A Hybrid Multistart Heuristic for the Uncapacitated Facility Location Problem *

Mauricio G. C. Resende[†] Renato F. Werneck[‡]

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 of 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:

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

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 Technical Report TD-5RELRR. September 15, 2003.

†AT&T Labs Research, 180 Park Avenue, Florham Park, NJ 07932. Electronic address: mgcr@research.att.com.

‡Department of Computer Science, Princeton University, 35 Olden Street, Princeton, NJ 08544. Electronic address: rwerneck@cs.princeton.edu. 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, 24]), but the NP-hard nature of the problem 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 are symmetric and obey the triangle inequality). In 1997, Shmoys et al. [35] presented the first polynomial-time algorithm with a constant approximation factor (approximately 3.16). Several improved algorithms have been developed since then, with some of the latest [21, 22, 28] 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, since the pioneering work of Kuehn and Hamburger [26]. Since then, more sophisticated metaheuristics have been applied, such as simulated annealing [2], genetic algorithms [25], tabu search [13, 30], and the so-called "complete local search with memory" [13]. Dual-based methods, such as Erlenkotter's dual ascent [10], Guignard's Lagragean 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 heuristic is presented by Hoefer 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. [28]; swap-based local search; Michel and Van Hentenryck's tabu search [30]; and the volume algorithm [3]. Hoefer'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 p-median problem in [33]. A series of minor adaptations is enough to build a very robust algorithm, 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 [33].

2 The Algorithm

In [33], 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 is a post-optimization phase, in which the solutions in the pool of elite solutions are combined among themselves 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 makes 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 [33].

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 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 [33], 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. The constructive algorithm used in each iteration i is random: each facility is opened with probability q_i; if no facility whatsoever is opened, a single one is chosen at random. We define q_1 to be 0.5: in the first iteration, approximately half of the facilities will be opened. For $i > 1$, we define $q_i = \bar{k_i}/m$, where $\bar{k_i}$ is the average number of facilities in the solutions found (after local search) in the first $i - 1$ iterations. The idea here is to use good known solutions to estimate a reasonable number of facilities to open.

Local search. The local search used in [33] 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 (as well as 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*).

The actual implementation of the local search is essentially the same as in [33] and described in detail in [34]. Two modifications must be made to the original algorithm. First, we have to take into account setup costs (inexistent in the p-median problem). This can be handled by properly initializing *save* and *loss* (these are auxiliary data structures, as explained in [34]). Second, we must support individual insertions and deletions, which can be trivially accomplished because the profits associated with these moves are already represented by *save* and *loss*, respectively. The details of the transformation are detailed in [32].

Path-relinking. Path-relinking is an intensification procedure originally devised for scatter search and tabu search [14, 15, 27], but often used with other methods, such as GRASP [31]. In this paper, we apply the variant described in [33]. 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 [33]. They are based on the notion of *symmetric difference* between two solutions S_a and S_b, defined as $|S_a \setminus S_b| + |S_b \setminus S_a|$.[1] 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.

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.

Post-optimization. Once the multistart phase is over, all elite solutions are combined with one another, also with path-relinking. 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 [33], 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).

[1] 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.

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* [29]. The source code for the algorithm is available from the authors upon request.

The algorithm was tested on all classes from the UflLib [20] and on class GHOSH, described in [13]. In every case, the number of users and potential facilities is the same. 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]. All instances have known optima (by construction), but are meant to be challenging for algorithms based on local search. There are two values of n, 133 and 307, each with 40 instances. They determine two subclasses, to which we refer as FPP11 and FPP17, respectively.[2] Each distance matrix has only $n+1$ non-infinite values, all taken from the set $\{0, 1, 2, 3, 4\}$. The setup cost is always 3000.

- GAP: Instances designed by Kochetov [23] to have large duality gaps, often 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. There are three subclasses (GAPA, GAPB, and GAPC), each with 30 instances. Each customer in GAPA is covered by 10 facilities; each facility in GAPB covers exactly 10 customers; subclass GAPC combines both constraints: each customer is covered by 10 facilities, and each facility covers 10 customers.

- GHOSH: Class created by Ghosh in [13], following the guidelines set up by Körkel in [24]. 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.[3] Connection costs are integers taken uniformly at random

[2] As explained in [23], the values of n take the form $k^2 + k + 1$, with k being 11 or 17.

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

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 service 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]. The number of users is either 50, 70, 100, 150, or 200. There are 50 instances in total, 10 for each value of n. 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 [25]. 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 opening 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 UflLib website [19], with the exception of those in class GHOSH, created with a generator kindly provided by D. Ghosh [12]. Recall that five of these eight classes were used in Hoefer's comparative analysis [18, 20]: BK, GR, M*, MED, and ORLIB.

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 FPP, GAP, BK, GR, and ORLIB.

We used the best upper bounds shown in [19] for MED and M* (upper bounds that are not proved optimal were obtained either by tabu search or 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 in this case.

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	0.001	0.28
FPP	27.999	7.36
GAP	5.935	1.63
GHOSH	-0.039	30.66
GR	0.000	0.31
M*	0.000	7.45
MED	-0.392	284.88
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 UflLib, 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	AVERAGE	TIME (S)
mo1	100	1305.95	0.961
mo2	100	1432.36	0.997
mo3	100	1516.77	0.912
mo4	100	1442.24	0.877
mo5	100	1408.77	0.814
mp1	200	2686.48	3.602
mp2	200	2904.86	4.017
mp3	200	2623.71	3.404
mp4	200	2938.75	3.772
mp5	200	2932.33	4.052
mq1	300	4091.01	8.547
mq2	300	4028.33	7.332
mq3	300	4275.43	9.113
mq4	300	4235.15	9.417
mq5	300	4080.74	10.239
mr1	500	2608.15	25.009
mr2	500	2654.74	25.486
mr3	500	2788.25	24.762
mr4	500	2756.04	26.291
mr5	500	2505.05	24.993
ms1	1000	5283.76	103.334
mt1	2000	10069.80	568.605

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	0.062
cap102	50	854704.20	0.060
cap103	50	893782.11	0.070
cap104	50	928941.75	0.079
cap131	50	793439.56	0.110
cap132	50	851495.32	0.097
cap133	50	893076.71	0.128
cap134	50	928941.75	0.138
cap71	50	932615.75	0.036
cap72	50	977799.40	0.041
cap73	50	1010641.45	0.059
cap74	50	1034976.97	0.051
capa	1000	17156454.48	7.094
capb	1000	12979071.58	5.982
capc	1000	11505594.33	5.788

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 [30], 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 comparred with both lower and upper bounds). On average, HYBRID found solutions that are at least 0.15% better than previous bounds, 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 within 0.178% or less from optimality, possibly better (depending on how good the lower bounds are).

Classes BK and GR are larger (they have 220 and 50 instances, respectively), so we aggregate the data 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 5-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

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	28.6
med0500-100	326754	328540	326796.2	0.013	-0.531	24.7
med0500-1000	99099	99325	99169.0	0.071	-0.157	18.8
med1000-10	1432737	1439285	1434201.1	0.102	-0.353	136.7
med1000-100	607591	609578	607890.1	0.049	-0.277	97.3
med1000-1000	220479	221736	220560.0	0.037	-0.531	117.0
med1500-10	1997302	2005877	2000851.4	0.178	-0.251	374.9
med1500-100	866231	870182	866458.2	0.026	-0.428	269.3
med1500-1000	334859	336263	334968.1	0.033	-0.385	299.5
med2000-10	2556794	2570231	2558125.5	0.052	-0.471	591.2
med2000-100	1122455	1128392	1122837.5	0.034	-0.492	464.8
med2000-1000	437553	439597	437693.8	0.032	-0.433	540.3
med2500-10	3095135	3114458	3100447.3	0.172	-0.450	1171.6
med2500-100	1346924	1352322	1347620.1	0.052	-0.348	759.0
med2500-1000	534147	536546	534430.5	0.053	-0.394	908.3
med3000-10	3567125	3586599	3570779.2	0.102	-0.441	1304.5
med3000-100	1600551	1611186	1602386.3	0.115	-0.546	1382.9
med3000-1000	643265	645680	643557.6	0.045	-0.329	1385.5

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	0.305
C	100	0.0080	0.441
D01	80	0.0001	0.219
D02	80	0.0000	0.201
D03	80	0.0000	0.196
D04	80	0.0000	0.168
D05	80	0.0000	0.163
D06	80	0.0000	0.187
D07	80	0.0000	0.172
D08	80	0.0000	0.166
D09	80	0.0000	0.173
D10	80	0.0000	0.167
E01	100	0.0000	0.469
E02	100	0.0055	0.577
E03	100	0.0188	0.473
E04	100	0.0000	0.455
E05	100	0.0000	0.369
E06	100	0.0000	0.396
E07	100	0.0000	0.409
E08	100	0.0000	0.413
E09	100	0.0000	0.351
E10	100	0.0000	0.355

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	0.100
70	0.162
100	0.304
150	0.585
200	1.048

Intel Mobile Celeron running at 650 MHz).[4]

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 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	257922.2	-0.022	5.07
GA250B	250	276184.2	6.5	276053.2	-0.047	7.77
GA250C	250	333058.4	17.3	332897.2	-0.048	7.11
GA500A	500	511251.6	18.1	511148.5	-0.020	33.62
GA500B	500	538144.0	6.4	537876.7	-0.050	42.74
GA500C	500	621881.8	24.7	621470.8	-0.066	53.88
GA750A	750	763840.4	213.3	763737.9	-0.013	97.99
GA750B	750	796754.2	71.4	796385.7	-0.046	115.36
GA750C	750	900349.8	146.5	900197.0	-0.017	125.21
GS250A	250	257832.6	207.1	257806.9	-0.010	4.97
GS250B	250	276185.2	79.2	276035.2	-0.054	7.43
GS250C	250	333671.6	134.6	333671.6	0.000	7.95
GS500A	500	511383.6	824.3	511198.2	-0.036	40.66
GS500B	500	538480.4	409.4	537920.0	-0.104	45.14
GS500C	500	621107.2	347.4	621059.2	-0.008	47.14
GS750A	750	763831.2	843.2	763711.5	-0.016	91.84
GS750B	750	796919.0	396.0	796603.0	-0.040	107.10
GS750C	750	901158.4	499.7	900197.2	-0.107	118.37

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.

[4] From [9], we can infer that these processors 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 8: Results for FPP and GAP instances: average percent errors (with respect to the optimal solutions) and average running times for each subclass.

SUBCLASS	n	AVG%ERR	TIME (S)
GAPA	100	5.13	1.38
GAPB	100	5.93	1.82
GAPC	100	6.74	1.88
FPP11	133	7.16	2.63
FPP17	307	48.84	22.24

With that in mind, we compare the results obtained by our algorithm with those obtained by Michel and Van Hentenryck's tabu search algorithm [30], the best among the algorithms tested in [18], based on experimental evidence. We refer to this method as TABU.

To ensure that running times are comparable, we downloaded the source code for an implementation of TABU from the UflLib, compiled it with the same parameters used for HYBRID, and ran it on the same machine. Since it 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 [30], the algorithm was run with 500 iterations. However, with these many iterations TABUis 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 TABU (with 500 iterations) and HYBRID (with 8 iterations and 5 elite solutions).

CLASS	HYBRID		TABU	
	AVG%DEV	TIME (S)	AVG%DEV	TIME (S)
BK	0.028	0.082	0.076	0.152
FPP	66.491	1.730	97.061	0.604
GAP	9.502	0.369	16.499	0.244
GHOSH	-0.032	7.887	0.002	4.621
GR	0.000	0.087	0.103	0.158
M*	0.004	2.087	0.011	1.615
MED	-0.369	75.231	0.073	69.552
ORLIB	0.000	0.046	0.024	0.155

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).

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 0.369% 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, solutions found by our algorithm for GAP were almost 10% off optimality, but TABU did even worse: the difference was greater than 16%. The hardest class of all seems to be FPP: the average deviation from optimality was 66% for our algorithm, and almost 100% for TABU. Even though HYBRID does slightly better 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 that is indeed the case, we performed longer runs of both algorithms. For tabu search, we varied the number of iterations (the only parameter); the values used were 500 (the original value), 1000, 2000, 4000, 8000, 16000, 32000, and 64000. HYBRID has two input parameters (number of iterations and number of elite solutions). We tested the following pairs: 4:3, 8:5, 16:7, 32:10 (the original parameters), 64:14, 128:20, 256:28, and 512:40.[5] The results are summarized in Tables 10 (for HYBRID) and 11 (for TABU), which contains the average error and the mean running time for each choice of parameters.

The same information is represented graphically on Figures 3 (for FPP) and 2 (GAP). 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 appropriate graph, and the points were then linearly interpolated.

Within the same time frame, the graphs show HYBRID has superior results when compared with TABU. Not only does it obtain better results with small

[5]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.

Table 10: HYBRID results on hard classes. Average errors and mean running times.

CLASS	ITER.	ELITE	AVG%ERR	TIME (S)
FPP	4	3	82.832	0.58
	8	5	65.265	1.59
	16	7	48.413	3.49
	32	10	27.610	7.15
	64	14	13.279	13.79
	128	20	2.307	25.33
	256	28	0.018	48.17
	512	40	0.009	93.59
GAP	4	3	12.961	0.14
	8	5	9.543	0.37
	16	7	7.407	0.78
	32	10	5.932	1.63
	64	14	4.561	3.23
	128	20	3.541	6.49
	256	28	2.700	12.54
	512	40	1.685	24.69

Table 11: TABU results on hard class. Average errors and mean running times.

CLASS	ITER.	AVG%ERR	TIME (S)
FPP	500	97.06	0.60
	1000	94.22	1.04
	2000	91.14	1.97
	4000	86.81	3.86
	8000	83.67	7.34
	16000	79.32	14.34
	32000	75.16	27.71
	64000	71.15	52.60
GAP	500	16.50	0.25
	1000	14.38	0.46
	2000	12.40	0.88
	4000	10.62	1.68
	8000	8.94	3.27
	16000	7.72	6.24
	32000	7.02	11.85
	64000	6.35	22.62

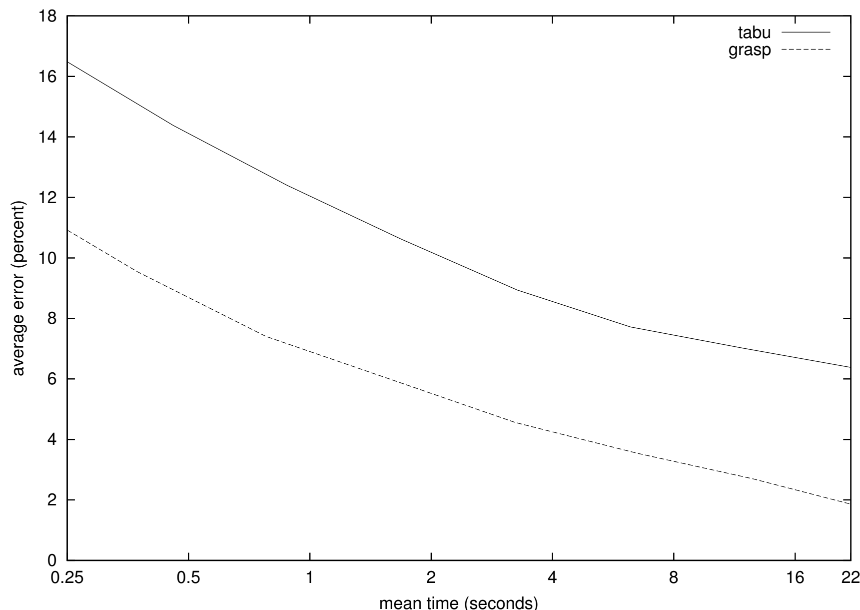

Figure 2: GAP class. Mean percent deviation obtained by HYBRID and TABU for several sets of parameters. Data taken from Tables 10 and 11.

running times (less than a second), but it also makes better use of any additional time allowed. Take class GAP: within 0.25 second, our algorithm can obtain solutions that are more than 10% off optimality (on average); in 20 seconds, the error is down to 2%. TABU, on the other had, varies from 16% to 6%.

But the most remarkable differences between the algorithms were observed on class FPP. If given less than one second, both do very badly: HYBRID finds solutions that are almost 80% away from optimality on average; TABU is even worse, with 90%. However, once the programs are allowed longer runs, HYBRID improves at a much faster rate than TABU. Within 50 seconds, HYBRID already finds near-optimal solutions on all cases (the average error is below 0.02%), whereas solutions found by TABU are still more than 70% off optimality on average.

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 [33]. 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

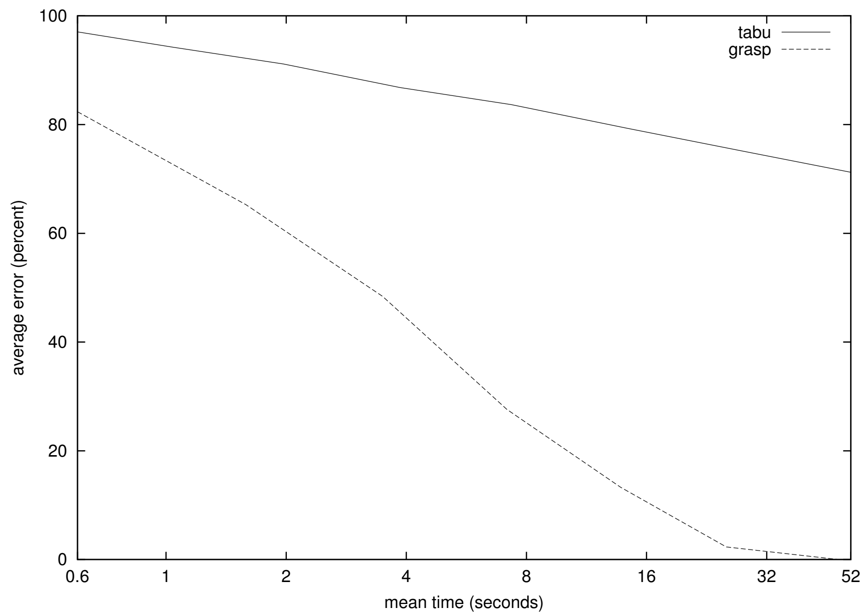

Figure 3: FPP class. Average percentage deviation obtained by HYBRID and TABU for several sets of parameters. Data taken from Tables 10 and 11.

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.

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 Investigaćão 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. E. Beasley. A note on solving large p-median problems. *European Journal of Operational Research*, 21:270–273, 1985.

[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. Courneéjols. A projection method for the uncapacited 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. Greed 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. Hoefer. 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. Hoefer. Ufllib, 2002. *http://www.mpi-sb.mpg.de/units/ag1/projects/-benchmarks/UflLib/*.

[20] M. Hoefer. Experimental comparision 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*, 2003. To appear.

[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] M. Körkel. On the exact solution of large-scale simple plant location problems. *European Journal of Operational Research*, 39:157–173, 1989.

[25] J. Kratica, D. Tosic, V. Filipovic, and I. Ljubic. Solving the simple plant location problem by genetic algorithm. *RAIRO Operations Research*, 35:127–142, 2001.

[26] A. A. Kuehn and M. J. Hamburger. A heuristic program for locating warehouses. *Management Science*, 9(4):643–666, 1963.

[27] M. Laguna and R. Martí. GRASP and path relinking for 2-layer straight line crossing minimization. *INFORMS Journal on Computing*, 11:44–52, 1999.

[28] 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.

[29] 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.

[30] L. Michel and P. Van Hentenryck. A simple tabu search for warehouse location. *European Journal on Operations Research*, 2003. To appear.

[31] 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.

[32] 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.

[33] M. G. C. Resende and R. F. Werneck. A hybrid heuristic for the p-median problem. Technical Report TD-5NWRCR, AT&T Labs Research, 2003.

[34] 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.

[35] 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.