

# A Hybrid Multistart Heuristic for the Uncapacitated Facility Location Problem

Mauricio G. C. Resende\*      Renato F. Werneck†

February 1, 2005

## Abstract

We present a multistart heuristic for the uncapacitated facility location problem, based on a very successful method we originally developed for the  $p$ -median problem. We show extensive empirical evidence to the effectiveness of our algorithm in practice. For most benchmarks instances in the literature, we obtain solutions that are either optimal or a fraction of a percentage point away from it. Even for pathological instances (created with the sole purpose of being hard to tackle), our algorithm can get very close to optimality if given enough time. It consistently outperforms other heuristics in the literature.

## 1 Introduction

Consider a set  $F$  of *potential facilities*, each with a *setup cost*  $c(f)$ , and let  $U$  be a set of *users* (or *customers*) that must be served by these facilities. The cost of serving user  $u$  with facility  $f$  is given by the *distance*  $d(u, f)$  between them (often referred to as *service cost* or *connection cost* as well). The *facility location problem* consists in determining a set  $S \subseteq F$  of facilities to open so as to minimize the total cost (including setup and service) of covering all customers:

$$\text{cost}(S) = \sum_{f \in S} c(f) + \sum_{u \in U} \min_{f \in S} d(u, f).$$

Note that we assume that each user is allocated to the closest open facility, and that this is the *uncapacitated* version of the problem: there is no limit to the number of users a facility can serve. Even with this assumption, the problem is NP-hard [8].

---

\*AT&T Labs Research, 180 Park Avenue, Florham Park, NJ 07932. Electronic address: [mocr@research.att.com](mailto:mocr@research.att.com).

†Department of Computer Science, Princeton University, 35 Olden Street, Princeton, NJ 08544. Electronic address: [rwerneck@cs.princeton.edu](mailto:rwerneck@cs.princeton.edu). Research at Princeton University supported by the Aladdin project, NSF Grant no. CCR-9626862. The results presented in this paper were obtained while this author was a summer intern at AT&T Labs Research.

This is perhaps the most common location problem, having been widely studied in the literature, both in theory and in practice.

Exact algorithms for this problem do exist (some examples are [7, 25]), but its NP-hard nature makes heuristics the natural choice for larger instances.

Ideally, one would like to find heuristics with good performance guarantees. Indeed, much progress has been made in terms of approximation algorithms for the metric version of this problem (in which all distances obey the triangle inequality). In 1997, Shmoys et al. [37] presented the first polynomial-time algorithm with a constant approximation factor (roughly 3.16). Several improved algorithms have been developed since then, with some of the latest [21, 22, 29] being able to find solutions within a factor of around 1.5 from the optimum. Unfortunately, there is not much room for improvement in this area. Guha and Khuller [16] have established a lower bound of 1.463 for the approximation factor, under some widely believed assumptions.

In practice, however, these algorithms tend to be much closer to optimality for non-pathological instances. The best algorithm proposed by Jain et al. in [21], for example, has a performance guarantee of only 1.61, but was always within 2% of optimality in their experimental evaluation.

Although interesting in theory, approximation algorithms are often outperformed in practice by more straightforward heuristics with no particular performance guarantees. Constructive algorithms and local search methods for this problem have been used for decades, starting from the pioneering work of Kuehn and Hamburger [27]. Since then, more sophisticated metaheuristics have been applied, such as simulated annealing [2], genetic algorithms [26], tabu search [13, 31, 38, 39], and the so-called “complete local search with memory” [13]. Dual-based methods, such as Erlenkotter’s dual ascent [10], Guignard’s Lagrangean dual ascent [17], and Barahona and Chudak’s volume algorithm [3] have also shown promising results.

An experimental comparison of some state-of-the-art heuristics is presented by Hoeyer in [20] (slightly more detailed results are presented in [18]). Five algorithms are tested: JMS, an approximation algorithm presented by Jain et al. in [22]; MYZ, also an approximation algorithm, this one by Mahdian et al. [29]; swap-based local search; Michel and Van Hentenryck’s tabu search [31]; and the volume algorithm [3]. Hoeyer’s conclusion, based on experimental evidence, is that tabu search finds the best solutions within reasonable time, and recommends this method for practitioners.

In this paper, we provide an alternative that can be even better in practice. It is a hybrid multistart heuristic akin to the one we developed for the  $p$ -median problem in [36]. A series of minor adaptations is enough to build a very robust algorithm for the facility location problem, capable of obtaining near-optimal solutions for a wide variety of instances of the facility location problem.

The remainder of the paper is organized as follows. In Section 2, we describe our algorithm and its constituent parts. Section 3 presents empirical evidence to the effectiveness of our method, including a comparison with Michel and Van Hentenryck’s tabu search. Final remarks are made in Section 4.

```

function HYBRID (seed, maxit, elitesize)
1  randomize(seed);
2  init(elite, elitesize);
3  for i = 1 to maxit do
4      S ← randomizedBuild();
5      S ← localSearch(S);
6      S' ← select(elite, S);
7      if (S' ≠ NULL) then
8          S' ← pathRelinking(S, S');
9          add(elite, S');
10     endif
11     add(elite, S);
12 endfor
13 S ← postOptimize(elite);
14 return S;
end HYBRID

```

Figure 1: Pseudocode for HYBRID, as given in [36].

## 2 The Algorithm

In [36], we introduce a new hybrid metaheuristic and apply it to the  $p$ -median problem. Figure 1 reproduces the outline of the algorithm, exactly as presented there.

The method works in two phases. The first is a multistart routine with intensification. In each iteration, it builds a randomized solution and applies local search to it. The resulting solution ( $S$ ) is combined, through a process called *path-relinking*, with some other solution from a pool of *elite solutions* (which represents the best solutions found thus far). This results in a new solution  $S'$ . The algorithm then tries to insert both  $S'$  and  $S$  into the pool; whether any of those is actually inserted depends on its value, among other factors. The second phase is post-optimization, which combines the solutions in the pool with one another in a process that hopefully results in even better solutions.

We call this method HYBRID because it combines elements of several other metaheuristics, such as scatter and tabu search (which make heavy use of path-relinking) and genetic algorithms (from which we take the notion of generations). A more detailed analysis of these similarities is presented in [36].

Of course, Figure 1 presents only the outline of an algorithm. Many details are left to be specified, including which problem it is supposed to solve. Although originally proposed for the  $p$ -median problem, there is no specific mention to it in the code, and in fact the same framework could be applied to other problems. In this paper, our choice is facility location.

Recall that the  $p$ -median problem is very similar to facility location: the only difference is that, instead of assigning setup costs to facilities, the  $p$ -median

problem must specify  $p$ , the exact number of facilities that must be opened. With minor adaptations, we can reuse several of the components used in [36], such as the constructive algorithm, local search, and path-relinking.

The adaptation of the  $p$ -median heuristic shown in this paper is as straightforward as possible. Although some problem-specific tuning could lead to better results, the potential difference is unlikely to be worth the effort. We therefore settle for simple, easy-to-code variations of the original method.

**Constructive heuristic.** In each iteration  $i$ , we first define the number of facilities  $p_i$  that will be open. This number is  $\lceil m/2 \rceil$  in the first iteration; for  $i > 1$ , we pick the average number of facilities in the solutions found (after local search) in the first  $i - 1$  iterations. Now that we have  $p_i$ , we execute the **sample** procedure exactly as described in [36]. It adds facilities one by one. In each step, the algorithm chooses  $\lceil \log_2(m/p_i) \rceil$  facilities uniformly at random and selects the one among those that reduces the total service cost the most.

**Local search.** The local search used in [36] is based on swapping facilities. Given a solution  $S$ , we look for two facilities,  $f_r \in S$  and  $f_i \notin S$ , which, if swapped, lead to a better solution. A property of this method is that it keeps the number of open facilities constant. This is required for the  $p$ -median problem, but not for facility location, so in this paper we also allow “pure” insertions and deletions (in addition to swaps). All possible insertions, deletions, and swaps are considered, and the best among those is performed. The local search stops when no improving move exists, in which case the current solution is a *local minimum* (or *local optimum*). This local search is known as *flip + swap* [23].

The actual implementation of the local search is essentially the same used in [36] (and described in detail in [34, 35]) for the  $p$ -median problem. We briefly recall the main ideas here. Let  $profit(f_i, f_r)$  be the amount by which the solution value is reduced if  $f_i$  is the facility inserted and  $f_r$  the one removed. The algorithm computes the profit associated to every pair  $(f_i, f_r)$  (with  $f_i \in S$  and  $f_r \notin S$ ). If the maximum profit is positive, we perform the corresponding swap and repeat; otherwise, we stop. The computation is divided into three components:

- $save(f_i)$ : decrease in solution value due to the insertion of  $f_i$  (with no associated removal);
- $loss(f_r)$ : increase in solution value due to the removal of  $f_r$  (with no associated insertion);
- $extra(f_i, f_r)$ : a positive correction term that accounts for the fact that the insertion of  $f_i$  and the removal of  $f_r$  may not be independent (a user previously assigned to  $f_r$  may be reassigned to  $f_i$ ); the definition of  $extra$  is such that the following relation holds:

$$profit(f_i, f_r) = save(f_i) - loss(f_r) + extra(f_i, f_r).$$

Instead of computing these values from scratch in each iteration, our implementation just updates them from one iteration of the local search to another. To achieve this goal, *save* and *loss* are represented as arrays; *extra*, being the only term that depends on both  $f_i$  and  $f_r$ , is represented as a matrix.

It can be shown [35] that  $extra(f_i, f_r)$  is nonzero only when  $f_i$  and  $f_r$  are “close” to each other.<sup>1</sup> One only has to worry explicitly about the nonzero terms; all others are determined implicitly. This observation is crucial for a fast implementation of the local search procedure, since it allows *extra* to be represented as a sparse matrix. In instances from the literature, this implementation has been shown [35] to be up to three orders of magnitude faster than previous methods (even though its worst-case complexity,  $O(mn)$ , is the same) for the  $p$ -median problem.

As mentioned in [34], this algorithm can be adapted to the facility location problem in a very natural way. First, we must take setup costs into account, which can be accomplished simply by subtracting them from *save* and *loss*. Second, we must consider that single insertions or deletions are now valid moves (and not only swaps). But this comes for free: *save* and *loss* already represent the profits obtained with insertions and deletions, respectively. These are the only differences between the algorithms.

**Path-relinking.** Path-relinking is an intensification procedure originally devised for scatter search and tabu search [14, 15, 28], but often used with other methods, such as GRASP [32, 33]. In this paper, we apply the variant described in [36]. It takes two solutions as input,  $S_1$  and  $S_2$ . The algorithm starts from  $S_1$  and gradually transforms it into  $S_2$ . The operations that change the solution in each step are the same used in the local search: insertions, deletions, and swaps. In this case, however, only facilities in  $S_2 \setminus S_1$  can be inserted, and only those in  $S_1 \setminus S_2$  can be removed. In each step, the most profitable (or least costly) move—considering all three kinds—is performed. The procedure returns the best local optimum in the path from  $S_1$  to  $S_2$ . If no local optimum exists, one of the extremes is chosen with equal probability.

**Elite solutions.** The *add* operation in Figure 1 must decide whether a new solution should be inserted into the pool or not. The criteria we use here are similar to those proposed in [36]. They are based on the notion of *symmetric difference* between two solutions  $S_a$  and  $S_b$ , defined as  $|S_a \setminus S_b| + |S_b \setminus S_a|$ .<sup>2</sup> A new solution will be inserted into the pool only if its symmetric difference to each cheaper solution already there is at least four. Moreover, if the pool is full, the new solution must also cost less than the most expensive element in the pool; in that case, the new solution replaces the one (among those of equal or greater cost) it is most similar to.

---

<sup>1</sup>More precisely, when there is at least one user for which  $f_i$  is closest than the second closest facility in the original solution.

<sup>2</sup>This definition is slightly different from the one we used for the  $p$ -median problem, since now different solutions need not have the same number of facilities.

**Intensification.** After each iteration, the solution  $S$  obtained by the local search procedure is combined (with path-relinking) with a solution  $S'$  obtained from the pool, as shown in line 8 of Figure 1. Solution  $S'$  is chosen at random, with probability proportional to its symmetric difference to  $S$ . Path-relinking is always performed from the best to the worst solution among the two.

**Post-optimization.** Once the multistart phase is over, all elite solutions are combined with one another, also with path-relinking (within each pair, path-relinking is performed from the worst to the best solution). The solutions thus produced are used to create a new pool of elite solutions (subject to the same rules as in the original pool), to which we refer as a new *generation*. If the best solution in the new generation is strictly better than the best previously found, we repeat the procedure. This process continues until a generation that does not improve upon the previous one is created. The best solution found across all generations is returned as the final result of the algorithm.

**Parameters.** As the outline in Figure 1 shows, the procedure takes only two input parameters (other than the random seed): the number of iterations in the multistart phase and the size of the pool of elite solutions. In [36], we set those values to 32 and 10, respectively. In the spirit of keeping changes to a minimum, we use the same values here for the “standard version” of our algorithm.

Whenever we need versions of our algorithm with shorter or longer running times (to ensure a fair comparison with other methods), we change both parameters. Recall that the running time of the multistart phase of the algorithm depends linearly on the number of iterations, whereas the post-optimization phase depends quadratically (roughly) on the number of elite solutions (because all solutions are combined among themselves). Therefore, if we want to multiply the average running time of the algorithm by some factor  $x$ , we just multiply the number of multistart iterations by  $x$  and the number of elite solutions by  $\sqrt{x}$  (rounding appropriately).

We observe that running time and solution quality are determined by several design choices, and not only the number of iterations and of elite solutions. Consider the intensification strategies, for instance. To reduce the running time of the algorithm, we could decide not to run the post-optimization phase, or not to run path-relinking during the multistart phase (or not at all). Or, to increase solution quality, we could consider performing path-relinking between two solutions  $S_1$  and  $S_2$  in both ways (from  $S_1$  to  $S_2$  and from  $S_2$  to  $S_1$ ) and picking the best. We could also try other constructive heuristics. These and other variants of the algorithm are studied in the context of the  $p$ -median problem in [36]. The variant reported here achieved the best balance overall between running time and solution quality. The most important aspects of the algorithm are the fast implementation of the local search and the use of path-relinking. Other aspects, such as the constructive heuristic, the methods for maintaining the pool of elite solutions, and the precise criteria for adding and removing solutions from it played relatively minor roles.

## 3 Empirical Results

### 3.1 Experimental Setup

The algorithm was implemented in C++ and compiled with the SGI MIPSPro C++ compiler (v. 7.30) with flags `-O3 -OPT:Olimit=6586`. The program was run on an SGI Challenge with 28 196-MHz MIPS R10000 processors, but each execution was limited to a single processor. All times reported are CPU times measured by the `getrusage` function with a precision of 1/60 second. The random number generator we used was Matsumoto and Nishimura's *Mersenne Twister* [30]. The source code for the algorithm is available from the authors upon request.

The algorithm was tested on all classes from the UfLib [20] at the time of writing and on class GHOSH, described in [13]. In every case, the number of users is the same as the number of potential facilities. The reader is referred to [19] and [13] for detailed descriptions of each class. A brief overview is presented below:

- BK: Generated based on the description provided by Bilde and Krarup [6]. There are 220 instances in total, with 30 to 100 users. Connection costs are always picked uniformly at random from  $[0, 1000]$ . Setup costs are always at least 1000, but the exact range depends on the subclass (there are 22 of those, with 10 instances each).
- FPP: Class introduced by Kochetov [23, 24]. Each instance corresponds to a finite projective plane of order  $k$ , with  $n = k^2 + k + 1$  points and  $n$  lines. In the UFL instance, the distance between  $i$  and  $j$  is an integer between 0 and 4 if point  $j$  is on line  $i$ , and infinity otherwise; at most  $n + 1$  values are finite. Setup costs are 3000. There are two subclasses, each with 40 instances: FPP11 (with  $k = 11$  and  $n = 133$ ) and FPP17 (with  $k = 17$  and  $n = 307$ ). Although optimal solutions in this class can be found in polynomial time, the instances are hard for algorithms based on the flip+swap local search, since each instance has a large number of strong local optima and the distance between them is at least  $2k$  [23].
- GAP: Also designed by Kochetov [23, 24], these instances have large duality gaps, usually greater than 20%. They are hard especially for dual-based methods. Setup costs are always 3000. The service cost associated with each facility is infinity for most customers, and between 0 and 5 for the remaining few (the end result resembles a set covering instance). There are three subclasses (GAPA, GAPB, and GAPC), each with 30 instances. Each customer in GAPA is covered by 10 facilities (i.e., the service cost for all others is infinity); each facility in GAPB covers exactly 10 customers; subclass GAPC (the hardest) combines both constraints: each customer is covered by 10 facilities, and each facility covers 10 customers. On all cases, assignments are made at random.

- **GHOSH**: Class created by Ghosh in [13], following the guidelines set up by Körkel in [25]. There are 90 instances in total, with  $n = m$  on all cases. They are divided into two groups of 45 instances, one symmetric and the other asymmetric. Each group contains three values of  $n$ : 250, 500, and 750.<sup>3</sup> Connection costs are integers taken uniformly at random from [1000, 2000]. For each value of  $n$  there are three subclasses, each with five instances; they differ in the range of values from which setup costs are drawn: it can be [100, 200] (range A), [1000, 2000] (B) or [10000, 20000] (C). Each subclass is named after its parameters: **GS250B**, for example, is symmetric, has 250 nodes, and service costs ranging from 1000 to 2000.
- **GR**: Graph-based instances by Galvão and Raggi [11]. There are 50 instances in total, 10 for each value of  $n$  (50, 70, 100, 150, and 200). Connection costs are given by the corresponding shortest paths in the underlying graph. (Instances are actually given as distance matrices, so there is no overhead associated with computing shortest paths.)
- **M\***: This class was created with the generator introduced by Kratica et al. in [26]. These instances have several near-optimal solutions, which according the authors makes them close to “real-life” applications. There are 22 instances in this class, with  $n$  ranging from 100 to 2000.
- **MED**: Originally proposed for the  $p$ -median problem by Ahn et al. in [1], these instances were later used in the context of uncapacitated facility location by Barahona and Chudak [3]. Each instance is a set of  $n$  points picked uniformly at random in the unit square. A point represents both a user and a potential facility, and connection costs are determined by the corresponding Euclidean distances. All values are rounded up to 4 significant digits and made integer [20]. Six values of  $n$  were used: 500, 1000, 1500, 2000, 2500, and 3000. In each case, three different setup costs were tested:  $\sqrt{n}/10$ ,  $\sqrt{n}/100$ , and  $\sqrt{n}/1000$ .
- **ORLIB**: These instances are part of Beasley’s OR-Library [4]. Originally proposed as instances for the capacitated version of the facility location problem in [5], they can be used in the uncapacitated setting as well (one just has to ignore the capacities).

All instances were downloaded from the UfLib website [19], with the exception of those in class **GHOSH**, created with a generator kindly provided by D. Ghosh [12]. Five of these eight classes were used in Hofer’s comparative analysis [18, 20]: **BK**, **GR**, **M\***, **MED**, and **ORLIB**.

---

<sup>3</sup>These are actually the three largest values tested in [13]; some smaller instances are tested there as well.

## 3.2 Results

### 3.2.1 Quality Assessment

As already mentioned, the “standard” version of our algorithm has 32 multistart iterations and 10 elite solutions. It was run ten times on each instance available, with ten different random seeds (1 to 10).

Although more complete data will be presented later in this section, we start with a broad overview of the results we obtained. Table 1 shows the average deviation (in percentage terms) obtained by our algorithm with respect to the best known bounds. All optima are known for BK, GR, and ORLIB. We used the best upper bounds shown in [19] at the time of writing for FPP, GAP, MED and M\* (upper bounds that are not proved optimal were obtained by various algorithms, including tabu search and local search). For GHOSH, we used the bounds shown in [13]; some were obtained by tabu search, others by complete local search with memory. Table 1 also shows the mean running times obtained by our algorithm.

To avoid giving too much weight to larger instances, we used geometric means for times.

Table 1: Average deviation with respect to the best known upper bounds and mean running times of HYBRID (with 32 iterations and 10 elite solutions) for each class.

CLASS	AVG%DEV	TIME (s)
BK	0.002	0.28
FPP	33.375	7.66
GAP	5.953	1.64
GHOSH	-0.039	34.31
GR	0.000	0.32
M*	0.000	7.86
MED	-0.391	369.67
ORLIB	0.000	0.17

In terms of solution quality, our algorithm does exceedingly well for all five classes tested in [18]. It matched the best known bounds (usually the optimum) on every single run of GR, M\*, and ORLIB. The algorithm did have a few unlucky runs on class BK, but the average error was still only 0.001%. On MED, the solutions it found were on average 0.4% better than the best upper bounds shown in [18].

Our method also handles very well the only class not in the UfLib, GHOSH. It found solutions at least as good as the best in [13]. This is especially relevant considering that we are actually comparing our results with the best among *two* algorithms in each case (tabu search and complete local search with memory).

The two remaining classes, GAP and FPP, were created with the intent of being hard. At least for our algorithm, they definitely are: on average, solutions

were within 28% and 6% from optimality, respectively. This is several orders of magnitude worse than the results obtained for other classes. However, as Subsection 3.2.2 will show, the algorithm can obtain solutions of much better quality if given more time.

**Detailed results.** For completeness, Tables 2 to 8 show the detailed results obtained HYBRID on each of the eight classes of instances. They refer to the exact same runs used to create Table 1.

Tables 2 and 3 show the results for  $M^*$  and ORLIB, respectively. For each instance, we show the best known bounds (which were matched by our algorithm on all runs on both classes) and the average running time.

Table 2: Results for  $M^*$  instances. Average solution values for HYBRID and mean running times (with 32 iterations and 10 elite solutions). All runs matched the best bounds shown in [18].

NAME	$n$	VALUE	TIME (s)
mo1	100	1305.95	0.988
mo2	100	1432.36	1.030
mo3	100	1516.77	0.960
mo4	100	1442.24	0.892
mo5	100	1408.77	0.815
mp1	200	2686.48	3.695
mp2	200	2904.86	4.125
mp3	200	2623.71	3.500
mp4	200	2938.75	3.887
mp5	200	2932.33	4.169
mq1	300	4091.01	8.919
mq2	300	4028.33	7.802
mq3	300	4275.43	9.508
mq4	300	4235.15	9.834
mq5	300	4080.74	10.813
mr1	500	2608.15	27.221
mr2	500	2654.74	27.646
mr3	500	2788.25	26.417
mr4	500	2756.04	27.595
mr5	500	2505.05	26.989
ms1	1000	5283.76	113.395
mt1	2000	10069.80	701.167

Results for class MED are shown in Table 4. For each instance, we present the best known lower and upper bounds, as given in Table 12 of [18]. Lower bounds were found by the volume algorithm [3], and upper bounds by either local search or tabu search [31], depending on the instance. The average solution value obtained by HYBRID in each case is shown in Table 4 in absolute and percentage terms (in the latter case, when compared with both lower and upper bounds).

Table 3: Results for ORLIB instances. Average running times for HYBRID with 32 iterations and 10 elite solutions. The optimum solution value was found on all runs.

NAME	$n$	OPTIMUM	TIME (S)
cap101	50	796648.44	0.055
cap102	50	854704.20	0.056
cap103	50	893782.11	0.072
cap104	50	928941.75	0.077
cap131	50	793439.56	0.105
cap132	50	851495.32	0.097
cap133	50	893076.71	0.131
cap134	50	928941.75	0.140
cap71	50	932615.75	0.034
cap72	50	977799.40	0.039
cap73	50	1010641.45	0.053
cap74	50	1034976.97	0.049
capa	1000	17156454.48	7.380
capb	1000	12979071.58	6.245
capc	1000	11505594.33	6.148

On average, HYBRID found solutions that are at least 0.15% better than previous bounds, and sometimes the gains were upwards of 0.5%. In fact, our results were in all cases much closer to the lower bound than to previous upper bounds. Average solution values are at most 0.191% away from optimality, possibly less (depending on how good the lower bounds are).

Since classes BK and GR have more instances (220 and 50, respectively), we aggregate them into subclasses. Each subclass contains 10 instances built with the exact same parameters (such as number of elements and cost distribution), just with different random seeds. Table 5 presents the results for BK: for each subclass, we present the average error obtained by the algorithm and the average running time. Table 6 refers to class GR and presents the average running times only, since the optimal solution was found in every single run.

Table 7 shows the results for class GHOSH, which is divided into five-instance subclasses. The table shows the best bounds found in [13], by either tabu search or complete local search with memory (we picked the best in each case). For reference, we also show the running times reported in [13], but the reader should bear in mind that they were found on a machine with a different processor (an Intel Mobile Celeron running at 650 MHz).<sup>4</sup>

The last three columns in the table report the results obtained by HYBRID: the solution value, the average deviation with respect to the upper bounds, and

<sup>4</sup>From [9], we can infer that this processor and the one we use have similar speeds, or at least within the same order of magnitude. The machine in [9] that is most similar to Ghosh's is a Celeron running at 433 MHz, capable of 160 Mflop/s. According to the same list, the speed of our processor is 114 Mflop/s (based on an entry for an SGI Origin 2000 at 195 MHz).

Table 4: Results for MED instances. Columns 2 and 3 show the best known lower and upper bounds, as given in Tables 11 and 12 of [18]. The next three columns show the quality obtained by HYBRID: first the average solution value, then the average percentage deviation from the lower and upper bounds, respectively. The last column shows the average running times of our method.

NAME	LOWER	UPPER	AVERAGE	AVG%L	AVG%U	TIME (s)
med0500-10	798399	800479	798577.0	0.022	-0.238	33.2
med0500-100	326754	328540	326805.4	0.016	-0.528	32.9
med0500-1000	99099	99325	99169.0	0.071	-0.157	23.6
med1000-10	1432737	1439285	1434185.4	0.101	-0.354	173.9
med1000-100	607591	609578	607880.4	0.048	-0.278	148.8
med1000-1000	220479	221736	220560.9	0.037	-0.530	141.7
med1500-10	1997302	2005877	2001121.7	0.191	-0.237	347.8
med1500-100	866231	870182	866493.2	0.030	-0.424	378.7
med1500-1000	334859	336263	334973.2	0.034	-0.384	387.2
med2000-10	2556794	2570231	2558120.8	0.052	-0.471	717.5
med2000-100	1122455	1128392	1122861.9	0.036	-0.490	650.8
med2000-1000	437553	439597	437690.7	0.031	-0.434	760.0
med2500-10	3095135	3114458	3100224.7	0.164	-0.457	1419.5
med2500-100	1346924	1352322	1347577.6	0.049	-0.351	1128.2
med2500-1000	534147	536546	534426.6	0.052	-0.395	1309.4
med3000-10	3567125	3586599	3570818.8	0.104	-0.440	1621.1
med3000-100	1600551	1611186	1602530.9	0.124	-0.537	1977.6
med3000-1000	643265	645680	643541.8	0.043	-0.331	2081.4

Table 5: Results for BK instances: average percent errors with respect to the optima and average running times of HYBRID (with 32 iterations and 10 elite solutions).

SUBCLASS	$n$	AVG%ERR	TIME (S)
B	100	0.0000	0.310
C	100	0.0160	0.450
D01	80	0.0001	0.223
D02	80	0.0000	0.211
D03	80	0.0000	0.199
D04	80	0.0000	0.170
D05	80	0.0000	0.162
D06	80	0.0000	0.186
D07	80	0.0000	0.174
D08	80	0.0000	0.166
D09	80	0.0000	0.175
D10	80	0.0000	0.166
E01	100	0.0000	0.476
E02	100	0.0000	0.588
E03	100	0.0188	0.512
E04	100	0.0000	0.464
E05	100	0.0000	0.376
E06	100	0.0000	0.408
E07	100	0.0000	0.416
E08	100	0.0000	0.418
E09	100	0.0000	0.352
E10	100	0.0000	0.353

Table 6: Results for GR instances: average running times of HYBRID (with 32 iterations and 10 elite solutions) as a function of  $n$  (each subclass contains 10 instances). Every execution found the optimal solution.

$n$	TIME (S)
50	0.098
70	0.163
100	0.308
150	0.602
200	1.123

the running time (all three values are averages taken over the 50 runs in each subclass).

Table 7: Results for GHOSH instances. The upper bounds are the best reported by Ghosh in [13], with the corresponding running times (obtained on a different machine, an Intel Mobile Celeron running at 650 MHz). The results for HYBRID (with 32 iterations and 10 elite solutions) are shown in the last three columns.

INSTANCE		UPPER BOUND [13]		HYBRID		
NAME	$n$	VALUE	TIME (s)	VALUE	AVG%DEV	TIME (s)
GA250A	250	257978.4	18.3	257922.1	-0.022	5.7
GA250B	250	276184.2	6.5	276053.6	-0.047	8.2
GA250C	250	333058.4	17.3	332897.2	-0.048	7.4
GA500A	500	511251.6	18.1	511147.4	-0.020	40.3
GA500B	500	538144.0	6.4	537868.2	-0.051	52.2
GA500C	500	621881.8	24.7	621475.2	-0.065	57.4
GA750A	750	763840.4	213.3	763741.0	-0.013	117.5
GA750B	750	796754.2	71.4	796393.5	-0.045	127.1
GA750C	750	900349.8	146.5	900198.6	-0.017	136.5
GS250A	250	257832.6	207.1	257807.9	-0.010	5.3
GS250B	250	276185.2	79.2	276035.2	-0.054	8.0
GS250C	250	333671.6	134.6	333671.6	0.000	8.3
GS500A	500	511383.6	824.3	511203.0	-0.035	43.5
GS500B	500	538480.4	409.4	537919.1	-0.104	52.6
GS500C	500	621107.2	347.4	621059.2	-0.008	50.8
GS750A	750	763831.2	843.2	763713.9	-0.015	112.6
GS750B	750	796919.0	396.0	796593.7	-0.041	126.3
GS750C	750	901158.4	499.7	900183.8	-0.108	130.3

Finally, average solution qualities and running times are shown to each subclass of FPP and GAP in Table 8.

### 3.2.2 Comparative Analysis

We have seen that our algorithm obtains solutions of remarkable quality for most classes of instances tested. On their own, however, these results do not mean much. Any reasonably scalable algorithm should be able to find good solutions if given enough time.

With that in mind, we compare the results obtained by our algorithm with those obtained by Michel and Van Hentenryck’s tabu search algorithm [31], which achieved the best experimental results among the algorithms tested in [18]. We refer to this method as TABU. Starting from a random solution, in each iteration it executes a *flip* operation, i.e., it opens or closes an individual facility. This defines a neighborhood that is more restricted than the one we use, which also allows swaps. While the best neighbor can be found considerably faster, the local search tends to reach local optima sooner. To escape them, the method

Table 8: Results for FPP and GAP instances: average percent errors (relative to the best upper bounds in [18]) and average running times for each subclass.

SUBCLASS	$n$	AVG%ERR	TIME (S)
GAPA	100	5.14	1.41
GAPB	100	5.98	1.81
GAPC	100	6.74	1.89
FPP11	133	8.48	2.58
FPP17	307	58.27	25.18

uses a tabu list, which forbids facilities recently inserted or removed from being flipped. The algorithm stops after executing 500 consecutive iterations without an improvement in the objective function

We downloaded the source code for an implementation of TABU from the UflLib. To ensure that running times are comparable, we compiled it with the same parameters used for HYBRID and ran the program on the same machine. Since TABU has a randomized component (the initial solution), we ran it 10 times for each instance in the class, with different seeds for the random number generator (the seeds were 1 to 10).

As suggested in [31], the algorithm was run with 500 non-improving consecutive iterations as the stopping criterion. However, with this number of iterations TABU is much faster than the standard version of HYBRID (with 32 iterations and 10 elite solutions). For a fair comparison, we also ran a faster version of our method, with only 8 iterations and 5 elite solutions. The results obtained by this variant of HYBRID and by TABU are summarized in Table 9. For each class, the average solution quality (as the percentage deviation with respect to the upper bound in [18]) and the mean running times are shown.

Table 9: Average deviation with respect to the best known upper bounds and mean running times in each class for HYBRID (with 8 iterations and 5 elite solutions) and TABU (with 500 non-improving iterations as the stop criterion).

CLASS	HYBRID		TABU	
	AVG%DEV	TIME (s)	AVG%DEV	TIME (s)
BK	0.028	0.087	0.071	0.155
FPP	69.367	1.741	95.711	0.650
GAP	9.573	0.348	15.901	0.259
GHOSH	-0.032	8.816	0.002	4.570
GR	0.000	0.090	0.100	0.160
M*	0.004	2.196	0.011	1.750
MED	-0.364	92.387	0.073	92.854
ORLIB	0.000	0.048	0.028	0.160

Note that both algorithms have similar running times, much lower than those presented in Table 1. Even so, both algorithms find solutions very close to the optimal (or best known) on five classes: BK, GHOSH, GR, M\*, and ORLIB. Although in all cases HYBRID found slightly better solutions, both methods performed rather well: on average, TABU was always within 0.1% of the best previously known bounds, and HYBRID was within 0.03%. Our algorithm was actually able to improve the bounds for GHOSH (presented on [13]), whereas TABU could only match them (albeit in slightly less time than HYBRID).

Although there are some minor differences between the algorithms for these five classes, it is not clear which is best. Both usually find the optimal values in comparable times. In a sense, these instances are just too easy for either method. We need to look at the remaining classes to draw any meaningful conclusion.

Consider class MED. Both algorithms run for essentially the same time on average. TABU almost matches the best bounds presented in [18]. This was expected, since most of those bounds were obtained by TABU itself (a few were established by local search). However, HYBRID does much better: on average, the solutions it finds are 0.364% below the reference upper bounds.

Even greater differences were observed for GAP and FPP: these instances are meant to be hard, and indeed they are for both algorithms. On average, the solutions HYBRID found for GAP instances were almost 10% off optimality; TABU did even worse, with an average error of 16%. The hardest class is FPP: the average deviation from optimality was almost 70% for our algorithm, and more than 95% for TABU. Even though HYBRID does slightly better than TABU on both classes, the results it provides are hardly satisfactory for a method that is supposed to find near-optimal solutions.

Note, however, that the mean time spent on each instance is around one second, which is not much. Being implementations of metaheuristics, both algorithms should behave much better if given more time. To test if this is indeed the case, we performed longer runs of each method.

We tested two different versions of tabu search. In the first variant (TABU), we vary the number of iterations in the stopping criterion: 500 (the original value), 1000, 2000, 4000, 8000, 16000, 32000, and 64000. The second variant, TABUMS, is a multistart version of the algorithm. After it reaches 500 non-improving iterations, it starts again from a new (random) solution. The best solution overall is picked as the final result. We tested this method with 1 to 128 restarts, corresponding to 500 to 64000 final non-improving iterations overall.

We also tested three different versions of our algorithm. The first is the standard version depicted in Figure 1, HYBRID. It has two input parameters (number of iterations and number of elite solutions), so we varied both at the same time. We tested the following pairs: 4:3, 8:5, 16:7, 32:10 (the original parameters), 64:14, 128:20, 256:28, 512:40, 1024:57, and 2048:80. Note that to move from one pair to the next we multiply the number of iterations by 2 and the number of elite solutions by  $\sqrt{2}$ , as mentioned in Section 2.

The second version is MULTISTART+PR, which is similar to HYBRID but does not execute post-optimization; however, it does execute path-relinking between

multistart iterations, as in the original HYBRID. The best solution found after all iterations are completed is picked as the result. We ran this algorithm with the same set of parameters (number of iterations and elite solutions) as HYBRID.

The third version we tested (MULTISTART) does not execute path-relinking at any stage: in each iteration, it finds a greedy randomized solution and executes local search on it. The best solution found over all iterations is picked as the result. We also ran it with the same set of parameters as HYBRID, but note that this variant has no use for the set of elite solutions.

The purpose of MULTISTART and MULTISTART+PR is to assess the importance of path-relinking to the overall quality of the algorithm.

The results are summarized in Tables 10 (for our algorithm) and 11 (for tabu search). For each method and choice of parameters, we present the average percentage error and the geometric mean of the running times (in seconds). The same information is represented graphically on Figures 2 (for GAP) and 3 (FPP). Each graph represents the average solution quality as a function of time. They were built directly from Tables 10 and 11: each line in a table became a point in the corresponding graph, and the points were then linearly interpolated.

For these series, TABUMS seems to be a better choice than TABU. The difference is particularly noticeable for the FPP instances, where local optima are very far apart. By restarting the tabu search from random solutions at regular intervals, TABUMS is able to explore the search space more effectively than TABU, which relies only on tabu lists to escape local optima. Our method executes even more starts and has the additional advantage of using a more powerful local search. This helps explain why all three variants (HYBRID, MULTISTART+PR and MULTISTART) obtained significantly better results than tabu search, as the pictures show.

Take class GAP. Within 0.25 second, HYBRID can obtain solutions that are more than 10% off optimality (on average); in 20 seconds, the error is down to 2%. The behavior of MULTISTART+PR is almost identical. MULTISTART is slightly better than both in the beginning, but worse for longer runs, which indicates that path-relinking is important to ensure the robustness of the algorithm. All three variants, however, are better than TABU and TABUMS, whose errors ranged from approximately 16% (in 0.3 second) to around 6% (in 20 seconds).

Even more remarkable differences in performance are observed on class FPP. If given less than one second, all algorithms perform poorly: HYBRID and MULTISTART+PR find solutions that are almost 80% away from optimality on average; TABU and TABUMS are even worse, with 90%; MULTISTART is the best overall, with 70%. However, once longer runs are allowed, HYBRID and (to a lesser degree) MULTISTART+PR improve at a much faster rate than the other methods. Within 50 seconds, HYBRID already finds near-optimal solutions on all cases (the average error is below 0.02%), whereas solutions found by TABUMS are still more than 50% off optimality on average (TABU is even worse, at 70%). Although MULTISTART does significantly better than tabu-based algorithm, it is still at least 5% away from optimality; once again, this shows how important path-relinking is to ensure solution quality.

We stress that such large differences in solution quality are not likely to be

Table 10: Results on hard classes: average percentage errors and mean running times (in seconds). HYBRID refers to the full algorithm, MULTISTART+PR refers to the algorithm without post-optimization (but with path-relinking between multistart iterations), and MULTISTART refers to the version with no path-relinking at all.

CLASS	ITER.	ELITE	MULTISTART		MULTISTART+PR		HYBRID	
			%ERR	TIME	%ERR	TIME	%ERR	TIME
FPP	4	3	90.60	0.19	86.05	0.39	82.79	0.58
	8	5	83.17	0.31	76.76	0.72	69.37	1.63
	16	7	73.44	0.56	64.19	1.39	53.17	3.41
	32	10	62.10	1.04	50.80	2.76	33.37	7.11
	64	14	51.38	2.01	38.00	5.47	15.84	13.61
	128	20	41.67	3.94	22.23	10.86	4.29	25.15
	256	28	33.69	7.80	9.30	21.68	0.02	47.51
	512	40	25.23	15.50	0.56	43.72	0.01	91.50
	1024	57	14.31	30.89	0.01	88.16	0.00	173.78
	2048	80	4.98	61.72	0.00	177.87	0.00	330.88
GAP	4	3	16.39	0.05	14.11	0.10	12.93	0.14
	8	5	13.59	0.09	11.46	0.19	9.57	0.35
	16	7	11.64	0.17	9.36	0.37	7.47	0.77
	32	10	9.73	0.32	7.62	0.73	5.95	1.63
	64	14	8.14	0.63	6.21	1.45	4.69	3.27
	128	20	7.14	1.26	5.06	2.92	3.83	6.47
	256	28	5.95	2.50	3.75	5.90	2.73	12.52
	512	40	5.00	4.99	2.65	11.92	1.67	24.75
	1024	57	3.81	9.96	1.79	23.62	1.17	48.62
	2048	80	2.77	19.89	1.17	46.57	0.82	92.09

Table 11: TABU results for hard classes. Two variants are analyzed: TABU starts from a random solution and executes a “pure” tabu search from there; TABUMS runs tabu with until it executes 500 consecutive non-improving iterations, then repeats the procedure from a new random solution. For each method, we show the average percentage error obtained and the mean running times (in seconds).

CLASS	ITER.	TABU		TABUMS	
		AVG%ERR	TIME (s)	AVG%ERR	TIME (s)
FPP	500	95.62	0.63	95.62	0.63
	1000	94.70	1.07	91.69	1.22
	2000	91.03	2.04	87.93	2.43
	4000	86.81	3.82	82.41	4.84
	8000	83.67	7.26	75.04	9.68
	16000	79.32	14.21	66.02	19.47
	32000	75.16	27.45	57.75	38.98
	64000	71.15	52.08	50.77	77.92
GAP	500	15.98	0.26	15.98	0.26
	1000	13.70	0.48	13.21	0.52
	2000	11.66	0.94	11.26	1.04
	4000	10.62	1.67	9.66	2.03
	8000	8.94	3.24	8.07	4.04
	16000	7.72	6.19	7.03	8.10
	32000	7.02	11.75	6.00	16.21
	64000	6.35	22.43	5.06	32.46

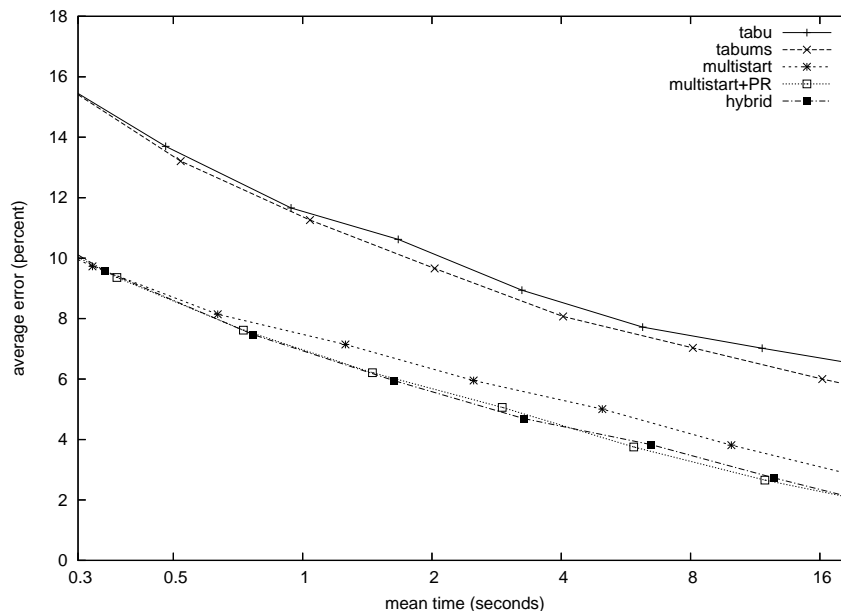


Figure 2: GAP class. Average percent deviation obtained by HYBRID and TABU (and their variants) for several sets of parameters. Data taken from Tables 10 and 11.

observed on “real-world” instances. Classes FPP and GAP are by no means typical, since they were designed with the specific purpose of being hard to solve. For all other classes tested, which have no adversarial structure, the performance of TABU was much closer to that of HYBRID. This, however, does not mean the results for FPP and GAP are irrelevant. Practical instances are unlikely to be as hard as those, but nothing guarantees that they will be as well-behaved as the other classes presented here. Since HYBRID is more robust than TABU in extreme cases, it is more likely to be faster in moderately difficult ones.

As a final observation, we note that when this paper was undergoing minor revisions before publication, a referee pointed us to the work of M. Sun [38, 39], still unpublished. The author proposes another tabu search for the uncapacitated facility location problem, also based on inserting and removing individual facilities (no swaps are performed directly). It solves all ORLIB instances to optimality, as in our case. In the GHOSH series, the solutions it finds are on average slightly better than ours, although there are instances for which HYBRID finds the best solutions. On these two classes (the only ones from the literature tested by the author), the algorithm seems to be at least as effective as ours. We refer the reader to [39] for a more detailed analysis.

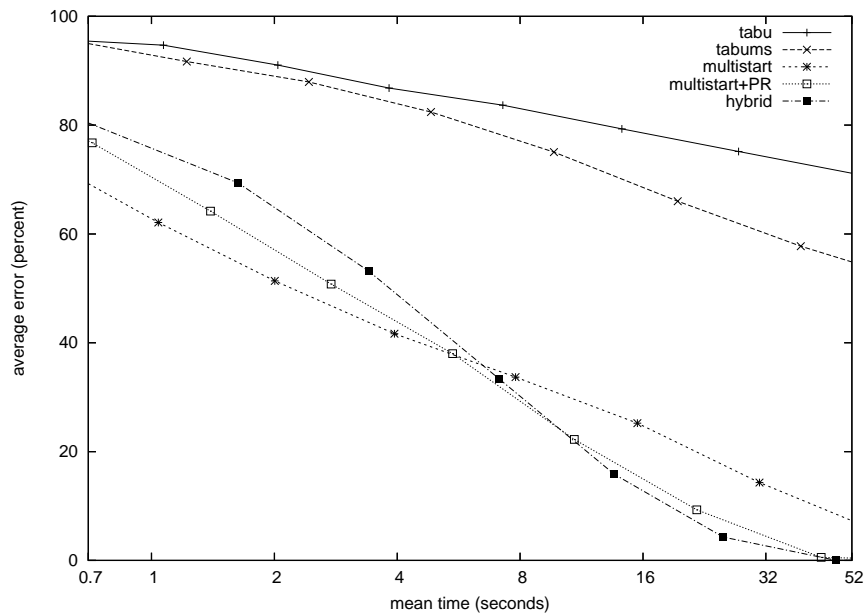


Figure 3: FPP class. Average percentage deviation obtained by HYBRID and TABU (and their variants) for several sets of parameters. Data taken from Tables 10 and 11.

## 4 Concluding Remarks

We have studied a simple adaptation to the facility location problem of Resende and Werneck’s multistart heuristic for the  $p$ -median problem [36]. The resulting algorithm has been shown to be highly effective in practice, finding near-optimal or optimal solutions of a large and heterogeneous set of instances from the literature. In terms of solution quality, the results either matched or surpassed those obtained by some of the best algorithms in the literature on every single class, which shows how robust our method is. The combination of fast local search and path-relinking within a multistart heuristic has proved once again to be a very effective means of finding near-optimal solutions for an NP-hard problem.

**Acknowledgements.** We thank two anonymous referees for their helpful comments, and D. Ghosh for providing us with his series of instances.

## References

- [1] S. Ahn, C. Cooper, G. Cornuéjols, and A. M. Frieze. Probabilistic analysis of a relaxation for the  $k$ -median problem. *Mathematics of Operations Research*, 13:1–31, 1998.
- [2] M. L. Alves and M. T. Almeida. Simulated annealing algorithm for the simple plant location problem: A computational study. *Revista Investigação Operacional*, 12, 1992.
- [3] F. Barahona and F. Chudak. Near-optimal solutions to large scale facility location problems. Technical Report RC21606, IBM, Yorktown Heights, NY, USA, 1999.
- [4] J. Beasley. OR-Library: Distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41:1069–1072, 1990. <http://mscmga.ms.ic.ac.uk/info.html>.
- [5] J. E. Beasley. Lagrangean heuristics for location problems. *European Journal of Operational Research*, 65:383–399, 1993.
- [6] O. Bilde and J. Krarup. Sharp lower bounds and efficient algorithms for the Simple Plant Location Problem. *Annals of Discrete Mathematics*, 1:79–97, 1977.
- [7] A. R. Conn and G. Cornuéjols. A projection method for the uncapacitated facility location problem. *Mathematical Programming*, 46:273–298, 1990.
- [8] G. Cornuéjols, G. L. Nemhauser, and L. A. Wolsey. The uncapacitated facility location problem. In P. B. Mirchandani and R. L. Francis, editors, *Discrete Location Theory*, pages 119–171. Wiley-Interscience, New York, 1990.
- [9] J. J. Dongarra. Performance of various computers using standard linear equations software. Technical Report CS-89-85, Computer Science Department, University of Tennessee, 2003.
- [10] D. Erlenkotter. A dual-based procedure for uncapacitated facility location. *Operations Research*, 26:992–1009, 1978.
- [11] R. D. Galvão and L. A. Raggi. A method for solving to optimality uncapacitated facility location problems. *Annals of Operations Research*, 18:225–244, 1989.
- [12] D. Ghosh, 2003. Personal communication.
- [13] D. Ghosh. Neighborhood search heuristics for the uncapacitated facility location problem. *European Journal of Operational Research*, 150:150–162, 2003.

- [14] F. Glover. Tabu search and adaptive memory programming: Advances, applications and challenges. In R. S. Barr, R. V. Helgason, and J. L. Kennington, editors, *Interfaces in Computer Science and Operations Research*, pages 1–75. Kluwer, 1996.
- [15] F. Glover, M. Laguna, and R. Martí. Fundamentals of scatter search and path relinking. *Control and Cybernetics*, 39:653–684, 2000.
- [16] S. Guha and S. Khuller. Greedy strikes back: Improved facility location algorithms. *Journal of Algorithms*, 31:228–248, 1999.
- [17] M. Guignard. A Lagrangean dual ascent algorithm for simple plant location problems. *European Journal of Operations Research*, 35:193–200, 1988.
- [18] M. Hoeyer. Performance of heuristic and approximation algorithms for the uncapacitated facility location problem. Research Report MPI-I-2002-1-005, Max-Planck-Institut für Informatik, 2002.
- [19] M. Hoeyer. Uflib, 2002. <http://www.mpi-sb.mpg.de/units/ag1/projects/-benchmarks/UflLib/>.
- [20] M. Hoeyer. Experimental comparison of heuristic and approximation algorithms for uncapacitated facility location. In *Proceedings of the Second International Workshop on Experimental and Efficient Algorithms (WEA 2003)*, number 2647 in Lecture Notes in Computer Science, pages 165–178, 2003.
- [21] K. Jain, M. Mahdian, E. Markakis, A. Saberi, and V. V. Vazirani. Greedy facility location algorithms analyzed using dual fitting with factor-revealing LP. *Journal of the ACM*, 50(6):795–824, 2003.
- [22] K. Jain, M. Mahdian, and A. Saberi. A new greedy approach for facility location problems. In *Proc. ACM Symposium on the Theory of Computing (STOC’02)*, 2002.
- [23] Y. Kochetov. Benchmarks library, 2003. <http://www.math.nsc.ru/LBRT/-k5/Kochetov/bench.html>.
- [24] Y. Kochetov and D. Ivanenko. Computationally difficult instances for the uncapacitated facility location problem. In *Proceedings of the 5th Metaheuristics International Conference (MIC2003)*, pages 41:1–41:6, 2003.
- [25] M. Körkel. On the exact solution of large-scale simple plant location problems. *European Journal of Operational Research*, 39:157–173, 1989.
- [26] J. Kratica, D. Tosić, V. Filipović, and I. Ljubić. Solving the simple plant location problem by genetic algorithm. *RAIRO Operations Research*, 35:127–142, 2001.
- [27] A. A. Kuehn and M. J. Hamburger. A heuristic program for locating warehouses. *Management Science*, 9(4):643–666, 1963.

- [28] M. Laguna and R. Martí. GRASP and path relinking for 2-layer straight line crossing minimization. *INFORMS Journal on Computing*, 11:44–52, 1999.
- [29] M. Mahdian, Y. Ye, and J. Zhang. Improved approximation algorithms for metric facility location problems. In *Proceedings of the 5th APPROX Conference*, volume 2462 of *Lecture Notes in Computer Science*, pages 229–242, 2002.
- [30] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.
- [31] L. Michel and P. Van Hentenryck. A simple tabu search for warehouse location. *European Journal on Operations Research*, 157(3):576–591, 2003.
- [32] M. G. C. Resende and C. C. Ribeiro. Greedy randomized adaptive search procedures. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, pages 219–249. Kluwer, 2003.
- [33] M. G. C. Resende and C. C. Ribeiro. GRASP with path-relinking: Recent advances and applications. In T. Ibaraki, K. Nonobe, and M. Yagiura, editors, *Metaheuristics: Progress as Real Problem Solvers*, pages 29–63. Springer, 2005.
- [34] M. G. C. Resende and R. F. Werneck. A fast swap-based local search procedure for location problems. Technical Report TD-5R3KBH, AT&T Labs Research, 2003.
- [35] M. G. C. Resende and R. F. Werneck. On the implementation of a swap-based local search procedure for the  $p$ -median problem. In R. E. Ladner, editor, *Proceedings of the Fifth Workshop on Algorithm Engineering and Experiments (ALENEX'03)*, pages 119–127. SIAM, 2003.
- [36] M. G. C. Resende and R. F. Werneck. A hybrid heuristic for the  $p$ -median problem. *Journal of Heuristics*, 10(1):59–88, 2004.
- [37] D. B. Shmoys, É. Tardos, and K. Aardal. Approximation algorithms for facility location problems. In *Proceedings of the 29th ACM Symposium on Theory of Computing*, pages 265–274, 1997.
- [38] M. Sun. A tabu search procedure for the uncapacitated facility location problem. In C. Rego and B. Alidaee, editors, *Adaptive Memory and Evolution: Tabu Search and Scatter Search*. Kluwer, 2004. To appear.
- [39] M. Sun. Solving uncapacitated facility location problems using tabu search. Submitted to *Journal of Heuristics*, 2005.