.28	795102	3698.01	712840	738641.40	206.16	26.35	10.35	23.69	7.10
240_800_69_50_80_1_60_USA	1106592	6079.38	981444	4447.55	820219	863668.15	205.13	25.88	16.43	21.95	12.00
240_900_69_60_80_1_89_USA	914468	6852.66	798589	4358.11	748671	764124.20	213.66	18.13	6.25	16.44	4.32
245_800_69_50_80_1_60_USA	1209802	7913.75	970942	5020.98	805618	865575.95	197.86	33.41	17.03	28.45	10.85
245_900_69_60_80_1_89_USA	913041	7197.28	823255	4956.09	722742	756280.90	198.66	20.84	12.21	17.17	8.14
250_800_69_50_80_1_60_USA	1132659	7800.79	997938	4978.50	822817	882979.55	215.68	27.36	17.55	22.04	11.52
250_900_69_60_80_1_89_USA	991167	8180.05	819535	5100.97	732975	746082.45	225.36	26.05	10.56	24.72	8.96

References

Alumur, S., Kara, B., 2008. Network hub location problems: the state of the art. Eur. J. Opera. Res. 190 (1), 1–21.

Brimberg, J., Mladenović, N., Todosijević, R., Urošević, D., 2017. Less is more: solving the max-mean diversity problem with variable neighborhood search. Inform. Sci. 382, 179–200.

Brimberg, J., Mladenović, N., Todosijević, R., Urošević, D., 2019. A non-triangular hub location problem. Optim. Lett. doi:10.1007/s11590-019-01392-2.

Campbell, J.F., 1994. Integer programming formulations of discrete hub location problems. Eur. J. Opera. Res. 72 (2), 387–405.

Campbell, J.F., O'Kelly, M., 2012. Twenty-five years of hub location research. Transp. Sci. 46 (2), 153–169.

Contreras, I., Fernndez, E., Marin, A., 2010. The tree of hubs location problem. Eur. J. Opera. Res. 202 (2), 390–400.

Contreras, I., Tanash, M., Vidyarthi, N., 2017. Exact and heuristic approaches for the cycle hub location problem. Ann. Opera. Res. 258 (2), 655–677.

Corberán, A., Peiró, J., Campos, V., Marti, R., 2016. Strategic oscillation for the capacitated hub location problem with modular links. J. Heurist. 22 (2), 221–244.

Costa, L.R., Aloise, D., Mladenović, N., 2017. Less is more: basic variable neighborhood search heuristic for balanced minimum sum-of-squares clustering. Inform. Sci. 415, 247–253.

Ernst, A., Krishnamoorth, M., 1996. Efficient algorithms for the uncapacitated single allocation p-hub median problem. Locat. Sci. 4 (3), 139–154.

Farahani, R.Z., Hekmatfar, M., Boloori, A., Nikbakhsh, E., 2013. Hub location problems: a review of models, classification, solution techniques, and applications. Comput. Indus. Eng. 64 (4), 1096–1109.

Gelareh, S., Gendron, B., Hanafi, S., Neamatian Monemi, R., Todosijević, R., 2019. The selective traveling salesman problem with draft limits. J. Heurist. doi:10.1007/s10732-019-09406-z.

Hansen, P., Mladenović, N., 2006. First vs. best improvement: an empirical study. Discret. Appl. Math. 154 (5), 802–817.

Hansen, P., Mladenović, N., Todosijević, R., Hanafi, S., 2017. Variable neighborhood search: basics and variants. EURO J. Comput. Optim. 5 (3), 423–454.

Hoff, A., Peiró, J., Corberán, A., Martí, R., 2017. Heuristics for the capacitated modular hub location problem. Comput. Opera. Res. 86, 94–109.

Ilic, A., Urosevic, D., Brimberg, J., Mladenovic, N., 2010. A general variable neighborhood search for solving the uncapacitated single allocation p-hub median problem. Eur. J. Opera. Res. 206 (2), 289–300.

Mahmutogullari, A.I., Kara, B.Y., 2015. Hub location problem with allowed routing between nonhub nodes. Geogr. Anal. 47 (4), 410–430.

Mjirda, A., Todosijević, R., Hanafi, S., Hansen, P., Mladenović, N., 2017. Sequential variable neighborhood descent variants: an empirical study on the traveling salesman problem. Int. Trans. Opera. Res. 24 (3), 615–633.

Mladenović, N., Hansen, P., 1997. Variable neighborhood search. Comput. Opera. Res. 24 (11), 1097–1100.

Mladenović, N., Todosijević, R., Urošević, D., 2016. Less is more: basic variable neighborhood search for minimum differential dispersion problem. Inform. Sci. 326, 160–171.

Mohri, S.S., Karimi, H., Kordani, A.A., Nasrollahi, M., 2018. Airline hub-and-spoke network design based on airport capacity envelope curve: a practical view. Comput. Indus. Eng. 125, 375–393.

Momayezi, F., Chaharsooghi, S.K., Sepehri, M.M., Kashan, A.H., 2018. The capacitated modular single-allocation hub location problem with possibilities of hubs disruptions: modeling and a solution algorithm. Opera. Res. 1–28.

O'Kelly, M.E., 1987. A quadratic integer program for the location of interacting hub facilities. Eur. J. Opera. Res. 32 (3), 393–404.

Peiró, J., Corberán, A., Marti, R., 2014. GRASP for the uncapacitated r-allocation p-hub median problem. Comput. Opera. Res. 43 (1), 50–60.

Rastani, S., Setak, M., Karimi, H., 2016. Capacity selection for hubs and hub links in hub location problems. Int. J. Manag. Sci. Eng. Manag. 1 (3), 123–133.

Taherkhani, G., Alumur, S., 2018. Profit maximizing hub location problems. Omega. doi:j.omega.2018.05.016

Tanash, M., Contreras, I., Vidyarthi, N., 2017. An exact algorithm for the modular hub location problem with single assignments. Comput. Opera. Res. 85, 32–44.

Yaman, H., Carello, G., 2005. Solving the hub location problem with modular link capacities. Comput. Opera. Res. 32, 3227–3245.