On the implementation of a swap-based local search procedure for the p-median problem

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

Abstract

We present a new implementation of a widely used swap-based local search procedure for the p-median problem. It produces the same output as the best implementation described in the literature and has the same worst-case complexity, but, through the use of extra memory, it can be significantly faster in practice: speedups of up to three orders of magnitude were observed.

1 Introduction

The p-median problem is defined as follows. Given a set F of m facilities, a set U of n users (or customers), a distance function $d : U \times F \to \mathcal{R}$, and a constant $p \leq m$, determine which p facilities to open so as to minimize the sum of the distances from each user to the closest open facility.

Being a well-known NP-complete problem [2], one is often compelled to resort to heuristics to deal with it in practice. Among the most widely used is the swap-based local search proposed by Teitz and Bart [10]. It has been applied on its own [8, 13] and as a key subroutine of more elaborate metaheuristics [3, 7, 9, 12]. The efficiency of the local search procedure is of utmost importance to the effectiveness of these methods. In this paper, we present a novel implementation of the local search procedure and compare it with the best alternative described in the literature, proposed by Whitaker in [13]. In practice, we were able to obtain significant (often asymptotic) gains.

This paper is organized as follows. In Section 2, we give a precise description of the local search procedure and a trivial implementation. In Section 3, we describe Whitaker's implementation. Our own implementation is described in Section 4. Experimental evidence to the efficiency of our method is presented in Section 5. Final remarks are presented in Section 6.

Notation and assumptions. Before proceeding with the algorithms themselves, let us establish some notation. As already mentioned, F is the set of potential facilities and U the set of users that must be served. The basic parameters of the problem are $n = |U|$, $m = |F|$, and p, the number of facilities to open. Although $1 \leq p \leq m$ by definition, we will ignore trivial cases and assume that $1 < p < m$ and that $p < n$ (if $p \geq n$, we just open the facility that is closest to each user). We assume nothing about the relationship between n and m.

In this paper, u denotes a generic user, and f a generic facility. The cost of serving u with f is $d(u, f)$, the distance between them. A *solution* S is any subset of F with p elements (representing the open facilities). Each user u must be assigned to the closest facility $f \in S$, the one that minimizes $d(u, f)$. This facility will be denoted by $\phi_1(u)$; similarly, the second closest facility to u in S will be denoted by $\phi_2(u)$. To simplify notation, we will abbreviate $d(u, \phi_1(u))$ as $d_1(u)$, and $d(u, \phi_2(u))$ as $d_2(u)$. We often deal specifically with a facility that is a candidate for insertion; it will be referred to as f_i (by definition $f_i \notin S$); similarly, a candidate for removal will be denoted by f_r ($f_r \in S$, also by definition).

Throughout this paper, we assume the *distance oracle* model, in which the distance between any customer and any facility can be found in $O(1)$ time. This is the case if there is a distance matrix, or if facilities and users are points on the plane, for instance. In this model, all values of ϕ_1 and ϕ_2 for a given solution S can be straighforwardly computed in $O(pn)$ total time.

2 The Swap-based Local Search

Introduced by Teitz and Bart in [10], the standard local search procedure for the p-median problem is based on swapping facilities. For each facility $f_i \notin S$, the procedure determines which facility $f_r \in S$ (if any) would improve the solution the most if f_i and f_r were interchanged (i.e., if f_i were inserted and f_r removed from the solution). If one such "improving" swap exists, it is performed, and the procedure is repeated from the

*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.

new solution. Otherwise we stop, having reached a *local minimum* (or *local optimum*). Our main concern here is the time it takes to run each iteration of the algorithm: given a solution S, how fast can we find a neighbor S'?

It is not hard to come up with an $O(pmn)$ implementation for this procedure. First, in $O(pn)$ time, determine the closest and second closest facilities for each user. Then, for each candidate pair (f_i, f_r), determine the profit that would be obtained by replacing f_r with f_i, using the following expression (recall that $d_1(u)$ and $d_2(u)$ represent distances from u to the closest and second closest facilities, respectively):

$$\begin{aligned} \mathit{profit}(f_i, f_r) &= \sum_{u:\phi_1(u)\neq f_r} \max\{0, [d_1(u) - d(u, f_i)]\} \\ &\quad - \sum_{u:\phi_1(u)=f_r} (\min\{d_2(u), d(u, f_i)\} - d_1(u)). \end{aligned}$$

The first summation accounts for users whose closest facility is not f_r; these will be reassigned to f_i only if that is profitable. The second summation refers to users originally assigned to f_r, which will be reassigned either to their original second closest facilities or to f_i, whichever is more advantageous. The entire expression can be computed in $O(n)$ time for each pair of facilities. Since there are $p(m-p) = O(pm)$ pairs to test, the whole procedure takes $O(pmn)$ time per iteration.

There are several papers in the literature that use this implementation, or avoid using the swap-based local search altogether, mentioning its intolerable running time [7, 9, 12]. All these methods would greatly benefit from Whitaker's implementation (or from ours, for that matter), described in the next section.

3 Whitaker's Implementation

In [13], Whitaker describes the so-called *fast interchange* heuristic, an efficient implementation of the local search procedure defined above. Even though it was published in 1983, Whitaker's implementation was not widely used until 1997, when Hansen and Mladenović [3] applied it as a subroutine of a Variable Neighborhood Search (VNS) procedure. Their implementation is based on Whitaker's, with a minor difference: Whitaker prefers a *first improvement* strategy (a swap is made as soon as a profitable one is found), while Hansen and Mladenović prefer *best improvement* (all swaps are evaluated and the most profitable executed). In our analysis, we assume best improvement is used.

The key aspect of this implementation is its ability to find in $\Theta(n)$ time the best possible candidate for removal, given a certain candidate for insertion. The pseudocode for a function that does that, adapted from [3], is presented in Figure 1.[1] Function `findOut` takes as input a candidate for insertion (f_i) and returns f_r, the most profitable facility to be swapped out, as well as the profit itself (*profit*).

In the code, w represents the total amount saved by reassigning users to f_i, independently of which facility is removed; it takes into account all users currently served by facilities that are farther than f_i. The loss due to the removal of f_r is represented by $v(f_r)$, which accounts for users that have f_r as their closest facility. Each such user will be reassigned (if f_r is removed) either to its original second closest facility or to f_i; the reassignment cost is computed in line 7. Line 10 determines the best facility to remove, the one for which $v(f_r)$ is minimum. The overall profit of the corresponding swap (considering also the gains in w) is computed in line 11.

Since this function runs in $O(n)$ time, it is now trivial to implement the swap-based local search procedure in $O(mn)$ time per iteration: simply call `findOut` once for each of the $m-p$ candidates for insertion and pick the most profitable one. If the best profit is positive, perform the swap, update the values of ϕ_1 and ϕ_2, and proceed to the next iteration. Updating ϕ_1 and ϕ_2 requires $O(pn)$ time in the worst case, but can be made faster in practice, as mentioned in [13]. Since our implementation uses the same technique, its description is deferred to the next section (Subsection 4.3.1).

4 An Alternative Implementation

Our implementation has some similarity with Whitaker's, in the sense that both methods perform the same basic operations. However, the order in which they are performed is different, and in our case partial results obtained are stored in auxiliary data structures. As we will see, this allows for the use of values computed in early iterations of the algorithm to speed up later ones.

4.1 Additional Structures. For each facility f_i not in S, let $\mathit{gain}(f_i)$ be the total amount saved when f_i is added to S (thus creating a new solution with $p+1$ facilities). The savings result from reassigning to f_i every customer whose current closest facility is farther than f_i itself:

$$\mathit{gain}(f_i) = \sum_{u\in U} \max\{0, d_1(u) - d(u, f_i)\}. \tag{4.1}$$

Similarly, for every $f_r \in S$, define $\mathit{loss}(f_r)$ as the increase in solution value resulting from the removal of f_r (with only $p-1$ facilities remaining). This is the

[1] Expressions of the form $a \xleftarrow{+} b$ in the code mean that the value of a is incremented by b units.

```
function findOut (S, f_i, φ_1, φ_2)
1    w ← 0;
2    forall (f ∈ S) do v(f) ← 0;
3    forall (u ∈ U) do
4        if (d(u, f_i) < d(u, φ_1(u)) then /* profit if f_i is close enough to u */
5            w +← [d(u, φ_1(u)) − d(u, f_i)];
6        else /* loss if facility that is closest to u is removed */
7            v(φ_1(u)) +← min{d(u, f_i), d(u, φ_2(u))} − d(u, φ_1(u));
8        endif
9    endforall
10   f_r ← argmin_{f∈S}{v(f)};
11   profit ← w − v(f_r);
12   return (f_r, profit);
end findOut
```

Figure 1: Function to determine, given a candidate for insertion (f_i), the best candidate for removal (f_r). Adapted from [3].

cost of transferring every customer assigned to f_r to its second closest facility:

$$loss(f_r) = \sum_{u:\phi_1(u)=f_r} [d_2(u) - d(u, f_r)]. \tag{4.2}$$

In the local search, we are interested in what happens when insertions and removals occur simultaneously. Let $profit(f_i, f_r)$ be the amount saved when f_i and f_r are swapped. We claim it can be expressed as

$$profit(f_i, f_r) = gain(f_i) - loss(f_r) + extra(f_i, f_r), \tag{4.3}$$

for a properly defined function $extra(f_i, f_r)$. Note that the profit will be negative if the neighboring solution is worse than S. To find the correct specification of *extra*, we observe that, for every customer u, one of the following cases must hold:

1. $\phi_1(u) \neq f_r$ (the customer was not assigned to f_r before the swap). We may save something by reassigning it to f_i, and the amount we save is included in $gain(f_i)$.
2. $\phi_1(u) = f_r$ (the customer was assigned to f_r before the swap). Three subscases present themselves:
   a. $d_1(u) \leq d_2(u) \leq d(u, f_i)$. Customer u should be assigned to $\phi_2(u)$, which was exactly the assumption made during the computation of $loss(f_r)$. The loss corresponding to this reassignment is therefore already taken care of.
   b. $d_1(u) \leq d(u, f_i) < d_2(u)$. Customer u should be reassigned to f_i, but during the computation of $loss(f_r)$ we assumed it would be reassigned to $\phi_2(u)$. This means that the estimate of the contribution of u to $loss(f_r)$ is overly pessimistic by $d_2(u) - d(u, f_i)$.
   c. $d(u, f_i) < d_1(u) \leq d_2(u)$. Customer u should be reassigned to f_i, with a profit of $d_1(u) - d(u, f_i)$ (correctly accounted for in the computation of $gain(f_i)$). However, the loss of $d_2(u) - d_1(u)$ predicted in the computation of $loss(f_i)$ will not occur.

The defintion of $extra(f_i, f_r)$ must handle cases (2*b*) and (2*c*) above, in which wrong predictions were made. Corrections can be expressed straightforwardly as individual summations, one for each case:

$$\begin{aligned} extra(f_i, f_r) \quad = \quad & \sum_{\substack{u:[\phi_1(u)=f_r]\wedge \\ [d_1(u)\leq d(u,f_i)<d_2(u)]}} [d_2(u) - d(u, f_i)] + \\ & + \sum_{\substack{u:[\phi_1(u)=f_r]\wedge \\ [d(u,f_i)<d_1(u)\leq d_2(u)]}} [d_2(u) - d(u, f_r)]. \end{aligned}$$

These summations can be merged into one:

$$extra(f_i, f_r) \quad = \sum_{\substack{u:[\phi_1(u)=f_r]\wedge \\ [d(u,f_i)<d_2(u)]}} [d_2(u) - \max\{d(u, f_i), d(u, f_r)\}]. \tag{4.4}$$

4.2 Local Search. Assume all values of *loss*, *gain*, and *extra* can be efficiently precalculated and stored in appropriate data structures (vectors for *loss* and *gain*,

```
function updateStructures (u, loss, gain, extra, φ1, φ2)
1     d1 ← d(u, φ1(u));
2     d2 ← d(u, φ2(u));
3     fr ← φ1(u);
4     loss(fr) +← (d2 − d1);
5     forall (fi ∉ S) do
6           if (d(u, fi) < d2) then
7                 gain(fi) +← max{0, d1 − d(u, fi)};
8                 extra(fi, fr) +← (d2 − max{d(u, fi), d(u, fr)});
9           endif
10    endfor
end updateStructures
```

Figure 2: Pseudocode for updating arrays in the local search procedure

a matrix for *extra*). Then, we can find the best swap in $O(pm)$ time by computing the profits associated with every pair of candidates using Equation 4.3.

To develop an efficient method to precompute *gain*, *loss*, and *extra*, we note that every entry in these structures is a summation over some subset of users (see Equations 4.1, 4.2, and 4.4). Moreover, the contribution of each user can be computed separately. Function `updateStructures`, shown in Figure 2, does exactly that. It takes as input a user u and its closest facilities (given by ϕ_1 and ϕ_2), and updates the contents of *loss*, *gain*, and *extra*. To compute all three structures from scratch, we just need to reset them (set all positions to zero) and call `updateStructures` once for each user. Together, these n calls perform precisely the summations defined in Equations 4.1, 4.2, and 4.4.

We now have all the elements necessary to build a full local search algorithm in $O(mn)$ time. In $O(pn)$ time, compute ϕ_1 and ϕ_2 for all users. In $O(pm)$ time, reset *loss*, *gain*, and *extra*. With n calls to `updateStructures`, each made in $O(m)$ time, determine their actual values. Finally, in $O(pm)$ time, find the best swap.

4.3 Acceleration. So far, our implementation seems to be merely a more complicated alternative to Whitaker's; after all, both have the same worst-case complexity. Furthermore, our implementation has the clear disadvantage of requiring an $O(pm)$-sized matrix, while $\Theta(n)$ memory positions are enough in Whitaker's. The extra memory, however, allows for significant accelerations, as this section will show.

When a certain facility f_r is replaced by a new facility f_i, values in *gain*, *loss*, *extra*, ϕ_1, and ϕ_2 become inaccurate. A straighforward way to update them for the next local search iteration is to recompute ϕ_1 and ϕ_2, reset the other arrays, and then call `updateStructures` again for all users. But we can potentially do better than that. The actions performed by `updateStructures` depend only on u, $\phi_1(u)$, and $\phi_2(u)$; no value is read from other structures. Therefore, if $\phi_1(u)$ and $\phi_2(u)$ do not change from one iteration to another, there is no need to call `updateStructures` again for u.

Given that, we consider a user u to be *affected* if there is a change in either $\phi_1(u)$ or $\phi_2(u)$ (or both) after a swap is made. Sufficient conditions for u to be affected after a swap between f_i and f_r are: (1) one of the two closest facilities is f_r itself; or (2) the new facility is closer to u than the original $\phi_2(u)$ is. Contributions to *loss*, *gain*, and *extra* need to be updated only for affected users. If the number of affected users is small (it often is) significant gains can be obtained.

Note, however, that we cannot simply call `updateStructures` for these users, since this function simply adds new contributions. Previous contributions must be subtracted before new additions are made. We therefore need a function similar to `updateStructures`, with subtractions instead of additions.[2] This function (call it `undoUpdateStructures`) must be called for all affected users *before* ϕ_1 and ϕ_2 are recomputed.

Figure 3 contains the pseudocode for the entire local search procedure, already taking into account the observations just made. Apart from the functions just discussed, three others appear in the code. The first, `resetStructures`, just sets all entries in the auxiliary structures to zero. The second, `findBestNeighbor`, runs through these structures and finds the most prof-

[2] This function is identical to the one shown in Figure 2, with all occurrences of $+\leftarrow$ replaced with $-\leftarrow$: instead of incrementing values, we decrement them.

```
procedure localSearch (S, φ1, φ2)
1     A ← U; /* A is the set of affected users */
2     resetStructures (gain, loss, extra);
3     while (TRUE) do
4           forall (u ∈ A) do updateStructures (u, gain, loss, extra, φ1, φ2);
5           (fr, fi, profit) ←findBestNeighbor (gain, loss, extra);
6           if (profit ≤ 0) break; /* if there's no improvement, we're done */
7           A ← ∅;
8           forall (u ∈ U) do /* find out which users will be affected */
9                 if ((φ1(u) = fr) or (φ2(u) = fr) or (d(u, fi) < d(u, φ2(u)))) then
10                      A ← A ∪ {u}
11                endif
12          endforall;
13          forall (u ∈ A) do undoUpdateStructures (u, gain, loss, extra, φ1, φ2);
14          insert(S, fi);
15          remove(S, fr);
16          updateClosest(S, fi, fr, φ1, φ2);
17    endwhile
end localSearch
```

Figure 3: Pseudocode for the local search procedure

itable swap using Equation 4.3. It returns which facility to remove (f_r), the one to replace it (f_i), and the profit itself (*profit*). The third function is `updateClosest`, which updates ϕ_1 and ϕ_2, possibly using the fact that the facility recently opened was f_i and the one closed was f_r.

The pseudocode reveals three potential bottlenecks of the algorithm: updating the auxiliarly data structures (*loss*, *gain*, and *extra*), updating closeness information, and finding the best neighbor once the updates are done. We now analyze each of these in turn.

4.3.1 Closeness. Updating closeness information, in our experience, has proven to be a relatively cheap operation. Deciding whether the newly inserted facility f_i becomes either the closest or the second closest facility to each user is trivial and can be done in $O(n)$ total time. A more costly operation is finding a new second closest facility for customers who had f_r (the facility removed) as either the closest or the second closest element. Updating each of these users requires $O(p)$ time, but since there usually are few of them, the total time spent tends to be small fraction of the entire local search procedure.

One should also note that, in some settings, finding the set of closest and second closest elements from scratch is itself a cheap operation. For example, in the graph setting, where distances between customers and facilities are given by shortest paths on an underlying graph, this can be accomplished in $\tilde{O}(|E|)$ time [11], where $|E|$ is the number of edges in the graph. For experiments in this paper, however, specialized routines were not implemented; we always assume arbitrary distance matrices.

4.3.2 Best Neighbor. The number of potential swaps in a given solution is $p(m-p)$. The straighforward way to find the most profitable is to compute $profit(f_i, f_r)$ (as defined by Equation 4.3) for all pairs and pick the best, which requires $\Theta(pm)$ operations. In practice, however, the best move can be found in less time. As defined, $extra(f_i, f_r)$ can be interpreted as a measure of the interaction between the neighborhoods of f_r and f_i. After all, as Equation 4.4 shows, only users that have f_r as their current closest facility and are also close to f_i contribute to $extra(f_i, f_r)$. In particular, if there are no users in this situation, $extra(f_i, f_r)$ will be zero. It turns out that, in practice, this occurs rather frequently, especially for larger values of p, when the average number of vertices assigned to each f_r is relatively small.

Therefore, instead of storing *extra* as a full matrix, one may consider a sparse representation in which only nonzero elements are explicit: each row becomes a linked list sorted by column number. A drawback of the sparse representation (in general) is the impossibility to make random accesses in $O(1)$ time. Fortunately, for our purposes, this is not necessary; `updateStructures`, `undoUpdateStructures`, and `bestNeighbor` (the only functions that access the matrix) can be implemented

so as to go through each row sequentially.

With the sparse matrix representation, one can implement `bestNeighbor` as follows. First, determine the facility f_i that maximizes $gain(f_i)$ and the facility f_r that minimizes $loss(f_r)$. Since all values in *extra* are nonnegative, this pair is at least as profitable as any pair $(f_{i'}, f_{r'})$ for which $extra(f_{i'}, f_{r'})$ is zero. Then, compute the exact profits (given by Equation 4.3) for all nonzero elements in *extra*. The whole procedure takes $O(m + \lambda pm)$ time, where λ is the fraction of pairs whose *extra* value is nonzero. This tends to be smaller as p increases. An interesting side-effect of using sparse matrices is that they often need significantly less memory than the standard full matrix representation.

4.3.3 Updates. As we have seen, keeping track of affected users can reduce the number of calls to `updateStructures`. We now study how to reduce the time spent in each of these calls.

Consider the pseudocode in Figure 2. Line 5 represents a loop through all facilities not in the solution, but line 6 shows that we can actually restrict ourselves to facilities whose distance to u (the candidate customer) is no greater than d_2 (the distance from u to its second closest facility). This may be a relatively small subset of the facilities, especially when p is large. This suggests a preprocessing step that builds, for each user u, a list with all facilities sorted in increasing order by their distance to u. During the local search, whenever we need the set of facilities whose distance to u is less than d_2, we just take the appropriate prefix of the precomputed list — potentially much smaller than m.

Building these lists takes $O(nm \log m)$ time, but it is done only once, not in every iteration of the local search procedure. This is true even if local search is applied several times within a metaheuristic (as in [3, 9], for instance): we still need to perform the preprocessing step only once.

A more serious drawback of this approach is memory usage. Keeping n lists of size m requires $\Theta(mn)$ memory positions, which may be prohibitive. On the other hand, one should expect to need only small prefixes most of the time. Therefore, instead of keeping the whole list in memory, it might be good enough to keep only prefixes. The list would then be used as a cache: if d_2 is small enough, we just take a prefix of the candidate list; if it is larger than the largest element represented, we look at all possible neighbors.

5 Empirical Analysis

5.1 Instances and Methodology. We tested our algorithm on three classes of problems. Two of them, TSP and ORLIB, are commonly studied in the literature for the p-median problem. The third, RW, is introduced here as a particularly hard case for our method.

Class TSP corresponds to three sets of points on the plane (with cardinality 1400, 3038, and 5934), originally used in the context of the traveling salesman problem [6]. In the case of the p-median problem, each point is both a user to be served and a potential facility, and distances are Euclidean. Following [4], we tested several values of p for each instance, ranging from 10 to approximately $n/3$.

Class ORLIB, originally introduced in [1], contains 40 graphs with 100 to 900 nodes, each with a suggested value of p (ranging from 5 to 200). Each node is both a user and a potential facility, and distances are given by shortest paths in the graph. All-pairs shortest paths are computed in advance for all methods tested, as it is usually done in the literature [3, 4].

Each instance in class RW is a square matrix in which entry (u, f) (an integer taken uniformly at random from the interval $[1, n]$) represents the cost of assigning user u to facility f. Four values of n were tested (100, 250, 500, and 1000), each with values of p ranging from 10 to $n/2$, totaling 27 combinations.[3] The program that created these instances (using the random number generator by Matsumoto and Nishimura [5]) is available from the authors upon request.

All tests were performed on an SGI Challenge with 28 196-MHz MIPS R10000 processors (with each execution of the program limited to one processor) and 7.6 GB of memory. All algorithms were coded in C++ and compiled with the SGI MIPSpro C++ compiler (v. 7.30) with flags `-O3 -OPT:Olimit=6586`. All running times shown in this paper are CPU times, measured with the `getrusage` function, whose precision is 1/60 second. In some cases, actual running times were too small for this precision; therefore, each algorithm was repeatedly run for at least 5 seconds; overall times were measured, and averages reported here.

For each instance tested, all methods were applied to the same initial solution, obtained by a greedy algorithm [13]: starting from an empty solution, we insert one facility at a time, always picking the one that reduces the solution cost the most. Running times mentioned in this paper refer to the local search only, they do not include the construction of the initial solution.

5.2 Results. This section presents an experimental comparison between several variants of our implemen-

[3] More precisely: for $n = 100$, we used $p = 10$, 20, 30, 40, and 50; for $n = 250$, $p = 10$, 25, 50, 75, 100, and 125; for $n = 500$, $p = 10$, 25, 50, 100, 150, 200, and 250; and for $n = 1000$, $p = 10$, 25, 50, 75, 100, 200, 300, 400, and 500.

tation and Whitaker's method, which will be referred to here as FI (*fast interchange*). We implemented FI based on the pseudocode in [3] (obtaining comparable running times); the key function is presented here in Figure 1. The same routine for updating closeness information (described in Section 4.3.1) was used for all methods (including FI).

We start with the most basic version of our implementation, in which *extra* is represented as a full (non-sparse) matrix. This version (called FM, for *full matrix*) incorporates some acceleration, since calls to `updateStructures` are limited to affected users only. However, it does *not* include the accelerations suggested in Sections 4.3.2 (sparse matrix) and 4.3.3 (preprocessing). For each instance tested, we computed the speedup obtained by our method when compared to FI, i.e., the ratio between the running times of FI and FM. Table 1 shows the best, (geometric) mean, and worst speedups thus obtained considering all instances in each class.[4] Values larger than one favor our method, FM.

Table 1: Speedup obtained by FM (full matrix, no preprocessing) over Whitaker's FI.

CLASS	BEST	MEAN	WORST
ORLIB	12.72	3.01	0.84
RW	12.42	4.14	0.88
TSP	31.14	11.68	1.85

The table shows that even the basic acceleration scheme achieves speedups of up to 30 for some particularly large instances. There are cases, however, in which FM is actually slower than Whitaker's method. This usually happens for smaller instances (with n or p small), in which the local search procedure performs very few iterations, insufficent to ammortize the overhead of using a matrix. On average, however, FM has proven to be from three to almost 12 times faster than FI.

We now analyze a second variant of our method. Instead of using a full matrix to represent *extra*, we use a sparse matrix, as described in Section 4.3.2. We call this variant SM. The results, obtained by the same process as above, are presented in Table 2.

As expected, SM has proven to be even faster than FM on average and in the best case (especially for the somewhat larger TSP instances). However, bad cases become even worse. This happens mostly for instances with small values of p: with the number of nonzero elements in the matrix relatively large, a sparse representation is not the best choice.

Table 2: Speedup obtained by SM (sparse matrix, no preprocessing) over Whitaker's FI.

CLASS	BEST	MEAN	WORST
ORLIB	17.21	3.10	0.74
RW	32.39	5.26	0.75
TSP	147.71	26.18	1.72

The last acceleration we study is the preprocessing step (Section 4.3.3), in which all potential facilities are sorted according to their distances from each of the users. Results for this variation (SMP, for *sparse matrix with preprocessing*) are presented in Table 3. Columns 2, 3, and 4 consider running times of the local search procedure only; columns 5, 6, and 7 also include preprocessing times.

Table 3: Speedup obtained by SMP (sparse matrix, full preprocessing) over Whitaker's FI.

CLASS	LOCAL SEARCH ONLY			INCL. PREPROCESSING		
	BEST	MEAN	WORST	BEST	MEAN	WORST
ORLIB	67.0	8.7	1.30	7.5	1.2	0.22
RW	113.9	15.1	1.40	9.6	2.1	0.18
TSP	862.1	177.6	3.27	79.2	20.3	1.33

The table shows that the entire SMP procedure (including preprocessing) is in general still much faster than Whitaker's FI, but often worse than the other variants studied in this paper (FM and SM). However, as already mentioned, metaheuristics often need to run the local search procedure several times, starting from different solutions. Since preprocessing is run only once, its cost can be quickly amortized. Based on columns 2, 3, and 4 of the table, it is clear that, once this happens, SMP can achieve truly remarkable speedups with respect not only to FI, but also to other variants studied in this paper. In the best case (instance rl5934 with $p = 1000$), it is almost 900 times faster than FI.

A more detailed analysis of this particular instance (rl5934, the largest we tested) is presented in Figure 4. It shows how p (the number of facilities to open) affects the running times of all four methods studied (FI, FM, SM, and SMP), and also some variants "between" SM and SMP. Recall that in SMP every user keeps a list of

[4]Since we are dealing with ratios, geometric (rather than arithmetic) means seem to be a more sensible choice; after all, if a method takes twice as much time for 50% of the instances and half as much for the other 50%, it should be considered roughly equivalent to the other method. Geometric means reflect that, whereas arithmetic means do not.

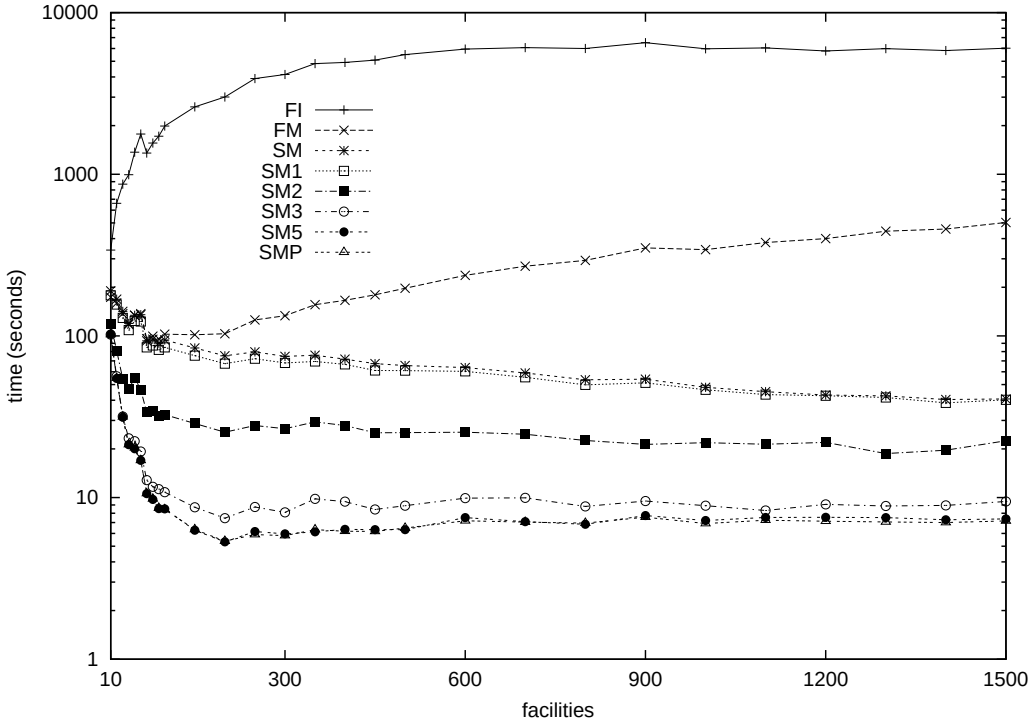

Figure 4: Instance rl5934: dependency of running times on p for different methods. Times are in logarithmic scale and do not include preprocessing.

all facilities sorted by distance; SM keeps no list at all. In a variant of the form SMq, each user keeps a limited list with the qm/p closest facilities (this is the "cache" version described in Section 4.3.3). Running times in the graph do not include preprocessing, which takes approximately one minute for this particular instance.

Since all methods discussed here implement the same algorithm, the number of iterations does not depend on the method itself. It does, however, depend on the value of p: in general, these two have a positive correlation. For some methods, such as Whitaker's FI and the full-matrix variation of our implementation (FM), an increase in p leads to greater running times (although our method is still 10 times faster for $p = 1500$). For SMP, which uses sparse matrices, time spent per iteration tends to decrease even faster as p increases: the effect of swaps becomes more local, with fewer users affected and fewer neighboring facilities visited in each call to `updateStructures`. This latter effect explains why keeping even a relatively small list of neighboring facilities for each user seems to be worthwhile. The curves for variants SMP and SM5 are practically indistinguishable in Figure 4, and both are much faster than SM.

6 Concluding Remarks

We have presented a new implementation of the swap-based local search for the p-median problem introduced by Teitz and Bart. Through the combination of several techniques — using a matrix to store partial results, a compressed representation for this matrix, and preprocessing — we were able to obtain speedups of up to three orders of magnitude with respect to the best previously known implementation, due to Whitaker. Our implementation is especially well suited to relatively large instances and, due to the preprocessing step, to situations in which the local search procedure is run several times for the same instance (such as within a metaheuristic). For small instances, Whitaker's can still be faster, but not by a significantly large margin.

Two lines of research suggest themselves from this work. First, it is still possible to improve the performance of our method, especially for small instances. One might consider, for example, incorporating the preprocessing step into the main procedure; this would allow operations to be performed as needed, instead of in advance (thus avoiding useless computations). A second line of research would be to test our method as a building block of more elaborate metaheuristics.

References

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

[2] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[3] P. Hansen and N. Mladenović. Variable neighborhood search for the p-median. *Location Science*, 5:207–226, 1997.

[4] P. Hansen, N. Mladenović, and D. Perez-Brito. Variable neighborhood decomposition search. *Journal of Heuristics*, 7(3):335–350, 2001.

[5] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[6] G. Reinelt. TSPLIB: A traveling salesman problem library. *ORSA Journal on Computing*, 3:376–384, 1991. *http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/*.

[7] E. Rolland, D. A. Schilling, and J. R. Current. An efficient tabu search procedure for the p-median problem. *European Journal of Operational Research*, 96:329–342, 1996.

[8] K. E. Rosing. An empirical investigation of the effectiveness of a vertex substitution heuristic. *Environment and Planning B*, 24:59–67, 1997.

[9] K. E. Rosing and C. S. ReVelle. Heuristic concentration: Two stage solution construction. *European Journal of Operational Research*, 97:75–86, 1997.

[10] M. B. Teitz and P. Bart. Heuristic methods for estimating the generalized vertex median of a weighted graph. *Operations Research*, 16(5):955–961, 1968.

[11] M. Thorup. Quick k-median, k-center, and facility location for sparse graphs. In *Proceedings of the 28th International Colloquium on Automata, Languages and Programming (ICALP 2001)*, volume 2076 of *Lecture Notes in Computer Science*, pages 249–260. Springer, 2001.

[12] S. Voss. A reverse elimination approach for the p-median problem. *Studies in Locational Analysis*, 8:49–58, 1996.

[13] R. Whitaker. A fast algorithm for the greedy interchange of large-scale clustering and median location prolems. *INFOR*, 21:95–108, 1983.