Adaptive Memory Programming for the Capacitated Modular Hub Location Problem

Arild Hoff

Molde University College, Norway. Arild.Hoff@hiMolde.no

Juanjo Peiró

Universitat de València, Spain. Juanjo.Peiro@uv.es

Ángel Corberán

Universitat de València, Spain. Angel.Corberan@uv.es

Rafael Martí

Universitat de València, Spain. Rafael.Marti@uv.es

Abstract

In this paper we study the hub location problem. The goal is to identify an optimal subset of facilities (hubs) to minimize the transportation cost while satisfying certain capacity constraints. In particular, we target the single assignment version, in which each node in the transportation network is assigned to only one hub to route its traffic. We consider here a realistic variant introduced previously, in which the capacity of edges between hubs is increased in a modular way. This reflects the practical situation in air traffic where the number of flights between two locations implies a capacity in terms of number of passengers. Then, the capacity can be increased in a modular way, as a factor of the number of flights. We propose heuristic methods to obtain high-quality solutions in short computational times. Specifically, we implement memory structures to create advanced search methods and compare them with previous heuristics on a set of benchmark instances. Memory structures have been widely implemented in the context of the tabu search methodology, usually embedded in local search algorithms. In this paper we explore an alternative design in which memory structures constitute the core of the constructive method and also of a path relinking post-processing. Statistical tests confirm the superiority of our proposal with respect to previous developments.

Keywords

Hub location, modular links, heuristic algorithms, memory structures.

1. Introduction

Let $G = (V, E)$ be a network, being V the set of nodes ($|V| = n$), and E the set of edges. For any pair of nodes $i, j \in V$, t_{ij} denotes the traffic to be transported from i to j. As it is customary in hub location problems, direct transportation between nodes is not possible, and therefore the traffic needs to be routed through the nodes designed as hubs. This is especially evident in air transportation in which is impractical to schedule a flight between every pair of cities in a region due to its high operational cost. Air flight companies thus resort to route their passengers through intermediate airports usually called hubs. Therefore, in this problem we first have to select which nodes in the network will act as transportation hubs. We will then distinguish between terminal and hub nodes.

In this paper we consider the *single assignment hub location problem*, which specifies that a terminal can only be assigned to a single hub. Assigning a terminal i to a hub k has a cost C_{ik}. For the sake of simplicity, we consider that each hub is assigned to itself. Each hub has a maximum transit capacity Q^h, which indirectly limits the number of terminals that can be assigned to it, in terms of their total traffic. In this variant of the problem, the number of hubs is not specified beforehand, and opening a hub at node i has a fixed installation cost C_{ii}.

There are two types of edges between nodes: access edges, used to connect terminals with hubs, and backbone edges, used to connect hubs with other hubs. Each backbone edge has a maximum traffic capacity of Q^b (in each direction), which has to be understood as a capacity factor. This characteristic models the real situation of the number of seats in a plane that operates in a route, for example. The total capacity of the route can be increased by adding more flights. In graph terms we will say that we add several copies of the backbone edge. Let R_{kl} be the cost of each copy of the edge connecting hubs k and l (i.e., the cost of a flight between k and l in our example). Therefore, if we represent by w_{kl} the number of edges (flights) between k and l, the maximum number of passengers that we can transport in this route is $Q^b w_{kl}$, and its associated cost $w_{kl} R_{kl}$. In short, the hub location problem considered in this paper consists of selecting a subset of nodes to be hubs, and assigning the rest of the nodes to them, in such a way the transportation cost is minimized while satisfying the capacity constraints.

Yaman and Carello (2005) introduced this hub location problem under the name “capacitated single assignment hub location problem with modular link capacities (CSHLPMLC)”. These authors formulated it as a mixed integer non-linear program as follows:

Consider the variables:

- the assignment variable x_{ik} is equal to 1 if terminal i is assigned to hub k, and 0 otherwise. If node i receives a hub, then x_{ii} takes value 1,
- z_{kl} is the traffic on an arc $(k, l) \in A$,
- w_{kl} is the number of copies of the edge $\{k, l\} \in E$.

The formulation is, then:

$$\min \sum_{i \in V} \sum_{k \in V} C_{ik} x_{ik} + \sum_{\{k,l\} \in E} R_{kl} w_{kl} \quad (1)$$

Subject to:

$$\sum_{k \in V} x_{ik} = 1, \forall i \in V \quad (2)$$

$$x_{ik} \leq x_{kk}, \forall i \in V, \forall k \in V \setminus \{i\} \quad (3)$$

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

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

$$Q^b w_{kl} \geq z_{kl}, \forall \{k,l\} \in E \quad (6)$$

$$Q^b w_{kl} \geq z_{lk}, \forall \{k,l\} \in E \quad (7)$$

$$x_{ik} \in \{0,1\}, \forall i,k \in V \quad (8)$$

$$w_{kl} \in \mathbb{Z}_+, \forall \{k,l\} \in E \quad (9)$$

$$z_{kl} \geq 0, \forall (k,l) \in A. \quad (10)$$

Constraints (2) imply that each node needs to be assigned to only one hub. Constraints (3) force node k to be a hub if a node i is assigned to it. Constraints (4) specify that the capacity of a given hub k cannot be less than the amount of traffic that transits through it, thus prohibiting allocations to k beyond its capacity Q^h. Constraints (5) add up the traffics through a given arc (k,l). Finally, constraints (6) and (7) fix the number of copies needed of each backbone edge.

Yaman and Carello (2005) also proposed a branch-and-cut algorithm and a heuristic method to solve this problem. The heuristic method, based on the tabu search methodology, feeds the branch-and-cut with a good initial upper bound. In particular, the solution provided by the metaheuristic is used to limit the number of variables considered by the exact method, by identifying a subset of nodes that represents the best potential locations for the hubs. The hubs selected in the best solution belong to this subset, as well as the two other hubs which appear most often in the best solutions found by the metaheuristic.

Corberán et al. (2016) proposed a heuristic method, based on a strategic oscillation over the search space, to solve the CSHLPMLC. In particular, their method iteratively constructs and partially destructs a solution. In this way, hubs are selected and deselected in search of the optimal set of hubs. This procedure is coupled with two local searches, one based on swapping the assignment of terminals to hubs, and another based on exchanges of terminals to a different hub. Their computational experimentation showed that this method outperforms the tabu search heuristic in Yaman and Carello (2005), and it is able to match the optimal solutions in the small size instances that CPLEX is able to solve.

In this paper we propose a new heuristic based on *Adaptive Memory Programming*. We basically introduce memory structures to enhance the performance of our methods. Memory-based strategies, which are the hallmark of the well-known tabu search methodology (Glover and Laguna 1997), and coined under the term adaptive memory programming, are founded on a quest for "integrating principles," by which alternative forms of memory are appropriately combined with effective strategies for exploiting them. Specifically, we propose different construction and local search methods and study the effectiveness of memory structures, and compare them with memory-less variants. Our experiments show that the adaptive memory features are capable of searching the solution space economically and effectively. Since local choices are guided by information collected during the search, these methods contrast with memoryless designs that heavily rely on semi-random processes that implement a form of sampling.

2. Construction Method

A solution S for our problem consists of a set of hubs H and an assignment of each terminal to a hub. Note that with this assignment, the routing of the traffics between any pair of nodes is univocally determined through their respective hubs.

Corberán et al. (2016) proposed a constructive method to obtain an initial solution that selects nodes to be hubs in a greedy fashion. Specifically, the authors considered an evaluation function to discriminate among candidate nodes based on the costs. Their method iteratively selects the hub nodes until there are enough hubs (in terms of capacity) to assign all the nodes in the network. An important characteristic of this method is that, each time a node is selected as a hub, it performs the associated assignment of terminals to this hub in a greedy fashion ignoring future hub selections. In this paper, we propose an alternative construction method that performs the assignment step after the hub selection, thus taking into account the complete set of hubs.

Our construction method starts by estimating the number of hubs p that provides enough capacity to assign all the terminals. We basically consider the total traffic between all pairs i, j of nodes, and divide it by the hub capacity Q^h . In mathematical terms:

$$p = \left\lceil \frac{\sum_{i,j \in V} t_{ij}}{Q^h} \right\rceil$$

Note that traffics t_{ii} can take a positive value in some applications, as in the case of postal deliveries sent to a central hub for sorting.

Since, for example, nodes with large amount of traffics may not be assigned to the same hub, the value of p above can underestimate the required number of hubs to route all the traffic in the network. In particular, we compute the number of nodes for which their traffic exceeds half of the hub capacity:

$$\sum_{j \in V \setminus \{i\}} t_{ij} + \sum_{j \in V \setminus \{i\}} t_{ji} + t_{ii} > \frac{Q^h}{2}.$$

It is likely that two nodes verifying the expression above do not share the same hub since the sum of their traffics may be larger than Q^h. Therefore, if the number of nodes verifying this expression is larger than our estimation of p, we change p to be this number of nodes.

The method we propose here is a multi-start algorithm. Many multi-start methods in combinatorial optimization resort to randomization to perform multiple constructions. Among them, GRASP methodology (Festa and Resende, 2011) is probably one of the most popular. However, we have developed our constructive algorithm under a different paradigm: adaptive memory. Instead of randomization, we apply frequency values, which record past hub appearances to discourage their selection in future constructions.

To decide which hubs to open, we use an evaluation function, described in what follows. Suppose that some nodes have already been selected as hubs. We denote them by H'. In order to select the next hub, the following four elements are considered:

- **Traffic value**. For each node i, we compute the traffic through it (incoming, outgoing, and internal traffic) with origin or destination not in H' as $t(i) = \sum_{j \in V \setminus \{i\}} t_{ij} + \sum_{j \in V \setminus \{i\}} t_{ji} + t_{ii} - \sum_{j \setminus H'} t_{ij} - \sum_{j \setminus H'} t_{ji}$. We then compute the relative value as $\frac{t(i)}{\max_{j \in V \setminus H'} t(j)}$.
- **Opening cost**. We compute the relative fixed installation cost for each node i as $\frac{C_{ii}}{\max_{j \in V \setminus H'} C_{jj}}$.
- **Assignment cost**. For each node i we compute the $m = n/p$ nodes (i.e. the average number of terminals assigned to a hub) with lowest assignment cost to i. The absolute assignment cost of node i, $a(i)$, is defined as the sum of these m costs. The relative assignment cost is computed as $\frac{a(i)}{\max_{j \in V \setminus H'} a(j)}$.
- **Frequency value**. For each node i, we record in $\text{freq}(i)$ the number of times (previous solutions) in which i has been a hub. The relative frequency is $\frac{\text{freq}(i)}{\max_{j \in V \setminus H'} \text{freq}(j)}$.

These four elements are merged into a single expression reflecting the attractiveness of a node to be selected as a hub. Since we cannot establish a priori their relative importance, we introduce some factors that will be empirically set. For each node i, $\text{attractive}(i)$ is defined as:

$$\text{attractive}(i) = +\delta \frac{t(i)}{\max_{j \in V \setminus H'} t(j)} - \alpha \frac{C_{ii}}{\max_{j \in V \setminus H'} C_{jj}} - \beta \frac{a(i)}{\max_{j \in V \setminus H'} a(j)} - \gamma \frac{\text{freq}(i)}{\max_{j \in V \setminus H'} \text{freq}(j)},$$

where $\alpha, \beta, \gamma, \delta \in [0,1]$, and $\alpha + \beta + \gamma + \delta = 1$.

Typically, a node with large traffics is attractive to be selected as a hub, whereas a high opening cost or large assignment costs discourages it. Regarding the frequencies, since we want to diversify the search, those nodes that have been selected as hubs in previous solutions are penalized in posterior constructions. In our computational experiments section, we will test different values of these parameters, which permits to isolate the effect of each of the three elements considered (apart from the frequency) and evaluate their contribution.

At each iteration, our multi-start constructive method selects the best hubs according to the attractive values. Once the hubs are selected, we proceed to assign the terminals to these hubs. Let H be the set of selected hubs. For each terminal i, we compute the best assignment cost $\bar{c}_i$ to the selected hubs as:

$$\bar{c}_i = min_{h \in H} C_{ih} .$$

Then, we order all the terminals according to the $\bar{c}_i$-values, where the terminal with the minimum value comes first. Following this order, we assign each terminal to its best hub (the one in H with the minimum cost) if it has enough capacity. If this hub does not have enough capacity to route the traffics of terminal i because of previous assignments, we consider the second best hub in H for i according to the assignment cost. We proceed in this way, trying to assign terminal i to the best hub in H that can manage its traffic. It may happen that none of the hubs in H can accommodate a given terminal. In this case we add this terminal to a list of *orphan nodes.* When the assignment procedure has explored all terminals, the orphan node with largest traffics is selected as a new hub and we assign as many orphan nodes as possible to it. This procedure, which searches among orphan nodes to select a new hub, is repeated until all nodes are assigned to a hub.

The following example illustrates the constructive method described above, that we call CM1. Suppose that we have a network with 15 nodes where we have estimated $p = 2$. We have applied $\text{attractive}(i)$ to all nodes and concluded that $H = \{3, 10\}$. Column "Order" of Table 1 shows the order of the terminals in V for assigning terminals to hubs according to the $\bar{c}_i$-values. The assignment process starts by terminal 4 and assigns it to hub 3, since it is the preferred hub for this terminal. The procedure continues by assigning terminal 1 to hub 10; terminal 7 to hub 3; terminals 2, 15, and 14 to hub 10; and terminal 9 to hub 3. By the time terminal 13 needs to be assigned, hub 10 is full of traffics, so terminal 13 is assigned to hub 3 (second best hub for terminal 13). Then, terminals 8, 11, and 5 are assigned to hub 3. At this point, hubs 3 and 10 are completely full, hence they cannot accommodate any other terminal. Node 6 (next terminal in the order) is then declared an orphan node. The same happens with node 12. Since no more nodes remain to be assigned, the orphan node with largest traffics is selected as a new hub. Suppose that this is the case of node 12. Therefore $H \leftarrow H \cup \{12\}$ and node 12 is assigned to itself. As there is still enough capacity in the new hub 12, terminal 6 is assigned to it. The assignment process finishes here because all terminals have been assigned to a hub.

Notice that the process implemented in CM1 is designed under the assumption that the best assignment for a terminal is the hub for which its assignment cost is the lowest. However, considering that this is a greedy process and that hubs are limited in terms of their capacity, it is clear that in many cases we cannot assign all the terminals to their preferred hub. For this reason, we also propose alternative construction methods to implement different search strategies. Construction method CM2 orders the terminals in non-increasing order of the $\bar{c}_i$-values (i.e., the terminal with the largest value comes first).

***Table 1**. Example of ordered nodes and possible hubs in CM1.*

Order	Terminal	Ordered hubs	
1st	4	3	10
2nd	1	10	3
3rd	7	3	10
4th	2	10	3
5th	15	10	3
6th	14	10	3
7th	9	3	10
8th	13	10	3
9th	8	3	10
10th	11	3	10
11th	5	3	10
12th	6	10	3
13th	12	10	3

The rationale behind this rule is to assign first the terminal with highest minimum assignment cost to the hubs. Constructive method CM3 computes the lowest and the second lowest assignment cost for each terminal i:

$$\bar{c}_i = min_{h\in H}\, C_{ih}\,,\ \ \bar{\bar{c}}_i = min_{h\in H\setminus\{h^*\}}\, C_{ih}\,,\ \text{where } h^* = \arg min_{h\in H}\,.$$

Then CM3 calculates the difference of the two values above, $d_i = \bar{\bar{c}}_i - \bar{c}_i$, to evaluate how "urgent" is to assign i to its best hub. It is clear that if d_i is large, we should try to assign i to its best hub because otherwise the assignment cost will be greater. On the contrary, if d_i is low, the assignment costs of i to its best and second best hubs are very similar. CM3 orders the terminals according to the d_i-values in non-increasing order and assigns them to their best available hub in this order.

Once a solution S is obtained with one of the three methods above, we evaluate it. To this end, different costs are involved:

- The fixed cost of opening/installing hubs.
- The fixed cost of assigning each terminal to its associated hub.
- The cost of installing the copies of the backbone edges needed to send the traffics between hubs.

As it is customary in multi-start methods, once a solution is constructed, we proceed to improve it. The improvement methods used are described in the next section.

3. Improvement Methods

Corberán et al. (2016) consider two neighborhoods, *Npairs* and *Nalone*, to improve a solution by changing the assignments of terminals to hubs. *Npairs* implements a classical exchange in which two terminals i and j, assigned to hubs k and l respectively ($k \neq l$), swap their corresponding hubs (i.e., the move assigns i to hub l, and j to hub k). The authors proposed a local search method, *LSpairs*, which implements this neighborhood with a first improvement strategy, i.e. by scanning the list of terminals and applying this exchange every time terminals are assigned to different hubs and the objective function is reduced (while the capacity limits are satisfied). Note that an efficient computation of the objective value after a move requires a detailed study. We refer the reader to Corberán et al. (2016) for such a study.

As the authors mentioned, the *Npairs* neighborhood turns out to be too restrictive due to the capacity constraints, so they complemented it with the *Nalone* neighborhood. This second neighborhood performs a simple insertion move in which the assignment of a terminal is changed from a hub to another hub. As in the previous neighborhood, authors proposed a local search method, *LSalone*, based on this move, which implements a first improvement strategy.

The experimental testing in Corberán et al. (2016) shows that the combination of the two neighborhoods is able to significantly improve the constructed solutions. Here, we want to go a step further by including the possibility of changing the hub selection of a given solution in the neighborhood exploration. As a matter of fact, we believe that further reductions in the objective function can be achieved by permitting the local search method to test different sets of hubs. However, including or removing a hub in a solution may cause a great change in S, and consequently the evaluation of such a move can be very costly, especially the computation of the number of backbone edges needed after any move. To overcome this difficulty, we propose a new neighborhood, *Ncluster*, in which we consider that each hub, together with its assigned terminals, form a set (or *cluster*) in the network. This neighborhood explores the change of hub within each cluster. In this way, the hub in a cluster changes its status to become a terminal and one of the terminals in this cluster is now the new hub of the cluster. The rest of the terminals in the cluster remain the same but assigned now to the new hub.

The proposed neighborhood *Ncluster* exhibits a tradeoff between search power and computational cost. On the one hand, it considers changing the hub in a solution, which is a major change that may lead to different types of solutions in the solutions' space. On the other, as it limits the exploration to changes within a cluster, there is no need to calculate the number of copies of the backbone edges to compute the value of the new solution.

The associated procedure, *LScluster*, works as follows. Let U_h be a cluster of nodes formed by a hub h and the set of its assigned terminals S, $U_h = \{h\} \cup S$. We define an evaluation function of the cluster of h, based on the opening cost of h and on the assignment cost of its terminals, as follows:

$$\text{eval}(U_h) = C_{hh} + \sum_{j \in S} C_{jh} \,.$$

This evaluation function induces an order in which the set of clusters will be explored in *LScluster*, where the cluster with the largest evaluation (the one with highest cost) is explored first, since we try to improve it in the first place. Steps 5 and 6 in Algorithm 1 show respectively the evaluation and ordering of the clusters.

Input: (s)
1 continue ← $TRUE$;
2 **while** continue is $TRUE$ **do**
3 continue ← $FALSE$;
4 Define $\mathcal{U} = \{U_{h \in H}\}$
5 Compute clusters' evaluation $\text{eval}(U_h) = C_{hh} + \sum_{j \in U_h} C_{jh}$, $\forall h \in H: |U_h| \geq 2$
6 Order $h \in H: |U_h| \geq 2$ by non-decreasing value of $\text{eval}(U_h)$
7 **foreach** $h \in H: |U_h| \geq 2$ **do**
8 Compute nodes' evaluation $\text{eval}(i) = C_{ii} + \sum_{j \in S \setminus \{i\}} C_{ji}$, $\forall i \in S$
9 Order $i \in S$ by non-decreasing value of $\text{eval}(i)$
10 **foreach** $i \in S$ **do**
11 $\bar{H} = H \setminus \{h\} \cup i$
12 $\bar{S} = S \setminus \{i\} \cup h$
13 $U_i = \{i\} \cup \bar{S}$
14 $\bar{\mathcal{U}} = \{U_{j \in \bar{H}}\}$
15 Compute cost solution value using $\bar{\mathcal{U}}$
16 **if** cost of solution using $\bar{\mathcal{U}}$ < cost of solution using $\mathcal{U}$ **then**
17 $H \leftarrow \bar{H}$
18 $i \leftrightarrow h$
19 continue ← $TRUE$
20 **end**
21 **end**
22 **end**
23 **end-while**
Output: s

Algorithm 1. Procedure *LScluster*

Once a cluster is selected in the main loop of Algorithm 1 (steps 7 to 22), we explore its terminals for a possible swapping with the current hub in the order given by their evaluation. Given an element i in cluster $U_h = \{h\} \cup S$, its evaluation is given by:

$$\text{eval}(i) = C_{ii} + \sum_{j \in S \setminus \{i\}} C_{ji}\,.$$

We explore the nodes in the cluster and perform the first improvement move. After we have scanned all the clusters and eventually performed one or more moves, *LSalone* and *LSpairs* are applied. It is clear that if the hub of a cluster changes, some of the nodes in another cluster could be assigned to the new hub. As mentioned, *Ncluster* does not check this point. Therefore, we consider the *Nalone and Npairs* neighborhoods to check these re-assignments and eventually perform further changes after a hub move. The application of *LSalone* and *LSpairs* may change a clusters' composition. We perform further steps in which we first go over the clusters with the *Ncluster* neighborhood exploration, and then apply the *LSalone* and *LSpairs*, as long as the solution improves. Our local search ends when no further improvement is possible.

4. Path Relinking

Path relinking (PR) was suggested as an approach to integrate intensification and diversification strategies in the context of tabu search (Glover and Laguna, 1997). This approach generates new solutions by exploring trajectories that connect high-quality solutions by starting from one of these solutions, called *initiating solution*, and generating a path in the neighborhood space that leads toward the other solutions, called *guiding solutions*. This is accomplished by introducing in the initiating solutions attributes contained in the guiding ones. In this way, we generate a sequence of intermediate solutions that "connect" the initiating solution with the guiding one.

The term relinking reflects the fact that this method links again two or more solutions with a path in the search space. In the original tabu search design, two or more good solutions are recorded during the search. Since tabu search describes a trajectory, these solutions can be viewed as linked in the search space by the chain of moves which originated them. After the tabu search execution, we can consider to create a new trajectory, or chain of moves, to go from one of these high-quality solutions to another one. This new trajectory is directed considering the target solutions, instead of the objective function as in the initial application of tabu search. The method is therefore called path relinking because it creates two paths joining two solutions.

Laguna and Martí (1999) adapted PR in the context of GRASP as a form of intensification. The relinking, in the context of multi-start algorithms, consists of finding a path between two solutions generated with the constructive method and, eventually, improve the solution in the path with a local search. Therefore, the relinking concept has a different interpretation within GRASP, since the solutions are not originally linked by a sequence of moves. The authors, however, kept the original name of the methodology in spite of the fact that the two solutions are linked for the first time. Resende et al. (2010) explored different implementations to hybridize these two methodologies:

- *Greedy Path Relinking*. In this method the moves in the path from a solution to another are selected in a greedy fashion, according to the objective function value.
- *Greedy Randomized Path Relinking*. Here the method creates a candidate list with the good intermediate solutions and randomly selects among them.
- *Truncated Path Relinking*. In this application of PR the path between two solutions is not completed. It is applied, for example, in problems where good solutions are found close to the end points (original solutions) in the path.
- *Evolutionary Path Relinking*. This method iterates over the set of high-quality solutions, applying successively the relinking mechanism. It has many similarities with the scatter search methodology (Laguna and Martí, 2003).

In this paper we explore a kind of *Greedy Truncated Path Relinking* to our problem as a post-process method. Let X and Y be two solutions to the CSHLPMLC, and let H_X and H_Y be their associated sets of hubs. The path relinking procedure $\mathrm{PR}(X, Y)$ starts with X, and gradually transforms it into Y, by swapping out hubs in X with hubs in Y. The hubs in both solutions, H_{XY}, will remain as hubs in all the intermediate solutions generated in the path between them. Let H_{X-Y} be the hubs in X that are not hubs in Y. H_{Y-X} is defined equivalently. Let $\mathrm{PR}_0(X, Y) = X$

be the initiating solution in the path from X to Y. To obtain the first solution $\mathrm{PR}_1(X,Y)$ in this path, we remove a hub $i \in H_{X-Y}$ and replace it with a hub $j \in H_{Y-X}$, thus obtaining

$$H_{\mathrm{PR}_1(X,Y)} = H_{\mathrm{PR}_0(X,Y)} \setminus \{i\} \cup \{j\}.$$

In the PR variant implemented here, the selection of the nodes i, j is the one minimizing the objective function. In general, to obtain $\mathrm{PR}_{t+1}(X,Y)$ from $\mathrm{PR}_t(X,Y)$, we evaluate the different hubs $i \in H_{\mathrm{PR}_t(X,Y)-Y}$ to be removed and the hubs $j \in H_{\mathrm{Y}-\mathrm{PR}_t(X,Y)}$ to be selected. The move associated with the minimum cost option is performed.

At each intermediate solution in the path from X to Y, a restricted neighborhood is explored to generate the next solution in the path. The neighborhood is restricted because only moves removing hub $i \in H_{\mathrm{PR}_t(X,Y)-Y}$ and selecting hub $j \in H_{\mathrm{Y}-\mathrm{PR}_t(X,Y)}$ are allowed. As the procedure moves from one intermediate solution to the next, the cardinalities of sets $H_{\mathrm{PR}_t(X,Y)-Y}$ and $H_{\mathrm{Y}-\mathrm{PR}_t(X,Y)}$ decrease by one element. Consequently, as the procedure nears the guiding solution, there are fewer allowed moves to explore. This is why Resende et al. (2010) suggested that the search tends to be less effective in the final stages. In truncated path relinking, a new stopping criterion is used. Instead of continuing the search until the guiding solution is reached, only a limited number of steps, $\mathrm{PR}_{\mathrm{steps}}$, are allowed, abandoning the path before reaching the final solution. We consider $\mathrm{PR}_{\mathrm{steps}}$ as a search parameter, and explore the performance of the method with different values in the next section.

In our algorithm we first apply a constructive method (either *CM1*, *CM2* or *CM3*) and the local search methods (*Ncluster, Nalone,* and *Npairs*) to populate a set with high-quality solutions (elite set, ES). Our PR mechanism is designed to work as a post-process method. For each pair of solutions in ES, the cardinality of hubs and the objective values are compared in order to decide which of them will be the starting and the guiding solution. In the case where the cardinalities of their corresponding set of hubs are different, the solution with the highest number of hubs is chosen as starting solution X, and the path relinking is performed against the guiding solution Y. When all hubs in H_{Y-X} have been incorporated, there are one or more hubs that should be removed in order to finally reach Y. The procedure does not do this and stops at this step, so the whole path is not examined completely. This is the reason to consider our procedure a *Truncated* PR. If H_X and H_Y have the same cardinality, the solution with poorest objective value is chosen as the starting point and the path relinking is performed towards the better solution. The assignment of terminals to the hubs of any intermediate solution explored during the path relinking process is performed using the same strategy of the construction process.

5. Computational Experiments

This section describes the computational experiments that we performed to test the effectiveness and efficiency of the procedures discussed above. The algorithms have been implemented in C and run on an Intel Xeon E3-1270 at 3.40 GHz and 16GB of RAM computer running Windows 7–64 bits.

The metrics that we use to measure the performance of the algorithms are:

- Value: Average objective value of the best solutions obtained with the algorithm on the instances considered in the experiment.
- Dev: Average percentage deviation from the best-known solution.
- Best: 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.

From previous publications, we have identified a benchmark with 170 instances from three well-known data sets:

- The CAB (Civil Aviation Board) data set, based on airline passenger flows between some important cities in the United States. It consists of a data file, presented by O'Kelly in 1987, with the distances and flows of a 25-nodes network. From this original file, a total of 23 instances with 10, 15, 20 and 25 nodes have been used.
- The AP (Australian Post) data set, based on real data from the Australian postal service and presented by Ernst and Krishnamoorthy in 1996. The size of the original data file is 200 nodes. Smaller instances can be obtained using a code from ORLIB (Beasley, 1990). We have used 70 instances with n ranging from 10 to 200. Regarding the flows between nodes, these instances do not have symmetric flows (i.e., for a given pair of nodes i and j, t_{ij} is not necessarily equal to t_{ji}). Moreover, some flows from one node to itself are positive (i.e., $t_{ii} > 0$ for a given i).
- The USA423 data set was introduced in Peiró et al. (2014), and is based on real airline data. It consists of a data file concerning 423 cities in the United States, where real distances and passenger flows for an accumulated 3 months period are considered. From the original benchmark, we have employed 77 instances in the experimentation below with n ranging from 20 to 250.

Each instance includes the three matrices (t_{ij}), (C_{ij}), (R_{kl}) and the two capacity values Q^b and Q^h. The entire set of instances is available at www.optsicom.es.

The experimental part is divided into two main blocks. The first block (scientific testing) is devoted to study the performance of the components of the algorithm, as well as to determine the best values for the key search parameters. They have been performed on the same training set of 36 instances used in Corberán et al. (2016). The second block of experiments (competitive testing) has the goal of comparing our procedure with the best published methods.

5.1 Scientific testing

From the 170 instances derived from the CAB, AP and USA423 data sets, the preliminary experiments are performed on the following set of 36 instances: 3 instances with $15 \leq n \leq 25$ from the CAB set, 21 instances with $10 \leq n \leq 195$ from the AP set, and 12 instances with $20 \leq n \leq 150$ from the USA423 set. These instances have been classified as small, medium, and large, with 12 instances in each group.

In our first experiment we have compared the combination of the different elements of the $\text{attractive}(i)$ function for the hub selection in the constructive method (see Section 2). Specifically, we have considered the following five alternative sets of parameter values:

- Alt1 = { α =0.25; β =0.25; γ =0.25; δ =0.25}
- AllAlpha = { α =1; β =0; γ =0; δ =0}, only considers the opening costs
- AllBeta = { α =0; β =1; γ =0; δ =0}, only considers the assignment costs
- AllGamma = { α =0; β =0; γ =1; δ =0}, only considers the frequency values
- AllDelta = { α =0; β =0; γ =0; δ =1}, only considers the traffics

Note that Alt1 is the only alternative among the above ones including the four elements in $\text{attractive}(i)$. The other alternatives isolate the effect of each element, so they measure their contribution to the complete evaluation.

Table 2. Average percentage deviation from the best solution (Dev)

Size	# instances	Alt1	AllAlpha	AllBeta	AllGamma	AllDelta
small	12	6.6%	30.7%	35.4%	31.4%	17.3%
medium	12	0.4%	17.9%	10.5%	16.1%	9.1%
large	12	2.8%	50.6%	29.2%	27.0%	5.9%
summary	36	3.3%	33.1%	25.1%	24.9%	10.8%

Table 2 shows the average percentage deviation from the best solution obtained with each alternative method when constructing 100 solutions on each instance. As expected, the best alternative is Alt1, where the four elements are combined. AllDelta (the alternative with $\delta = 1$) is the best among the remaining alternatives, which seems to indicate that the traffic through a node is a very important factor to choose it as a hub. Both alternatives, Alt1 and AllDelta, produce better values for Dev than AllBeta, which was the one used in Corberán et al. (2016), based on the assignment costs of terminals to hubs. The Friedman statistical test for multiple paired samples obtains a p-value lower than 0.0001, which confirms that there are differences among the five alternatives tested. Additionally, the ranks obtained with this test are Alt1=1.49, AllAlpha=3.61, AllBeta=3.76, AllGamma=3.64, and AllDelta=2.50, which are in line with the above results.

Figure 1 shows the average quality of the resulting solutions as a relative value of the best-known solutions found in Corberán et al. (2016). We include in this diagram the results obtained with a variant where hubs are randomly selected, called AltRand.

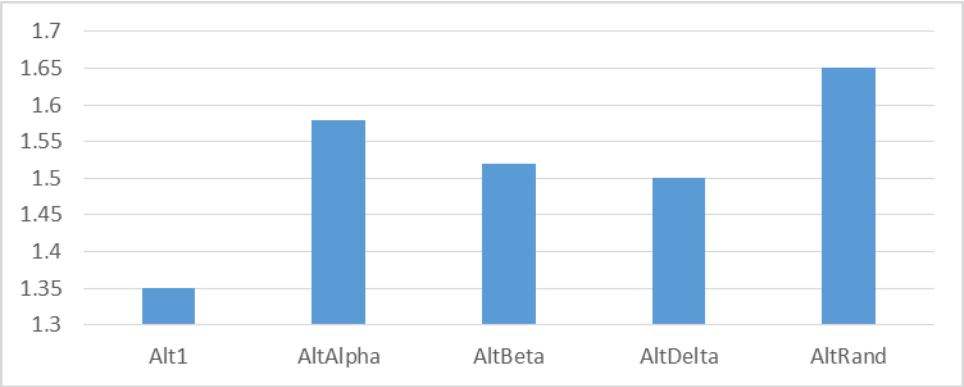

Figure 1. Average quality of solutions constructed with different strategies for hub selection

Figure 1 confirms that the single element giving the best results is the traffic value. However, in this experiment we have also evaluated the variation on quality among the elements when looking at the single instances. For some instances, the traffic value is clearly the most favorable element. For others however, the opening costs or the assignment costs give better results, while the other elements' results are poorer.

In the second experiment we extend the previous analysis by including other alternative sets of parameters in the constructive method. The new combinations are:

- Alt2 = { α =0.70; β =0.10; γ =0.10; δ =0.10}
- Alt3 = { α =0.10; β =0.70; γ =0.10; δ =0.10}
- Alt4 = { α =0.10; β =0.10; γ =0.70; δ =0.10}
- Alt5 = { α =0.10; β =0.10; γ =0.10; δ =0.70}
- Alt6 = { α =0.35; β =0.20; γ =0.10; δ =0.35}
- Alt7 = { α =0.40; β =0.10; γ =0.10; δ =0.40}
- Alt8 = { α =0.40; β =0.15; γ =0.05; δ =0.40}

The results obtained using these alternatives are then compared against Alt1, where the four parameters have the same weight. In Alt2, Alt3, Alt4, and Alt5, one of the parameters receives a larger weight value (0.70), while the other coefficients get a smaller weight (0.10). For Alt2 the highest weight is given to α, which is associated with the fixed cost of opening a hub. Similarly, Alt3 gives more weight to the cost of assigning terminals to hubs, while Alt4 penalizes the frequency of occurrence in previous iterations, and Alt5 gives more weight to the amount of traffic through the node. The Alt6, Alt7, and Alt8 alternatives try some other combinations of weights, where α and δ have higher values than the other parameters. This is based on the initial findings indicating that the opening costs and the traffics are the two most important elements when selecting hubs. Table 3 shows the average percentage deviation from the best solution found in this experiment after constructing 100 solutions on each instance.

Results in Table 3 clearly show that alternative Alt2 builds solutions with least deviation, followed by alternative Alt6, which performs especially well on large instances. We have performed the Friedman test with these alternatives and obtained a p-value < 0.0001, which confirms that there are significant differences between the alternatives. The test returned the

following ranges in line with the deviations in Table 3: Alt1 = 3.90, Alt2 = 2.54, Alt3 = 4.26, Alt4 = 7.68, Alt5 = 4.53, Alt6 = 3.97, Alt7 = 4.29, and Alt8 = 4.82.

Table 3. Average percentage deviation from best solution (Dev)

Size	# instances	Alt1	Alt2	Alt3	Alt4	Alt5	Alt6	Alt7	Alt8
small	12	11.3%	7.3%	14.0%	36.7%	10.6%	5.8%	4.8%	4.9%
medium	12	8.3%	1.1%	8.1%	19.8%	9.3%	7.5%	8.8%	8.4%
large	12	4.9%	5.4%	7.3%	24.6%	6.3%	4.6%	9.3%	10.7%
summary	36	8.2%	4.6%	9.8%	27.0%	8.7%	6.0%	7.6%	8.0%

To complement the analysis above, we represent in Figure 2 the boxplots of the percentage deviations of each alternative. These diagrams represent the 36 deviation values obtained with each alternative on the instances of the training set. It can be seen that Alt2 has the highest concentration of lower relative deviations, which means that it produces the best solutions.

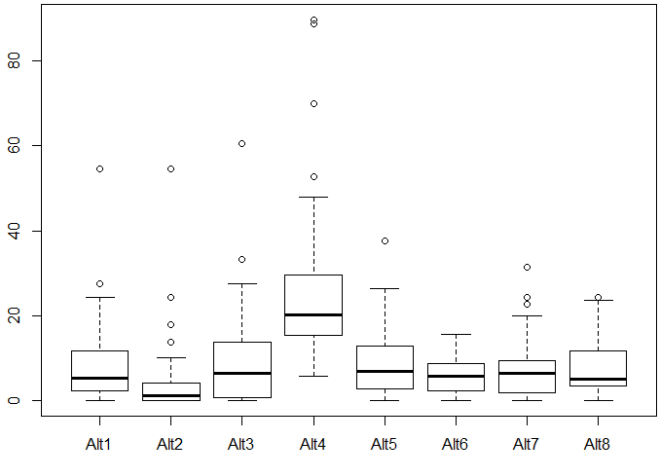

Figure 2. Box plot of different alternatives

After analyzing the methods for selecting hubs, we study the methods of assigning terminals to hubs (CM1, CM2, and CM3). Table 4 shows the results of this experiment on the training set. For each construction method, the combinations of parameters that have been found to be best in the previous experiment have been tested. In particular, Alt1, Alt2, Alt6, Alt7, and Alt8 are tested.

Table 4 show that the lowest average percentage deviation (8%) is obtained with CM3 and parameter combination Alt2. On the other hand, CM2 with Alt2 or Alt6 are close to it with a 9% deviation, while the CM1 strategy gives much poorer results on all parameter combinations. A non-parametric Wilcoxon test performed to compare CM2 and CM3 both using Alt2, gave a p-value of 0.789, which indicates that there is no significant difference between these two

configurations. We have chosen CM3 with Alt2 as the configuration for our constructive method and will use it in the following experiments.

Table 4. Average percentage deviation for constructive methods

	CM1					***CM2***					***CM3***				
	Alt1	***Alt2***	***Alt6***	***Alt7***	***Alt8***	***Alt1***	***Alt2***	***Alt6***	***Alt7***	***Alt8***	***Alt1***	***Alt2***	***Alt6***	***Alt7***	***Alt8***
small	15%	13%	14%	14%	11%	17%	16%	8%	11%	9%	15%	11%	9%	8%	8%
medium	23%	20%	22%	23%	23%	12%	6%	11%	12%	11%	13%	5%	12%	13%	13%
large	59%	39%	40%	44%	41%	10%	6%	10%	11%	14%	8%	9%	8%	13%	14%
summary	32%	24%	25%	27%	25%	13%	9%	9%	11%	12%	12%	8%	10%	11%	12%

In the fourth experiment, we study the contribution of the local search phase of the algorithm applied to the solutions obtained with the constructive method CM3 (and Alt2). In particular, we study five local search variants, as described in Section 3:

- *LSalone* - The method only applies the local search *LSalone* (Corberán et al. 2016).
- *LSpairs* – The method only applies the local search *LSpairs* (Corberán et al. 2016).
- *LSalone + LSpairs*– The method combines the two previous local searches.
- *LScluster* - The method only applies the new local search *LScluster*.
- *LSAllOnce* - The three methods, *LScluster*, *LSalone, and LSpairs* are applied once in this order.

It seems natural to ask whether we would be able to improve the solution further after performing the three local searches defining *LSAllOnce*. Since *LSpairs* and *LSalone* can change the clusters, it might happen that another terminal could be a better hub in the modified clusters. Similarly, when a hub is changed, it could be a better option to assign some terminals to a different hub. We therefore consider a sixth variant, called *LSAll*, in which these methods are repeatedly applied in a loop as shown in Algorithm 2.

Input: (s)
1 continue ← $TRUE$;
2 **while** continue is $TRUE$ **do**
3 continue ← $FALSE$;
4 *LScluster*
5 *LSpairs*
6 **if** solution improved after *LSpairs* **then**
7 continue ← $TRUE$;
8 **end**
9 *LSalone*
10 **if** solution improved after *LSalone* **then**
11 continue ← $TRUE$;
12 **end**
13 **end-while**
Output: s

Algorithm 2. Procedure *LSAll*

Table 5 reports the average percentage reduction from the constructed solution value obtained with these six local search variants by running 50 iterations (construction + local search) on the instances in the training set. It shows that, on average, *LSalone* improves the solution value by

4.7% with respect to the solutions obtained with the constructive method, while *LSpairs* improves it by a 5.0% on average. In line with the results in Corberán et al. (2016), Table 5 shows that a further reduction is achieved if both local searches are combined. In particular, *LSalone + Lspairs* exhibit an improvement of 8.2%. Considering these three previous methods as a basis for comparison, we can observe from this table the good performance of the new local search, *LScluster*, which obtains better results. Specifically, applying *LScluster* only, improves by 11.5% on average the constructed solutions. Another advantage of this search is that it is very fast, since the amount of traffic between the clusters is constant and the number of edges used in the backbone network (W_{kl}) do not change. The *LSAllOnce* column shows the result when applying *LScluster* first, then *LSalone* and then *LSpairs*. This process improves the solutions obtained considerably, reaching a reduction of 24% on average on the large size instances, and of 17.8% for all instances in the training set. Finally, The *LSAll* column shows the average results of the improvements after applying the loop procedure described in Algorithm 2. Clearly, it shows that changing the configuration of clusters gives further improvements, thus concluding that the *LSAll* is the best of the six alternatives. We have applied the Wilcoxon statistical test for two paired samples to *LSAllOnce* and *LSAll*, which returns a p-value lower than 0.0001, confirming the superiority of *LSAll*.

Table 5. Local search percentage reduction from construction

Size	LSalone	LSpairs	LSalone+ LSpairs	LScluster	LSAllOnce	LSAll
small	-2%	-2%	-5%	-9%	-13%	-18%
medium	-5%	-5%	-8%	-12%	-17%	-22%
large	-7%	-8%	-12%	-14%	-24%	-29%
summary	-4.7%	-5.0%	-8.2%	-11.5%	-17.8%	-22.8%

Figure 3 shows the search profile of the six methods described above. Specifically, we represent the objective value of the best known solution on a 130-nodes instance when applying different solution methods for 100 global iterations. We have added a seventh method, labeled "Constructions", which represents the best value obtained when applying the constructive method without any local search. The other six lines represent the best solution value obtained with the application of the construction method followed by each of the six local searches described above. Note that in addition to the LSalone+LSpairs combination previously considered, we have also included an eight method, LSpairs+LSalone, to study the application of these two methods in reverse order. The results in this diagram agree with those reported in Table 5, where the two methods *LSAllOnce* and *LSAll* present the best results, being *LSAll* the best one. It must be noted that *LSAll* is able to obtain the best results from the very beginning of the search process, and continues being the leader in the entire search horizon considered.

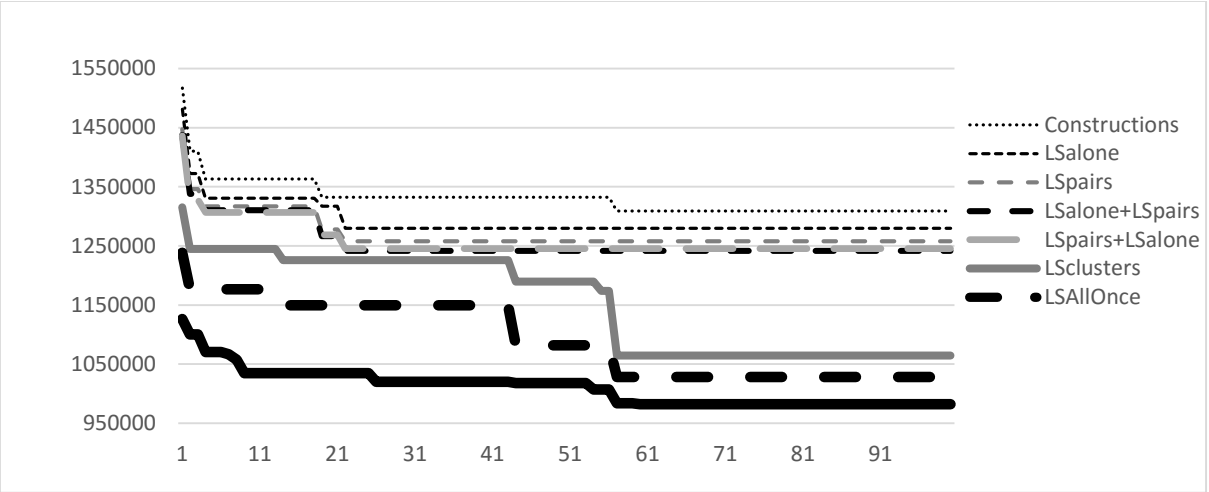

Figure 3. Search profile for the different local search methods

In the next experiment, we undertake to assess the performance of the PR post-process. To do this, all the instances in the training set have been executed for 20 iterations to create the solutions in the ES that will be used during PR. The path relinking procedure has been able to improve the previously obtained results in 22 out of the 36 instances in the training set, with an average improvement of 1.4% (denoted as -1.4% in the Summary row of Table 6). In this experiment we have also tested the strategy called two-sided path relinking (Festa and Resende 2011), in which the best direction to create the path is determined. In particular, we have tested the reversal of the direction considered so far in the PR process, moving from the best towards the poorer solution, and it has not produced any significant difference in the results. In fact, in 26 of the instances, PR gives the same best solution in the path, independently of the direction. Table 6 shows the results for the different groups of instances, together with the increase in the CPU time for including PR. It can be seen that the increase in the computing time decreases with the size of the instances and, in our opinion, the extra time consumed by PR is worth, especially in the large size instances, where a 2.3% of improvement is obtained.

Table 6. Path relinking contribution

Size	Deviation from best	CPU
small	-1.3%	+33.3%
medium	-0.7%	+16.4%
large	-2.3%	+13.9%
Summary	-1.4%	+21.2%

In our final experiment in the scientific testing, we explore the variant known as Truncated Path Relinking (Resende et al. 2010). As described in Section 4, this strategy considers the early termination of the PR without reaching the guiding solution, thus performing only a limited number of steps, PR_{steps}. Table 7 shows the average percentage improvement (reduction with respect to the initiating solution) achieved at each step of the path. As in previous tables, we divide the results according to the size of the instances and summarize them in an additional

row. We include a final row with the average increase in the CPU time due to the application of PR.

Table 7. Average percentage reduction in truncating path relinking

	PR_{steps}					
Size	**1**	**2**	**3**	**4**	**5**	**6**
small	-1.2%	-1.3%	-1.3%	-1.3%	-1.3%	-1.3%
medium	0.0%	-0.3%	-0.4%	-0.6%	-0.7%	-0.7%
large	-0.9%	-1.3%	-1.9%	-2.0%	-2.3%	-2.3%
Summary	-0.7%	-1.0%	-1.2%	-1.3%	-1.4%	-1.4%
CPU time	+2.9%	+5.4%	+7.0%	+8.1%	+10.0%	+10.6%

In line with Table 6, Table 7 clearly shows the overall contribution of PR. Moreover, this table shows that we achieve in two steps the best possible result in the small instances and, therefore, it is a good strategy to truncate the path at an early stage. Medium and large instances require more steps to achieve the best results in the path. At some point (step 5) the improvement stagnates and further steps do not produce better results.

5.2 Competitive testing

Once we have established the key-search parameters and explored the different variants of our method, we compare two variants of it with the best previous method. The two variants, AMP_10 and AMP_20, correspond to a termination criteria of 10 and 20 global iterations with full path relinking between the solutions in the Elite Set.

Table 8 reports the results of the comparison between the proposed procedure and the Strategic Oscillation method, SO, by Corberán et al. (2016). We consider the full set of 170 instances, and report the average percentage deviation (Dev), the running time in seconds (CPU), and the number of best solutions found with each method (#Best). These instances have been classified as small $(10 \leq n \leq 50)$, medium $(55 \leq n \leq 100)$, large $(110 \leq n \leq 150)$, extra-large $(155 \leq n \leq 200)$, and huge $(205 \leq n \leq 250)$. Detailed results on each instance are shown in Tables 9-13 in the Appendix.

Table 8. Comparison with best previous method

		Dev			***CPU***			***# Best***		
Size	***# inst***	**SO**	**AMP_10**	**AMP_20**	**SO**	**AMP_10**	**AMP_20**	**SO**	**AMP_10**	**AMP_20**
small	45	1.8%	0.9%	0.4%	3.0	0.3	0.8	13	23	33
medium	36	3.2%	1.1%	0.3%	37.8	11.2	24.8	6	8	30
large	27	10.6%	0.8%	0.0%	310.4	88.9	194.3	0	12	27
ex-large	37	13.2%	1.2%	0.0%	1606.3	185.8	397.8	0	17	37
huge	25	11.6%	1.2%	0.0%	4916.3	679.0	1403.7	0	12	25
Summary	170	7.4%	1.0%	0.2%	1130.7	156.9	329.3	19	72	152

Table 8 clearly shows the superiority of the proposed procedure with respect to the previous SO method. Specifically, AMP_20 is able to obtain 152 best solutions out of the 170 instances

considered, and shows an average computing time of 329.3 seconds. This compares favorably with the 19 best solutions obtained with SO, which uses 1130.7 seconds on average. The average deviation (Dev) of 0.2% of AMP_20 also compares well with the 7.4% of SO. We have performed a Wilcoxon test and the resulting p-value < 0.0001 confirms these conclusions. Regarding the results obtained with AMP_10, it can be seen that, despite being much better than those produced by SO, they are worse than those provided by AMP_20, although only half of the computing time is used.

To complement the information in Table 8, we have performed a final experiment to compare the evolution of the best solution found by AMP_20 and SO. Figure 4 reports the ratio between the best solution found with both methods at each iteration and the best value reported in Corberán et al. (2016), where a search horizon of 100 iterations has been considered. This diagram confirms that our proposal consistently outperforms the best previous published method.

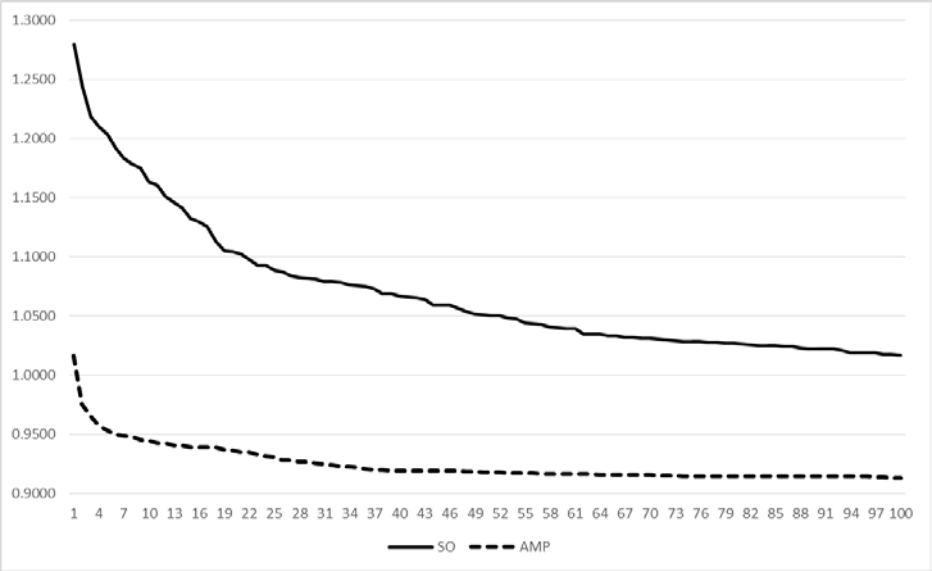

Figure 4. Search profile of best methods

6. Conclusions

Hub location problems are difficult combinatorial optimization problems that have recently received a lot of attention. In this paper, we have studied a capacitated modular hub location problem that has been modeled in the Literature as a mixed integer non-linear program, and for whose solution we have proposed an adaptive memory programming algorithm. We have tested the effects of a variety of search strategies, as those combining different constructive methods and neighborhood structures within a multi-start memory based framework. We have also explored a Path Relinking post-process to obtain improved outcomes. Our experiments show that the adaptive memory features are capable of searching the solution space economically and effectively, outperforming existing approaches.

Acknowledgements

This work was supported by the Spanish Ministerio de Economía y Competividad and Fondo Europeo de Desarrollo Regional (FEDER) (projects TIN-2015-65460-C02-01(MINECO/FEDER), MTM-2015-68097(MINECO/FEDER), PhD. grant BES-2013-064245) and by the Generalitat Valenciana (project Prometeo 2013/049). This support is gratefully acknowledged.

References

Beasley, J. E. (1990) OR-Library: distributing test problems by electronic mail. Journal of the Operational Research Society 41, 1069-1072.

Corberán, Á., Peiró, J., Campos, V., Glover, F., Martí, R. (2016) Strategic oscillation for the capacitated hub location problem with modular links. Journal of Heuristics 22, 221–244.

Ernst, A. T., Krishnamoorthy, M. (1996) Efficient algorithms for the uncapacitated single allocation p-hub median problem. Location Science 4, 139–154.

Festa P., Resende, M.G.C. (2011) GRASP: basic components and enhancements. Telecommunication Systems 46, 253-271.

Glover, F., Laguna, M. (1997) Tabu search, Kluwer, Norwell, MA.

Laguna, M., Martí, R. (1999) GRASP and path relinking for 2-layer straight line crossing minimization. INFORMS Journal on Computing 11, 44–52.

Laguna, M. , Martí, R. (2003) Scatter search: Methodology and implementations in C. Kluwer Academic Publishers, New York.

O'Kelly, M. E. (1987) a quadratic integer program for the location of interacting hub facilities. European Journal of Operational Research 32, 393–404.

Peiró, J., Corberán, Á., Martí, R. (2014) GRASP for the uncapacitated r-allocation p-hub median problem. Computers & Operations Research 43, 50–60.

Resende, M. G. C., Gallego, M. , Duarte, A. , Martí, R. (2010). GRASP and Path Relinking for the Max-Min Diversity Problem. Computers and Operations Research 37, 498–508.

Yaman, H., Carello, G. (2005) Solving the hub location problem with modular link capacities. Computers & Operations Research 32, 3227–3245.

Appendix

Table 9. SO and AMP on small size instances

	SO			AMP_10			AMP_20		
Instance	*Value*	*Dev*	*CPU*	*Value*	*Dev*	*CPU*	*Value*	*Dev*	*CPU*
10_600_89_60_40_1_60_CAB	237701	1.5%	0.22	234243	0.0%	0.00	234243	0.0%	0.03
10_700_50_60_8_1_60_AP	273801	0.0%	0.23	278212	1.6%	0.00	278212	1.6%	0.03
10_700_50_60_8_1_60_CAB	253255	3.9%	0.22	243854	0.0%	0.00	243854	0.0%	0.03
10_700_69_40_8_1_50_CAB	257691	4.5%	0.22	246610	0.0%	0.00	246610	0.0%	0.03
10_800_60_60_6_1_69_AP	245513	0.0%	0.23	263198	7.2%	0.00	263198	7.2%	0.03
10_800_60_60_6_1_69_CAB	244417	2.6%	0.58	238144	0.0%	0.02	238144	0.0%	0.03
10_800_60_80_8_1_80_CAB	247078	2.6%	0.22	240872	0.0%	0.02	240872	0.0%	0.03
15_500_50_60_40_1_60_CAB	289339	1.1%	0.36	286201	0.0%	0.02	286201	0.0%	0.09
15_600_80_89_6_1_69_CAB	293075	0.8%	0.44	290734	0.0%	0.02	290734	0.0%	0.09
15_600_80_89_8_1_60_CAB	308168	2.5%	0.41	300776	0.0%	0.02	300776	0.0%	0.09
15_700_89_60_40_1_60_CAB	289953	0.7%	0.39	288078	0.0%	0.03	288047	0.0%	0.11
15_800_50_60_40_1_60_CAB	290026	1.3%	0.39	286358	0.0%	0.02	286358	0.0%	0.08
15_900_80_89_40_1_60_CAB	290134	1.3%	0.41	286408	0.0%	0.02	286408	0.0%	0.09
20_700_50_60_8_1_60_AP	406266	3.2%	0.98	393788	0.0%	0.05	393788	0.0%	0.20
20_700_50_60_8_1_60_CAB	176142	0.0%	1.03	176783	0.4%	0.03	176783	0.4%	0.16
20_700_50_60_8_1_60_USA	127058	7.7%	0.92	117986	0.0%	0.05	117986	0.0%	0.17
20_700_69_40_8_1_50_AP	408538	0.0%	1.03	417187	2.1%	0.06	417187	2.1%	0.22
20_700_69_40_8_1_50_CAB	187305	2.3%	0.84	183013	0.0%	0.05	183013	0.0%	0.17
20_800_60_60_6_1_69_CAB	164351	0.0%	0.73	165518	0.7%	0.05	165276	0.6%	0.19
20_800_60_80_8_1_80_AP	363247	0.0%	1.19	363247	0.0%	0.03	363247	0.0%	0.22
20_800_60_80_8_1_80_CAB	170069	1.2%	0.75	167993	0.0%	0.03	167993	0.0%	0.19
20_800_60_80_8_1_80_USA	121071	4.9%	0.87	115370	0.0%	0.03	115370	0.0%	0.17
20_900_80_89_40_1_80_CAB	151342	0.4%	0.80	150767	0.0%	0.03	150767	0.0%	0.16
25_600_80_60_6_1_40_CAB	210578	0.0%	1.70	210808	0.1%	0.06	210808	0.1%	0.36
25_600_80_89_6_1_69_CAB	192213	0.0%	1.39	192388	0.1%	0.06	192388	0.1%	0.36
25_600_80_89_8_1_60_CAB	214944	3.9%	1.64	206901	0.0%	0.06	206814	0.0%	0.39
25_650_69_69_6_1_50_CAB	210278	4.0%	1.84	202169	0.0%	0.08	202169	0.0%	0.37
25_650_69_69_6_1_70_CAB	162862	0.7%	2.14	165762	2.5%	0.09	161771	0.0%	0.36
25_800_89_60_40_1_80_CAB	179893	0.0%	1.86	180375	0.3%	0.08	180210	0.2%	0.34
25_900_80_89_40_1_80_CAB	181207	0.7%	1.61	180028	0.0%	0.08	180028	0.0%	0.34
30_600_80_89_8_1_60_AP	383188	8.5%	3.46	353148	0.0%	0.12	353148	0.0%	0.53
30_700_69_40_8_1_50_USA	211640	0.0%	2.93	214726	1.5%	0.25	214726	1.5%	0.83
35_600_80_89_8_1_60_AP	448064	2.5%	3.98	437119	0.0%	0.28	437119	0.0%	0.89
35_600_80_89_8_1_60_USA	206472	0.0%	4.40	207966	0.7%	0.30	207966	0.7%	0.64
35_700_80_50_8_1_69_AP	436550	0.2%	4.23	435489	0.0%	0.27	435489	0.0%	0.58
40_600_80_89_8_1_60_USA	288503	2.7%	6.19	306713	9.1%	1.14	281012	0.0%	2.86
40_700_80_50_8_1_69_AP	478470	0.0%	6.43	488152	2.0%	0.59	486523	1.7%	1.20
40_700_80_50_8_1_69_USA	282629	2.8%	6.01	277734	1.0%	1.03	275037	0.0%	2.07
45_600_80_89_8_1_60_AP	521958	1.5%	8.44	528549	2.7%	0.78	514433	0.0%	1.84
45_700_69_40_8_1_50_AP	589832	0.6%	8.63	592707	1.1%	0.89	586337	0.0%	1.97
45_700_69_40_8_1_50_USA	345335	1.5%	6.83	340367	0.0%	1.42	340367	0.0%	3.29
50_600_80_89_8_1_60_USA	347675	5.0%	12.34	337159	1.8%	2.39	331097	0.0%	4.99
50_700_69_40_8_1_50_AP	598636	0.0%	11.50	610081	1.9%	1.48	610081	1.9%	3.23
50_700_80_50_8_1_69_AP	562786	2.4%	11.23	552563	0.5%	1.42	549699	0.0%	2.85
50_700_80_50_8_1_69_USA	328853	0.5%	10.87	337165	3.0%	2.32	327249	0.0%	5.05

Table 10. SO and AMP on medium size instances

	SO			AMP_10			AMP_20		
Instance	***Value***	***Dev***	***CPU***	***Value***	***Dev***	***CPU***	***Value***	***Dev***	***CPU***
55_500_60_69_60_1_50_AP	551996	4.9%	12.28	526290	0.0%	1.70	526290	0.0%	3.71
55_500_60_69_60_1_50_USA	356419	0.1%	11.51	355920	0.0%	3.28	355920	0.0%	7.39
55_800_69_50_80_1_60_AP	627685	1.4%	13.65	622691	0.6%	1.59	618881	0.0%	3.62
55_800_69_50_80_1_60_USA	356039	0.0%	10.31	367214	3.1%	3.45	367214	3.1%	7.36
60_500_60_69_60_1_50_AP	597551	10.4%	14.23	570095	5.3%	3.31	541254	0.0%	6.75
60_600_60_69_60_1_69_USA	324245	0.0%	14.34	339436	4.7%	4.71	339436	4.7%	9.38
60_800_69_50_80_1_60_AP	664374	0.0%	13.32	671543	1.1%	3.43	669005	0.7%	7.44
60_800_69_50_80_1_60_USA	375981	2.2%	14.96	375481	2.0%	4.56	368020	0.0%	10.25
65_500_60_69_60_1_50_AP	594968	0.5%	19.49	597554	0.9%	3.54	592242	0.0%	8.49
65_600_60_69_60_1_69_AP	596768	4.0%	19.06	574420	0.1%	3.78	573660	0.0%	8.07
65_600_60_69_60_1_69_USA	366133	6.1%	21.09	346628	0.5%	5.90	344968	0.0%	12.45
65_800_69_50_80_1_60_USA	405756	9.6%	21.08	370350	0.0%	7.47	370350	0.0%	14.24
70_500_60_69_60_1_50_AP	664745	4.7%	24.56	634622	0.0%	5.43	634622	0.0%	11.68
70_600_60_69_60_1_69_AP	615656	0.5%	22.25	612709	0.0%	4.60	612709	0.0%	11.17
70_600_60_69_60_1_69_USA	383492	1.8%	29.03	380610	1.0%	7.24	376830	0.0%	17.66
70_800_69_50_80_1_60_AP	769108	2.7%	27.27	754050	0.7%	4.88	748826	0.0%	10.47
75_500_60_69_60_1_50_USA	552080	0.0%	29.14	567428	2.8%	12.40	564299	2.2%	28.97
75_600_60_69_60_1_69_AP	665111	1.9%	35.23	656614	0.6%	6.13	652910	0.0%	13.74
75_600_60_69_60_1_69_USA	519474	3.9%	31.44	516299	3.3%	10.83	499866	0.0%	26.01
75_800_69_50_80_1_60_AP	828245	6.8%	32.87	784188	1.1%	6.65	775560	0.0%	14.40
80_500_60_69_60_1_50_AP	722632	2.5%	40.25	712368	1.1%	6.27	704828	0.0%	14.87
80_500_60_69_60_1_50_USA	632672	0.0%	35.09	641698	1.4%	16.58	638106	0.9%	33.81
80_800_69_50_80_1_60_AP	837210	1.4%	39.53	825392	0.0%	6.69	825392	0.0%	16.04
80_800_69_50_80_1_60_USA	673561	4.5%	36.57	644728	0.0%	14.90	644728	0.0%	34.76
85_500_60_69_60_1_50_AP	762920	1.7%	62.71	759237	1.2%	9.42	750455	0.0%	21.76
85_500_60_69_60_1_50_USA	777825	7.1%	53.17	738851	1.8%	20.47	725993	0.0%	48.85
85_800_69_50_80_1_60_AP	903683	4.2%	60.42	880568	1.5%	9.36	867232	0.0%	22.67
90_500_60_69_60_1_50_USA	804494	7.0%	53.98	752493	0.1%	32.56	751816	0.0%	68.49
90_600_60_69_60_1_69_AP	759377	0.3%	70.56	756916	0.0%	12.17	756916	0.0%	24.57
90_600_60_69_60_1_69_USA	708086	2.0%	54.34	694659	0.0%	26.97	694320	0.0%	54.07
90_800_69_50_80_1_60_AP	966669	4.6%	70.72	931584	0.8%	12.43	924069	0.0%	25.85
95_500_60_69_60_1_50_AP	905808	6.8%	75.52	848766	0.1%	15.60	848132	0.0%	30.09
95_500_60_69_60_1_50_USA	780646	4.1%	64.88	758687	1.2%	32.22	749775	0.0%	70.33
95_600_60_69_60_1_69_AP	826451	2.0%	82.84	814028	0.5%	13.04	810103	0.0%	28.94
95_600_60_69_60_1_69_USA	714260	3.7%	65.63	692781	0.6%	30.65	688970	0.0%	72.09
100_500_60_69_60_1_50_USA	792405	0.0%	77.82	794332	0.2%	40.48	794332	0.2%	91.62

Table 11. SO and AMP on large size instances

	SO			AMP_10			AMP_20		
Instance	***Value***	***Dev***	***CPU***	***Value***	***Dev***	***CPU***	***Value***	***Dev***	***CPU***
110_500_60_69_60_1_50_AP	996975	3.1%	150.65	970622	0.4%	26.99	967134	0.0%	60.89
110_600_60_69_60_1_69_AP	934826	2.3%	142.15	915806	0.2%	24.99	914181	0.0%	52.43
110_700_80_60_89_1_60_USA	1122255	13.6%	105.91	988074	0.0%	58.69	988074	0.0%	133.62
110_800_69_50_80_1_60_USA	1039753	13.1%	117.52	943367	2.6%	62.79	919493	0.0%	143.06
110_900_69_50_89_1_60_USA	1105052	8.6%	105.43	1025893	0.8%	52.98	1017557	0.0%	136.96
120_500_60_69_60_1_50_AP	1123004	8.0%	206.13	1053603	1.3%	43.59	1039932	0.0%	81.98
120_500_60_69_60_1_50_USA	1117652	14.3%	151.31	977587	0.0%	87.44	977587	0.0%	199.98
120_700_80_60_89_1_60_USA	1219108	11.5%	163.57	1093189	0.0%	74.66	1093189	0.0%	189.00
120_900_69_50_89_1_60_USA	1175745	8.8%	146.52	1103461	2.1%	83.34	1080591	0.0%	204.63
125_500_60_69_60_1_50_AP	1145428	4.7%	192.06	1093738	0.0%	66.02	1093577	0.0%	131.42
125_800_69_50_80_1_60_AP	1371476	9.9%	208.84	1265452	1.4%	45.52	1247724	0.0%	105.47
130_600_60_69_60_1_69_AP	1158759	8.5%	291.91	1067512	0.0%	67.16	1067512	0.0%	144.30
130_600_60_69_60_1_69_USA	1040651	6.3%	180.23	984860	0.6%	107.21	978893	0.0%	254.72
130_800_69_50_80_1_60_USA	1264063	14.2%	227.08	1106959	0.0%	117.71	1106959	0.0%	253.02
135_600_60_69_60_1_69_AP	1229722	9.9%	335.92	1118752	0.0%	66.55	1118752	0.0%	145.82
135_800_69_50_80_1_60_USA	1405752	24.8%	251.49	1126503	0.0%	132.34	1126503	0.0%	296.23
140_500_60_69_60_1_50_AP	1412597	11.1%	357.75	1271185	0.0%	71.54	1271185	0.0%	162.32
140_700_80_60_89_1_60_USA	1490007	21.6%	286.76	1225642	0.0%	165.41	1225642	0.0%	296.92
140_800_69_50_80_1_60_AP	1570932	8.9%	377.44	1442113	0.0%	83.24	1442113	0.0%	184.51
140_900_69_50_89_1_60_USA	1444192	16.2%	275.01	1242773	0.0%	136.96	1242773	0.0%	283.20
145_600_80_69_60_1_50_AP	1462902	7.8%	352.66	1368251	0.9%	90.17	1356427	0.0%	217.72
145_600_80_69_60_1_50_USA	596573	16.2%	710.75	519791	1.2%	112.31	513607	0.0%	239.69
145_800_69_50_80_1_60_AP	1603039	7.4%	357.12	1517218	1.6%	106.58	1493074	0.0%	230.67
145_800_69_50_80_1_60_USA	654681	8.4%	647.73	628893	4.1%	129.81	604117	0.0%	259.14
150_1000_69_60_80_1_69_USA	612031	5.8%	837.07	578217	0.0%	135.62	578217	0.0%	270.54
150_800_69_50_80_1_60_AP	1734477	11.1%	424.82	1584133	1.5%	114.37	1561333	0.0%	274.49
150_900_69_60_80_1_89_USA	606912	9.7%	775.96	562431	1.7%	136.29	553060	0.0%	293.43

Table 12. SO and AMP on extra-large size instances

	SO			AMP_10			AMP_20		
Instance	***Value***	***Dev***	***CPU***	***Value***	***Dev***	***CPU***	***Value***	***Dev***	***CPU***
155_1000_69_60_80_1_69_USA	650419	13.9%	919.46	571084	0.0%	149.53	571084	0.0%	304.79
155_500_60_69_60_1_50_AP	1427231	16.5%	635.50	1240808	1.2%	124.74	1225509	0.0%	279.15
155_800_69_50_80_1_60_AP	1602365	13.6%	604.55	1442949	2.3%	127.36	1410360	0.0%	275.13
160_600_60_69_60_1_69_AP	685604	13.5%	950.80	613597	1.5%	58.83	604311	0.0%	126.61
160_700_80_60_89_1_60_USA	704872	24.3%	1010.66	588205	3.8%	194.77	566847	0.0%	369.32
160_800_69_50_80_1_60_AP	826772	9.0%	1042.87	786230	3.6%	82.48	758780	0.0%	176.89
160_900_80_50_60_1_69_AP	775178	7.2%	1008.39	741112	2.5%	79.78	722878	0.0%	159.69
160_900_89_50_60_1_69_USA	530376	12.4%	1048.41	481774	2.1%	163.12	471754	0.0%	339.01
165_1000_69_60_80_1_69_USA	665409	9.4%	1302.26	625185	2.8%	172.64	608228	0.0%	369.96
165_800_69_50_80_1_60_AP	876244	14.7%	1109.94	764063	0.0%	114.51	764063	0.0%	227.33
165_800_69_50_80_1_60_USA	704389	6.1%	1302.59	667884	0.6%	244.26	663785	0.0%	466.76
170_500_60_69_60_1_50_AP	714822	21.0%	1152.12	590908	0.0%	168.00	590908	0.0%	293.91
170_600_89_60_69_1_80_USA	551600	15.5%	1304.46	491632	2.9%	260.48	477564	0.0%	485.13
170_700_80_60_89_1_60_USA	609581	6.7%	1361.95	605758	6.0%	240.73	571293	0.0%	471.38
170_900_69_60_80_1_89_USA	613150	4.8%	1403.26	589807	0.8%	196.72	585232	0.0%	474.49
170_900_80_50_60_1_69_AP	754262	2.4%	1251.03	742642	0.8%	116.22	736391	0.0%	266.60
175_500_60_69_60_1_50_AP	649148	8.4%	1427.86	598789	0.0%	100.08	598789	0.0%	206.36
175_600_60_69_60_1_69_AP	648587	5.5%	1664.50	615051	0.0%	134.62	615051	0.0%	291.06
175_800_69_50_80_1_60_USA	716778	6.3%	1625.83	725441	7.6%	261.61	674337	0.0%	559.54
175_900_69_60_80_1_89_USA	602257	0.1%	1434.86	601938	0.0%	201.47	601938	0.0%	447.89
180_1000_69_60_80_1_69_USA	740453	9.8%	1970.72	674578	0.0%	238.30	674578	0.0%	480.79
180_600_60_69_60_1_69_AP	746918	19.5%	1804.62	625016	0.0%	168.64	625016	0.0%	377.00
180_600_89_60_69_1_80_USA	558479	11.8%	1739.88	501098	0.4%	284.38	499331	0.0%	599.79
180_800_89_69_89_1_89_AP	797621	8.8%	1921.67	738870	0.8%	151.14	733082	0.0%	334.86
185_500_60_69_60_1_50_AP	858506	36.3%	1872.10	629924	0.0%	113.62	629924	0.0%	336.56
185_600_80_89_89_1_89_AP	700282	5.5%	2029.67	663550	0.0%	192.09	663550	0.0%	424.61
185_600_89_60_69_1_80_USA	530783	5.9%	1880.00	501049	0.0%	272.53	501049	0.0%	656.61
185_800_69_50_80_1_60_AP	890070	9.5%	1971.64	813219	0.0%	147.86	813219	0.0%	334.44
185_900_69_60_80_1_89_USA	621650	7.4%	2158.96	579041	0.0%	264.73	579041	0.0%	511.57
190_600_60_69_60_1_69_AP	773840	20.5%	2200.58	642125	0.0%	163.49	642125	0.0%	354.57
190_600_80_89_89_1_89_AP	800719	20.5%	2288.98	667778	0.5%	216.77	664307	0.0%	426.08
190_600_89_60_69_1_80_USA	522508	5.4%	2190.87	506162	2.1%	386.28	495751	0.0%	818.66
190_700_89_69_89_1_89_USA	575933	10.9%	2177.05	531110	2.2%	355.05	519531	0.0%	720.86
190_800_69_50_80_1_60_AP	1031614	27.1%	2298.88	811776	0.0%	166.27	811776	0.0%	379.62
195_600_60_69_60_1_69_AP	774165	18.8%	2487.11	651750	0.0%	170.76	651750	0.0%	401.27
195_800_69_50_80_1_60_AP	1107739	34.2%	2431.53	825666	0.0%	249.33	825402	0.0%	500.46
195_900_89_89_89_1_69_AP	1093874	26.2%	2449.22	866814	0.0%	142.29	866814	0.0%	470.57

Table 13. SO and AMP on huge size instances

	SO			AMP_10			AMP_20		
Instance	***Value***	***Dev***	***CPU***	***Value***	***Dev***	***CPU***	***Value***	***Dev***	***CPU***
200_500_60_69_60_1_50_AP	661633	5.0%	2686.72	630174	0.0%	205.88	630174	0.0%	456.88
200_700_80_60_89_1_60_USA	743444	17.5%	2976.42	632652	0.0%	422.74	632652	0.0%	912.87
200_700_89_69_89_1_89_USA	608272	10.0%	2985.89	558004	0.9%	444.05	552794	0.0%	1046.31
200_800_69_50_80_1_60_AP	763358	5.0%	2731.83	726953	0.0%	202.12	726953	0.0%	450.04
200_800_89_69_89_1_89_USA	645449	11.9%	2995.71	577027	0.0%	429.62	577027	0.0%	860.19
205_800_69_50_80_1_60_USA	800686	10.7%	3086.01	723185	0.0%	494.43	723185	0.0%	1002.65
205_900_69_60_80_1_89_USA	670343	6.2%	3570.73	641338	1.6%	477.98	631462	0.0%	881.68
210_800_69_50_80_1_60_USA	811601	9.9%	3832.57	761961	3.2%	474.52	738677	0.0%	1027.60
210_900_69_60_80_1_89_USA	699431	8.9%	3743.74	642226	0.0%	455.71	642226	0.0%	994.66
215_800_69_50_80_1_60_USA	897348	18.7%	3927.14	776169	2.7%	508.17	756041	0.0%	1067.44
215_900_69_60_80_1_89_USA	736413	11.8%	3653.79	662574	0.6%	440.71	658439	0.0%	952.68
220_800_69_50_80_1_60_USA	872528	11.9%	4275.09	785013	0.7%	669.57	779880	0.0%	1414.82
220_900_69_60_80_1_89_USA	725843	8.8%	4485.24	667050	0.0%	661.34	667050	0.0%	1204.99
225_800_69_50_80_1_60_USA	978703	7.4%	4993.80	911123	0.0%	689.23	911123	0.0%	1399.77
225_900_69_60_80_1_89_USA	812905	9.6%	4668.05	752181	1.4%	711.43	741512	0.0%	1366.89
230_800_69_50_80_1_60_USA	994501	11.5%	6096.23	892018	0.0%	694.07	892018	0.0%	1562.39
230_900_69_60_80_1_89_USA	859790	14.3%	5934.23	765328	1.8%	742.37	751951	0.0%	1509.29
235_800_69_50_80_1_60_USA	1067897	7.5%	6223.85	1026170	3.3%	800.88	993119	0.0%	1734.28
235_900_69_60_80_1_89_USA	967918	20.0%	6017.28	835859	3.7%	768.59	806302	0.0%	1662.56
240_800_69_50_80_1_60_USA	1106592	12.2%	6079.38	1019676	3.4%	1122.78	985860	0.0%	2515.60
240_900_69_60_80_1_89_USA	914468	12.2%	6852.66	846005	3.8%	788.71	815008	0.0%	1769.56
245_800_69_50_80_1_60_USA	1209802	20.8%	7913.75	1001725	0.0%	1264.32	1001725	0.0%	2322.69
245_900_69_60_80_1_89_USA	913041	8.5%	7197.28	841321	0.0%	903.16	841321	0.0%	1799.31
250_800_69_50_80_1_60_USA	1132659	11.7%	7800.79	1045446	3.1%	1299.77	1013736	0.0%	2508.24
250_900_69_60_80_1_89_USA	991167	17.2%	8180.05	845767	0.0%	1302.52	845767	0.0%	2668.66