A Fast Swap-based Local Search Procedure for Location Problems *

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, proposed in 1968 by Teitz and Bart. Our method produces the same output as the best alternatives described in the literature and, even though it does not have a better worst-case complexity, it can be significantly faster in practice: speedups of up to three orders of magnitude were observed. We also show that our method can be easily adapted to handle the facility location problem and to implement related procedures, such as path-relinking and tabu search.

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 $d : U \times F \rightarrow \mathcal{R}_+$, and an integer $p \leq m$, determine which p facilities to open so as to minimize the sum of the distances from each user to the closest open facility. In other words, given p, we want to minimize the cost of serving all customers.

Since this problem is NP-hard [9], a polynomial-time algorithm to solve it exactly is unlikely to exist. In practice, one often resorts to heuristics instead. The most natural options are constructive heuristics, methods that build solutions from scratch, usually in a greedy fashion [4, 23]. A step further is to use a *local search procedure*, which takes an existing solution as input and tries to improve it. It does so in an iterative fashion, examining *neighboring solutions*, those that differ from the original one by a small (problem- and algorithm-specific) modification.

*September 4, 2003. AT&T Labs Research Technical Report TD-5R3KBH.

†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 study concerns the local search proposed by Teitz and Bart [20], based on swapping facilities. In each iteration, the algorithm looks for a pair of facilities (one to be inserted into the current solution, another to be removed) that would lead to an improved solution if swapped. If such a pair exists, the swap is made and the procedure is repeated.

Arya et al. have shown in [1] that, in the metric cases, this algorithm always finds solutions that are within a factor of at most 5 from the optimum. However, for practical, non-pathological instances the gap is usually much smaller, just a few percentage points [18, 23]. This has made the algorithm very popular among practioners, often appearing as a key subroutine of more elaborate metaheuristics [7, 15, 17, 19, 22].

Our concern in this paper is not solution quality—the reader is referred to [18, 23] for insights on that matter. Our goal is to obtain the same solutions Teitz and Bart would, only in less time. We present an implementation that is significantly (often asymptotically) faster in practice than previously known alternatives.

The 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 the best alternative implementation described in the literature, proposed by Whitaker [23]. Our own implementation is described in Section 4. We show how it can be adapted to handle the facility location problem and to handle related operations (such as path-relinking and tabu search) in Section 5. Experimental evidence to the efficiency of our method is presented in Section 6. Final remarks are made in Section 7.

Notation and assumptions. Before proceeding to the study of 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.

We use u to denote a generic user, and f a generic facility. The cost of serving u with f is $d(u, f)$, the *distance* between them, which is always nonnegative. (We do not make any other assumption about the distance function; in particular, we do not assume that the triangle inequality is valid.) A *solution* S is any subset of F with p elements, and represents the set of open facilities. Every user u is assigned to the closest facility $f \in S$ (the one that minimizes $d(u, f)$). This facility will be denoted by $\phi_1(u)$. Our algorithm often needs to access the second closest facility to u in S as well; it 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 $d_2(u)$.[1] We

[1] More accurate representations of $\phi_1(u)$, $\phi_2(u)$, $d_1(u)$, and $d_2(u)$ would be $\phi_1^S(u)$, $\phi_2^S(u)$, $d_1^S(u)$, and $d_2^S(u)$, respectively. After all, each value is a function of S as well. Since the solution will be clear from context, we prefer the simpler representation in the interest of

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. 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: for each of the n customers, we explicitly find the distances to the p open facilities and pick the smallest. Problems defined by a distance matrix clearly fall into the distance oracle model, but an explicit matrix is not always necessary. If users and facilities are points on the plane, for example, distances can also be computed in constant time. There are cases, however, in which that does not happen, such as when the input is given as a sparse graph, with distances determined by shortest paths. In such situations, one must precompute the corresponding distance matrix in order to apply our method with the same worst-case running time.

2 The Swap-based Local Search

Introduced by Teitz and Bart in [20], the standard local search procedure for the p-median problem is based on swapping facilities. For each facility $f_i \notin S$ (the current solution), 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 such "improving" swaps exist, the best one is performed, and the procedure is repeated from the new solution. Otherwise we stop, having reached a *local minimum* (or *local optimum*). It has recently been proven [1] that this procedure is guaranteed to produce a solution whose value is at most 5 times the optimum. On non-pathological instances (those more likely to appear in practice), empirical evidence shows that the algorithm is often within a few percentage points of optimality (and often does find the optimal solution), being especially successful when both p and n are small [18].

Our main concern is not solution quality, but the time it takes to run each iteration of the algorithm. Given a solution S, we want to find an improving neighbor S' (if it exists) as fast as possible.

A straighforward implementation takes $O(pmn)$ time per iteration. Start by determining the closest and second closest open facilities for each user; this takes $O(pn)$ time. Then, for each candidate pair (f_i, f_r), compute the profit that would result from replacing f_r with f_i. To do that, one can reason about each user u independently. If the facility that currently serves u is not f_r (the facility to be removed), the user will switch to f_i only if this facility is closer, otherwise it will remain where it is. If u is currently assigned to f_r, the user will have to be reassigned, either to $\phi_2(u)$ (the second closest facility) or to $\phi_1(u)$ (the closest facility), whichever is closest. The net effect is summarized

readability.

by following expression:

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

The first summation accounts for facilities that are not currently assigned to f_r (these can only gain from the swap), and the second for facilities that are (they can gain or lose something with the swap). In the distance oracle model, the entire expression can be computed in $O(n)$ time for each candidate pair of facilites. There are p candidates for removal and $m - p$ for insertion, so the total number of moves to consider is $p(m - p) = O(pm)$. Each iteration therefore takes $O(pmn)$ time.

Several papers in the literature use this basic implementation, and others avoid using the swap-based local search altogether mentioning its intolerable running time [17, 19, 22]. These methods would greatly benefit from asymptotically faster implementations, such as Whitaker's or ours.

3 Whitaker's Implementation

In [23], 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 [7] applied it as a subroutine of a Variable Neighborhood Search (VNS) procedure. A minor difference between the implementations is that 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, even in references to "Whitaker's algorithm".

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 the function that does that, adapted from [7], is presented in Figure 1.[2] Function `findOut` takes as an 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*).

Given a certain candidate for insertion f_i, the function implicitly computes $\mathit{profit}(f_i, f_r)$ for all possible candidates f_r. What makes this procedure fast is the observation (due to Whitaker) that the profit can be decomposed into two components, which we call *gain* and *netloss*.

Component *gain* accounts for all users who would benefit from the insertion of f_i into the solution. Each is closer to f_i than to the facility it is currently assigned to. The difference between the distances is the amount by which the

[2] In the code, an expression of the form $a \overset{+}{\leftarrow} b$ means that the value of a is incremented by b units.

```
function findOut (S, f_i, φ_1, φ_2)
1     gain ← 0; /* gain resulting from the addition of f_i */
2     forall (f ∈ S) do netloss(f) ← 0; /* loss resulting from removal of f */
3     forall (u ∈ U) do
4           if (d(u, f_i) ≤ d_1(u)) then /* gain if f_i is close enough to u */
5                 gain +← [d_1(u) − d(u, f_i)];
6           else /* loss if facility that is closest to u is removed */
7                 netloss(φ_1(u)) +← min{d(u, f_i), d_2(u)} − d_1(u);
8           endif
9     endforall
10    f_r ← argmin_{f∈S}{netloss(f)};
11    profit ← gain − netloss(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 [7].

cost of serving that particular user will be reduced if f_i is inserted. Lines 4 and 5 of the pseudocode compute *gain*.

The second component, *netloss*, accounts for all other customers, those that would not benefit from the insertion of f_i into the solution. If the facility that is closest to u is removed, u would have to be reassigned either to $\phi_2(u)$ (its current second closest facility) or to f_i (the new facility), whichever is closest. In both cases, the cost of serving u will either increase or remain constant. Of course, this reassignment will only be necessary if $\phi_1(u)$ is the facility removed to make room for f_i. This explains why *netloss* is an array, not a scalar value: there is one value associated with each candidate for removal. All values are initially set to zero in line 2; line 7 adds the contribution of the relevant customers.

Given this $O(n)$-time function, it is 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 [23]. 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 are stored in auxiliary

data structures. As we will see, with this approach we can use values computed in early iterations to speed up later ones.

4.1 Additional Structures

Before we present our algorithm, let us analyze Whitaker's algorithm from a broader perspective. Its ultimate goal is to determine the pair (f_i, f_r) of facilities that minimize $\mathit{profit}(f_i, f_r,)$. To do so, it computes $\mathit{gain}(f_i)$ for every candidate for insertion, and $\mathit{netloss}(f_i, f_r)$ for every pair of candidates. (In the description in Section 3, *gain* is a scalar and *netloss* takes as input only the facility to be removed; however, both are computed inside a function that is called for each f_i, which accounts the additional dimension.) Implicitly, what the algorithm does is to compute profit as

$$\mathit{profit}(f_i, f_r, =)\mathit{gain}(f_i) - \mathit{netloss}(f_i, f_r, .)$$

Our algorithm defines $\mathit{gain}(f_i)$ precisely as in Whitaker's algorithm: it represents the total amount gained if f_i is added to S, regardless of which facility is removed:

$$\mathit{gain}(f_i) = \sum_{u \in U} \max\{0, d_1(u) - d(u, f_i)\}. \tag{1}$$

Our method differs from Whitaker's in the computation of *netloss*. While Whitaker's algorithm computes it explicitly, we do it in an indirect fashion. For every facility f_r in the solution, we define $\mathit{loss}(f_r)$ as the increase in solution value that results from the removal of f_r from the solution (assuming that no facility is inserted). This is the cost of transferring every customer assigned to f_r to its second closest facility:

$$\mathit{loss}(f_r) = \sum_{u:\phi_1(u)=f_r} [d_2(u) - d_1(u)]. \tag{2}$$

As defined, *gain* and *loss* are capable of determining the net effect of a single insertion or a single deletion, but not of a swap, which is nothing but an insertion and a deletion that occur simultaneously. Whitaker's algorithm can handle swaps because it computes *netloss* instead of *loss*. To compute *netloss* from *loss*, we use yet another function, $\mathit{extra}(f_i, f_r,)$, defined so that the following is true for all pairs (f_i, f_r):

$$\mathit{extra}(f_i, f_r) = \mathit{loss}(f_r) - \mathit{netloss}(f_i, f_r). \tag{3}$$

From the pseudocode in Figure 1, it is clear that $\mathit{netloss}(f_i, f_r)$ is actually defined as

$$\mathit{netloss}(f_i, f_r) = \sum_{\substack{u:[\phi_1(u)=f_r]\wedge \\ [d(u,f_i)>d_1(u)]}} [\min\{d(u, f_i), d_2(u)\} - d_1(u)]. \tag{4}$$

Sustituting the values in Equations 2 and 4 into Equation 3, we obtain an expression for *extra*:

$$extra(f_i, f_r) = \sum_{u:\phi_1(u)=f_r} [d_2(u) - d_1(u)] - \sum_{\substack{u:[\phi_1(u)=f_r]\wedge \\ [d(u,f_i)>d_1(u)]}} [\min\{d(u, f_i), d_2(u)\} - d_1(u)].$$

It is possible to simplify this expression. First, consider a user u for which $\min\{d(u, f_i), d_2(u)\} = d_2(u)$. It has no net contribution to *extra*: whatever is added in the first summation is subtracted in the second. Therefore, we can write

$$extra(f_i, f_r) = \sum_{\substack{u:[\phi_1(u)=f_r]\wedge \\ [d(u,f_i)<d_2(u)]}} [d_2(u) - d_1(u)] - \sum_{\substack{u:[\phi_1(u)=f_r]\wedge \\ [d_1(u)<d(u,f_i)<d_2(u)]}} [d(u, f_i) - d_1(u)].$$

Note that the range of the first summation contains that of the second. We can join both into a single summation,

$$extra(f_i, f_r) = \sum_{\substack{u:[\phi_1(u)=f_r]\wedge \\ [d(u,f_i)<d_2(u)]}} [d_2(u) - d_1(u) - \max\{0, d(u, f_i) - d_1(u)\}],$$

which can be further simplified to

$$extra(f_i, f_r) = \sum_{\substack{u:[\phi_1(u)=f_r]\wedge \\ [d(u,f_i)<d_2(u)]}} [d_2(u) - \max\{d(u, f_i), d_1(u)\}]. \tag{5}$$

This is our "final" expression for *extra*. We derived it algebraically from simpler expressions, but it is possible to get it directly with a bit of case analysis. This approach was used in the earlier version of our paper [16].

Given the expressions of *gain*, *loss*, and *extra* (Equations 1, 2, and 5), we can find the profit associated with each move in a very simple manner:

$$profit(f_i, f_r) = gain(f_i) - loss(f_r) + extra(f_i, f_r). \tag{6}$$

The interesting aspect of this decomposition of *profit* is that the only term that depends on both the facility to be inserted and the one to be removed is *extra*. Moreover, this term is always nonnegative (see Equation 5). This will be relevant in the implementation of the local search itself, as the next section will make clear.

4.2 Local Search

Our implementation of the local search procedure assumes that all necessary values (*loss*, *gain*, and *extra*) are stored in appropriate data structures: one-dimensional vectors for *loss* and *gain*, a two-dimensional matrix for *extra*.[3] Once

[3]Note that *gain* and *loss* could actually share the same m-sized vector, since they are defined for disjoint sets of facilities.

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

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

these structures are computed, one can easily find the best swap in $O(pm)$ time: just use Equation 6 to determine the profit for each candidate pair of facilities and pick the minimum.

To compute *gain*, *loss*, and *extra*, we note that every entry in these structures is a summation over some subset of users (see Equations 1, 2, and 5). The contribution of each user can therefore be computed independently. Function `updateStructures`, shown in Figure 2, does exactly that. Given a user u and its closest facilities in solution S (given by ϕ_1 and ϕ_2), it adds u's contribution to *loss*, *gain*, and *extra*. The total running time of the procedure is $O(m-p) = O(m)$, since it is essentially a loop through all the facilities that do not belong to the solution. Given this function, computing *gain*, *loss*, and *extra* from scratch is straightforward: first reset all entries in these structures, then call `updateStructures` once for each user. Together, these n calls perform precisely the summations defined in Equations 1, 2, and 5.

We now have all the elements necessary to build the local search procedure with $O(mn)$ operation. 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 taking in $O(m)$ time, determine their actual values. Finally, in $O(pm)$ time, find the best swap using Equation 6.

4.3 Acceleration

At first, our implementation seems to be merely a 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, whereas $\Theta(n+m)$ memory positions are enough for Whitaker's. The additional memory, however, allows for significant accelerations, as this section will show.

When a facility f_r is replaced by a new facility f_i, certain entries in *gain*, *loss*, *extra*, ϕ_1, and ϕ_2 become inaccurate. The 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.

The problem with this approach is that no information gathered in one iteration is used in subsequent ones. As a result, unnecessary, repeated computations are bound to occur. In fact, the actions performed by `updateStructures` depend only on u, $\phi_1(u)$, and $\phi_2(u)$; no value is read from other structures. If $\phi_1(u)$ and $\phi_2(u)$ do not change from one iteration to another, u's contribution to *gain*, *loss*, and *extra* will not change either. In other words, there is no need to call `updateStructures` again for u.

To deal with such cases, we keep track of *affected users*. A user u is *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. either $\phi_1(u)$ or $\phi_2(u)$ is f_r, the facility removed; or
2. f_i (the facility inserted) is closer to u than the original $\phi_2(u)$ is.

Contributions to *loss*, *gain*, and *extra* need only be updated for affected users. If there happens to be few of them (and there often are, Section 6.2.1 shows) significant gains can be obtained.

Note, however, that updating the contributions of an affected user u requires more than a call to `updateStructures`. This function simply adds new contributions, so we must also subtract the old contributions made by u. To acomplish this second task, we use a function similar to `updateStructures`, with subtractions instead of additions.[4] This function (`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 already discussed, three others appear in the code. Function `resetStructures`, sets all entries in *gain*, *loss*, and *extra* to zero. Function `findBestNeighbor` runs through these structures and finds the most profitable swap using Equation 6. It returns which facility to remove (f_r), the one to replace it (f_i), and the profit itself (*profit*). Finally, `updateClosest` 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 (Section 4.3.1 explains how this is done).

Restricting updates to affected users can result in significant speedups in the algorithm, as Section 6.2.1 shows. There are, however, other accelerations to exploit. The pseudocode reveals that all operations in the main loop run in linear time, with three exceptions:

- updating closeness information (calls to `updateClosest`);
- finding the best swap to be made (calls to `findBestNeighbor`);
- updating the auxiliary data structures (calls to `updateStructures` and `undoUpdateStructures`).

[4] This function is identical to the one shown in Figure 2, with all occurrences of $\stackrel{+}{\leftarrow}$ replaced with $\stackrel{-}{\leftarrow}$: 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 (S, u, gain, loss, extra, φ1, φ2);
5       (fr, fi, profit) ←findBestNeighbor (gain, loss, extra);
6       if (profit ≤ 0) then 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 (S, 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

These are the bottlenecks of the algorithm, since they all run in quadratic time in the worst case. The next three subsections analyze how each of the bottlenecks can be dealt with.

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 updating closeness information for customers who had f_r (the facility removed) as either the closest or the second closest element. With a straighforward implementation, updating each such affected user takes $O(p)$ time. Since there usually are few of them, the total time spent tends to be small fraction of the entire local search procedure.

It is actually possible to perform the entire update procedure in $O(n \log p)$ worst-case time. It suffices to keep, for each user u, the set of open facilities in a heap with priorities given by their distances to u. Since this solution requires $O(np)$ additional memory positions and is not significantly faster, we opted for using the straighforward implementation in our code.

In is important to mention that in some settings, finding the set of closest and second closest elements from scratch is itself a cheap operation, even in the worst case. 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 [21], where $|E|$ is the number of edges in the graph.[5]

In practice, the generic approach above seems to be good enough. Section 6.2.5 shows that there is not much to gain from accelerating this part of the algorithm; other procedures already dominate the running time of the local search. We therefore do not use specialized routines in this paper; we always deal with arbitrary distance matrices.

4.3.2 Best Neighbor

Given a solution, the straighforward way to find the most profitable swap is to compute $\mathit{profit}(f_i, f_r)$ (as defined in Equation 6) for all candidate pairs of facilities and pick the best. Each *profit* computation takes constant time and there are $p(m-p)$ potential swaps, so the entire procedure requires $\Theta(pm)$ operations. In practice, however, the best move can be found in less time.

It is convenient to think of $\mathit{extra}(f_i, f_r)$ as a measure of the interaction between the neighborhoods of f_r and f_i. After all, Equation 5 shows that only users that have f_r as their current closest facility and are also close to f_i (i.e., have f_i closer than their second closest open facility) contribute to $\mathit{extra}(f_i, f_r)$. In particular, if there are no users in this situation, $\mathit{extra}(f_i, f_r)$ will be zero. Section 6.2.2 shows that this occurs rather frequently in practice, especially when p is large (and hence the average number of users assigned to each f_r is small).

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

In particular, consider the implementation of `bestNeighbor`. First, it determines the facility f_i that maximizes $\mathit{gain}(f_i)$ and the facility f_r that minimizes $\mathit{loss}(f_r)$. Since all values in *extra* are nonnegative, the pair (f_i, f_r) is at least as profitable as any pair $(f_{i'}, f_{r'})$ for which $\mathit{extra}(f_{i'}, f_{r'})$ is zero. Then, the procedure computes the exact profits (given by Equation 6) 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. As already mentioned, this value tends to be smaller as p increases, thus making the algorithm not only faster, but also more memory-efficient (when compared to the "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

[5]The $\tilde{O}(\cdot)$ notation hides polylogarithmic terms.

these calls.

Consider the pseudocode in Figure 2. Line 5 represents a loop through all $m-p$ facilities outside the solution, but line 6 shows that we can actually restrict ourselves to facilities that are closer to u than $\phi_2(u)$ is. This is often a small subset of the facilities, especially when p is large.

This suggests a preprocessing step that builds, for each user u, a list of all facilities sorted by increasing distance to u. During the local search, whenever we need the set of facilities whose distance to u is less than $d_2(u)$, we just take the appropriate prefix of the precomputed list, potentially with much fewer than $m - p$ elements.

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 (such as in [7, 16, 19]): a single preprocessing step is enough.

A more serious drawback of this approach is memory usage. Keeping n lists of size m in memory requires $\Theta(mn)$ space, which may be prohibitive. An alternative is to keep only relatively small prefixes, not the full list. They would act as a cache: when $d_2(u)$ is small enough, we just take a prefix of the candidate list; when it is larger than the largest element represented, we explicitly look at all possible neighbors (each in constant time).

In some circumstances, the "cache" version may be faster than the "full" version of the algorithm, since preprocessing is cheaper. After all, instead of creating sorted lists of size m, we create smaller ones of size k (for some $k < m$). Each list can be created in $O(m + k \log k)$ time: first we find the k smallest elements among all m (in $O(m)$ time [3]), then we sort them (in $O(k \log k)$ time). For small values of k, this is an asymptotic improvement over the $O(m \log m)$ required (per list) in the "full" case.

4.3.4 The Reordering Problem

There is a slight incompatibility between the accelerations proposed in Sections 4.3.2 and 4.3.3. On the one hand, the sparse matrix data structure proposed in Section 4.3.2 guarantees efficient queries only when each row is accessed sequentially by column number (facility label). Section 4.3.3, on the other hand, assumes that facilites are accessed in nondecreasing order of *distance* from the user. Functions `updateStrutures` and `undoUpdateStrutures` must use both data structures: they take a list of facilities sorted by distance, but process them in nondecreasing order of label. We need to make these two operations compatible.

The simplest solution is to take the list of facilities sorted by distance and sort it again by label. If the list has size k, this takes $O(k \log k)$ time. In the worst case k is $O(m)$, so this introduces an extra $\log m$ factor in the complexity of the algorithm. In practice, however, k is rather small, and the overhead hardly noticeable. In fact, we used this approach in the preliminary version of our paper [14].

Even so, one would like to do better. Recall that the original list is actually

a prefix of the list of all facilities (sorted by distance). Even though the prefix varies in size, the underlying sorted list does not: it is a fixed permutation of facility labels. This means we need to solve the following generic problem:

> Let π be a fixed permutation of the labels $\{1, 2, \ldots, m\}$, and let π_k be the size-k prefix of π, for $1 \leq k \leq n$ ($\pi_n = \pi$, by definition). Given any k, sort π_k by label in $O(k)$ time. At most $O(m)$ preprocessing time is allowed.

To solve this, we use an algorithm that mimics insertion sort on a list, but takes advice from an "oracle" built during preprocessing. Assume we need to sort π_k, for some k. One way to do it is to take each element of π_k and insert it into a new list in order (of label). With standard insertion sort, this would take $O(k^2)$ time. However, if we knew in advance where to insert each element, the procedure would take $O(k)$ time. The oracle will give us exactly that.

Let $\pi(i)$ be the i-th element of π. We define $pred(i)$ to be the *predecessor* of $\pi(i)$, the element after which $\pi(i)$ should be inserted during the algoritm above. The oracle will give us $pred(i)$ for every i.

The values of $pred(i)$ are set in the preprocessing step. Initially, it creates an auxiliary doubly-linked list L containing $0, 1, 2, \ldots, m$, in this order (element 0 will act as a sentinel). This can be trivially done in $O(m)$ time. Then, it removes elements from L one by one in *reverse order* with respect to π. In other words, the first element removed from L is $\pi(m)$, then $\pi(m-1)$, and so on, until $\pi(1)$ is removed; in the end, only the sentinel (0) will remain in L. Upon removing element $\pi(i)$ from L, the algorithm sets $p(i)$ to be the predecessor of $\pi(i)$ (in L itself) at that particular moment. This procedure takes $O(m)$ time (for each of the n lists).

Note that this procedure is in fact a simulation insertion sort, but in reverse order. List L originally has all the elements of π_m; after one removal, we are left with π_{m-1}, and so on. At all times, L is sorted by label; if it has size k, it represents what the sequence looks like after the k-th element is inserted during insertion sort.

Given all the $pred(\cdot)$ values, sorting π_k is simple. We start with a list L' containing only the sentinel (0); it can be singly-linked, with forward pointers only. We then remove the first i elements of π (following π's own order), inserting each element $\pi(i)$ into L' right after $pred(i)$. Eventually, L' will contain all the elements of $\pi(k)$ sorted by label, as desired. The running time is only $O(k)$.

5 Generalization

Section 4 presented our algorithm as a local search procedure for the p-median problem. In fact, with slight modifications, it can also be applied to the facility location problem. Moreover, the ideas suggested here are not limited to local search: they can also be used to accelerate other important routines, such as path-relinking and tabu search. This section details the adaptations that must be made in each case.

5.1 Facility Location

The input of the *facility location problem* consists of a set of users U, a set of potential facilities F, a distance function $d : U \times F \to \mathcal{R}_+$, and a *setup cost function* $c : F \to \mathcal{R}_+$. The first three parameters are the same as in the p-median problem. The difference is that here the number of facilities to open is not fixed; there is, instead, a cost associated with opening each facilty, the *setup cost*. The more facilities are opened, the greater the setup cost will be. The objective is to minimize the total cost of serving all customers, considering the sum of the setup and service cost (distances).

Any valid solution to the p-median problem is a valid solution to the facility location problem. To use the local search procedure suggested here for this problem, we have to adjust the algorithm to compute the cost function correctly. As it is, the algorithm computes the service costs exactly, but assumes that the setup costs are zero. But including them is trivial: the service cost depends only on whether a facility is open or not; it does not depend on other facilities. Consider a facility f_i that is not in the solution; when evaluating whether it should be inserted as not, we must account for the fact that its setup cost will increase the solution value by $c(f_i)$. Similarly, simply closing a facility f_r that belongs to the solution saves us $c(f_r)$. To take these values into account, we just have to make sure that we initialize *gain* and *loss* with the symmetric of the corresponding setup costs, and not with zero as we do with the p-median problem. In other words, we initialize $gain(f_i)$ with $-c(f_i)$, and $loss(f_r)$ with $-c(f_r)$.

This is enough to implement a swap-based local search for the facility location problem. Note, however, that there is no reason to limit ourselves to swaps only—we could allow individual insertions and deletions as well. This is not possible with the p-median problem because the number of facilities is fixed, but there is no such constraint in the facility location problem.

No major change to the algorithm is necessary to support individual insertions and deletions. As already mentioned, $gain(f_i)$ is exactly the amount that would be saved if facility f_i were inserted into the solution. Similarly, $loss(f_r)$ represents how much would be lost if the facility were removed. Positive values of *gain* and negative values of *loss* indicate that the corresponding move is worth making. The greater the absolute value, the better, and we can find the maximum in $O(m)$ time. Furthermore, we can continue to compute associated with swaps if we wish to. In every iteration of the local search, we could therefore choose the best move among all swaps, insertions, and deletions. So we essentially gain the ability to make insertions and deletions at no extra cost.

We observe that the idea of a swap-based local search for the facility location problem is, of course, not new; it was first suggested by Kuehn and Hamburger in [10].

5.2 Other Applications

It is possible to adapt the algorithm to perform other routines, not only local search. (In this discussion, we will always deal with the p-median problem itself, although the algorithms suggested here also apply to facility location with minor adaptations.)

Consider the path-relinking operation [5, 6, 11]. It takes two solutions as inputs, S_1 and S_2, and gradually transforms the first (the *starting solution*) into the second (the *guiding solution*). It does so by swapping out facilities that are in $S_1 \setminus S_2$ and swapping in facilities from $S_2 \setminus S_1$. In each iteration of the algorithm, the best available swap is made. The goal of this procedure is to discover some promising solutions in the path from S_1 to S_2. The precise use of these solutions varies depending on the metaheuristic using this procedure.

We note that this function is remarkably similar to the swap-based local search procedure. Both are based on the same kind of move (swaps), and both make the cheapest move on each round. There are two main differences:

1. *Candidate moves*: In path-relinking, only a subset of the facilities in the solution are candidates for removal, and only a subset of those outside the solution are candidates for insertion—and these subsets change (i.e., get smaller) over time, as the algorithm advances into the path.

2. *Stopping criterion*: Whereas the local search procedure stops as soon as a local minimum is found, non-improving moves are allowed in path-relinking: it continues until the guiding solution is reached.

As long as we take these differences into account, the implementation of the local search procedure can also handle path-relinking. We need to define two functions: one to return the appropriate set of candidates for insertion and deletion, another to check if the move chosen by `bestNeighbor` should be made or not (i.e., to determine if the stopping criterion was met). In Section 4, these functions were defined implicitly: the candidates for insertion are all facilities outside the solution, the candidates for deletion are those in the solution, and the stopping criterion consists of testing whether the profit associated with a move if positive. Defining them explicitly is trivial for both local search and path-relinking.

In fact, by redefining these two functions appropriately, we can implement other routines, such as a simple version of tabu search. At all times, we could have two lists: one for elements that are forbidden to be inserted into the solution, another for elements that cannot be removed. The candidate lists would be the remaining facilities, and the stopping criterion could be any one used for tabu search (number of iterations, for instance).

6 Empirical Analysis

This section has two main goals. One is to present some empirical data to back up some of the claims we have made to guide our search for a faster algorithm.

The other goal is to demonstrate that the algorithms suggested here are indeed faster than previously existing implementations of the local search procedure for the p-median problem. To keep the analysis focused, we will not deal with the extensions proposed in Section 5.

6.1 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 set of instances that benefit less from our methods.

Class TSP contains three sets of points on the plane (with cardinality 1400, 3038, and 5934), originally used in the context of the traveling salesman problem [13]. In the p-median problem, each point is both a user to be served and a potential facility, and distances are Euclidean. Following [8], we tested several values of p for each instance, ranging from 10 to approximately $n/3$, when comparing our algorithm to Whitaker's.

Class ORLIB, originally introduced in [2], 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 [7, 8]. The shortest path computations are not included in the running times reported in this section, since they are the same for all methods (including Whitaker's).

Each instance in class RW is a square matrix in which entry (u, f) is an integer taken uniformly at random from the interval $[1, n]$ and 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.[6] The random number generator we used when creating these instances (and in the algorithm itself) was Matsumoto and Nishimura's *Mersenne Twister* [12].

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`. Both the source code and the instances are available from the authors upon request.

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, so 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 [23]: starting from an empty solution, we

[6] 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.

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.

6.2 Results

This section presents an experimental comparison of several variants of our implementation and Whitaker's method, *fast interchange* (we will use FI for short). We implemented FI based on the pseudocode in [7] (obtaining comparable running times); the most important function is presented here in Figure 1.

6.2.1 Basic Algorithm (FM)

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).

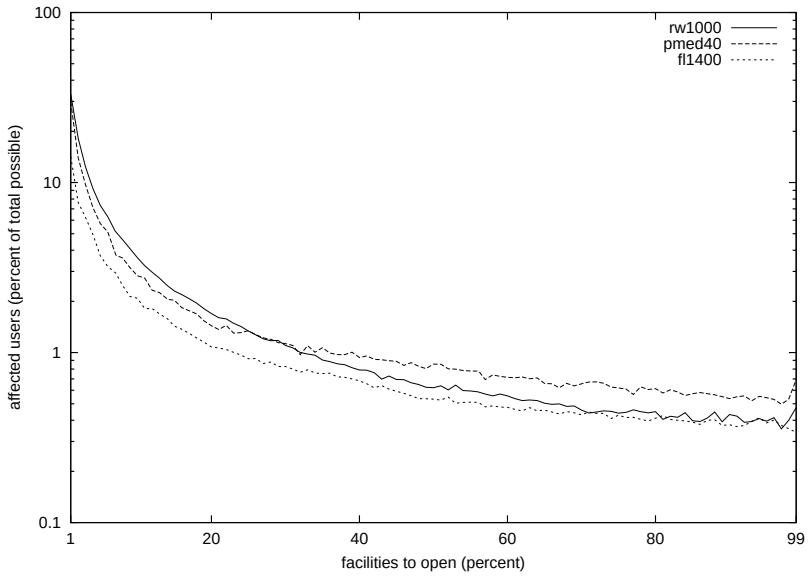

Figure 4: Percentage of users affected during a run of the local search as a function of p (percentage is taken over the set of all possible users that could have been affected, considering all iterations). One instance in each series is represented. Vertical axis is in logarithmic scale.

To demonstrate that keeping track of affected users can potentially lead to significant speedups, we devised the following experiment. We took one

instance from each class: pmed40 (class ORLIB, 900 nodes), fl1400 (class TSP, 1400 nodes), and rw1000 (class RW, 1000 nodes). Each instance was tested with 99 different values of p, from 1% to 99% of m. In each case, we computed how many calls to `updateStructures` and `undoUpdateStructures` would have been made if we were not keeping track of affected users, and how many calls were actually made (in both cases, we did not count calls at the start of the first iteration, which is just the initialization). The ratio between these values, in percentage terms, is shown in Figure 4.

It is clear that the average number of affected users is only a fraction of the total number of users, even with very few facilities, and drops significantly as the number of facilities to open increases. In all three instances, fewer than 1% of the users are affected on average once p increases. By exploiting this fact, our implementation definitely has the potential to be faster than FI.

To test if this is indeed the case in practice, we ran an experiment with all instances from the three classes. For each instance, 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.[7] Values greater than 1.0 favor our method, FM.

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

CLASS	BEST	MEAN	WORST
ORLIB	14.87	3.02	0.66
RW	12.69	4.28	1.10
TSP	29.03	11.23	1.80

The table shows that even the basic acceleration scheme achieves speedups of almost 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 more than 11 times faster than FI.

6.2.2 Sparse Matrix (SM)

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. Recall that our rationale for using a sparse matrix was

[7] 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.

that the number of nonzero elements in the *extra* memory is small. Figure 5 suggests that this is indeed true. For each of the three representative instances and each value of p, it shows fraction of elements that are nonzero (considering all iterations of the local search).

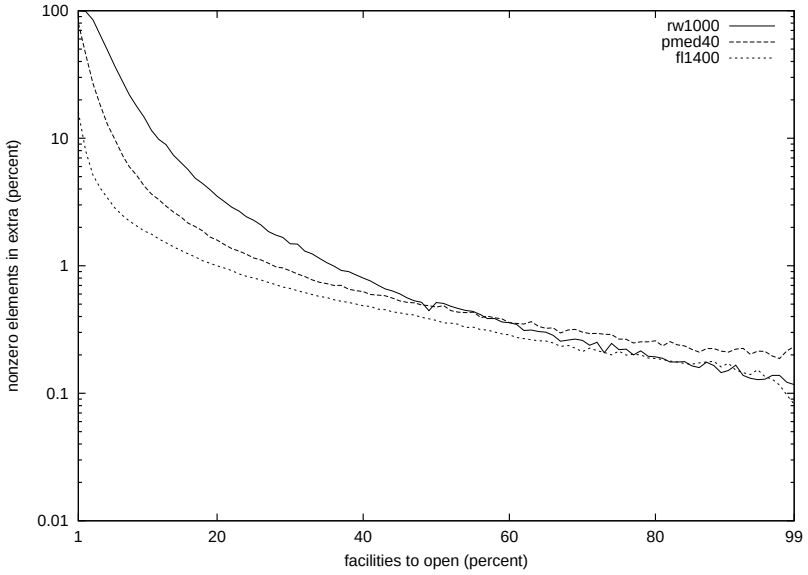

Figure 5: Percentage of entries in the extra matrix that have nonzero values as a function of p. One instance of each series is represented. Vertical axis is in logarithmic scale.

Although the percentage approaches 100% when the number of facilities to open is small, it drops very fast when p increases, approaching 0.1%. Note that rw1000, which is random, tends to have significantly more nonzeros for small values of p than other instances.

It is clear that the algorithm has a lot to benefit from representing only the nonzero elements of *extra*. However, the sparse matrix representation is much more involved than the array-based one, so some overhead is to be expected. Does it really reduce the running time of the algorithm in practice?

Table 2 shows that the answer to this question is "yes" most of the time. It represents the results obtained from all instances in the three classes, and contains the best, mean, and worst speedups obtained by SM over FI.

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 slightly 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	25.23	3.59	0.59
RW	69.98	6.66	0.94
TSP	145.37	26.27	1.65

6.2.3 Sparse Matrix with Preprocessing (SMP)

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. We call this variant SMP, for *sparse matrix with preprocessing*. The goal of the acceleration is to avoid looping through all m facilities in each call to function `updateStructures` (and `undoUpdateStructures`). We just have to find the appropriate prefix of the ordered list.

Figure 6 shows the average size of the prefixes (as a percentage of m) that are actually checked by the algorithm, as a function of p (for one instance on each class).

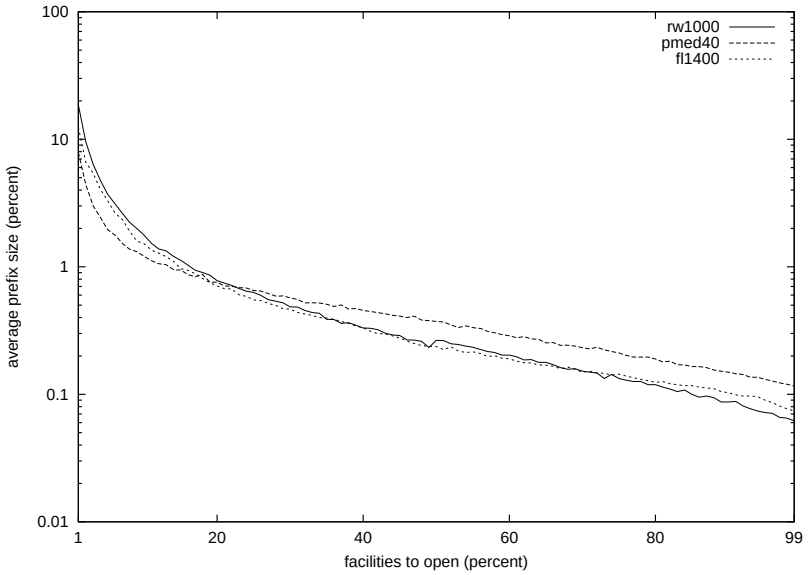

Figure 6: Percentage of facilities actually visited when updating structures, for several values of p. One instance of each series is represented. Vertical axis is in logarithmic scale.

As claimed before, the average prefix size is only tiny a fraction of m, for all but very small values of p. Considering only those prefixes instead of all facilities can potentially accelerate the local search. Of course, this does not come for free: the cost of preprocessing must be accounted for.

To determine the overall effect of these two conflicting factors, we tested SMP on all instances of our set. Table 3 shows the best, mean, and worst speedups obtained. 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			INCLUDING PREPROCESSING		
	BEST	MEAN	WORST	BEST	MEAN	WORST
ORLIB	78.59	9.76	1.42	19.08	2.98	0.63
RW	164.15	16.90	1.91	42.49	5.15	0.86
TSP	746.71	185.21	4.92	109.51	28.00	1.79

The table shows that the entire SMP procedure (including preprocessing) is on average still much faster than Whitaker's FI, but often slightly slower 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. Columns 2, 3, and 4 of the table show 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 $p = 900$), it is more than 700 times faster than FI.

Apart from the preprocessing time, another important downside of strategy SMP is memory usage: an array of size m is kept for each of the n customers. As mentioned in Section 4.3.3, one can use less memory by storing a vector with only a fraction of the m facilities for each customer. Table 4 shows what happens when we restrict the number of elements per vector to $5m/p$; we call this version of the local search SP5. In general, SMq is an algorithm that associates a list with qm/p facilities with each user; we parameterize the algorithm in terms of m/p because this is the average number of users assigned to each open facility.

Tables 3 and 4 show that using restricted lists (as opposed to m-sized ones) can make the algorithm significantly faster when preprocessing times are considered. This is true especially for large instances. On series TSP, for example, method SM5 is twice as fast as SMP. The gains from a faster preprocessing more than offset the potential extra time incurred during the actual local search. In fact, the table also shows that the time spent on the main loop is barely distinguishable; the partial lists are almost always enough for the algorithm. Local search within SM5 can actually be slightly *faster* than within SMP. The possible cause here are cache effects; since less data is kept in memory, there is more locality to be exploit by the hardware.

Table 4: Speedup obtained by SM5 (sparse matrix, with preprocessing, cache size $5p/m$) over Whitaker's FI.

CLASS	LOCAL SEARCH ONLY			INCLUDING PREPROCESSING		
	BEST	MEAN	WORST	BEST	MEAN	WORST
ORLIB	77.00	9.78	1.42	32.35	3.87	0.63
RW	155.73	16.62	1.91	78.70	7.86	0.97
TSP	775.33	186.21	4.86	253.77	57.98	2.13

6.2.4 Overall Comparison

To get a better understanding of the performance of all variants proposed in this paper, we study in detail the largest instance in our set (rl5934, with almost 6000 customers and facilities). Figures 7 and 8 show the running times of several methods (FI, FM, SM, SM1, SM2, SM3, SM5, and SMP for different values of p. The first figure considers the local search only, whereas the second accounts for preprocessing times as well.

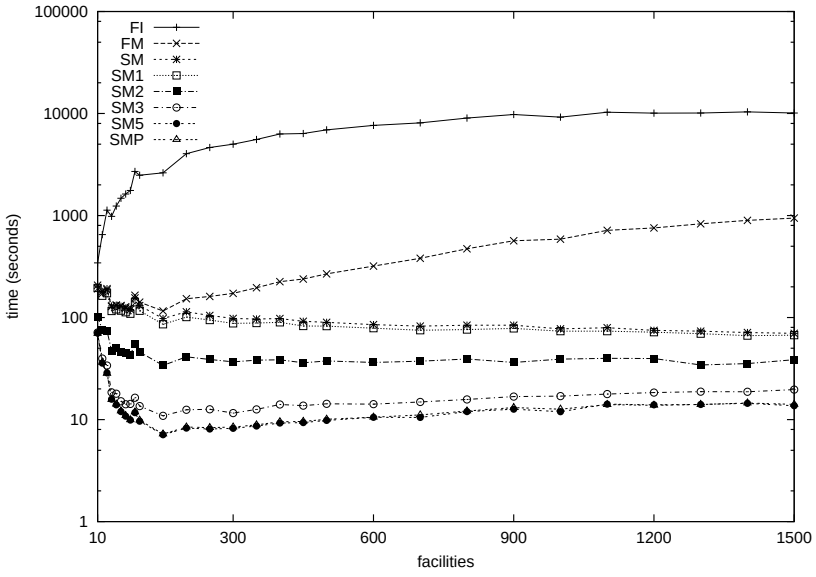

Figure 7: Instance rl5934: dependency of running times on p for different methods. Times are in logarithmic scale and do not include preprocessing.

The figures show that 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 $p =$

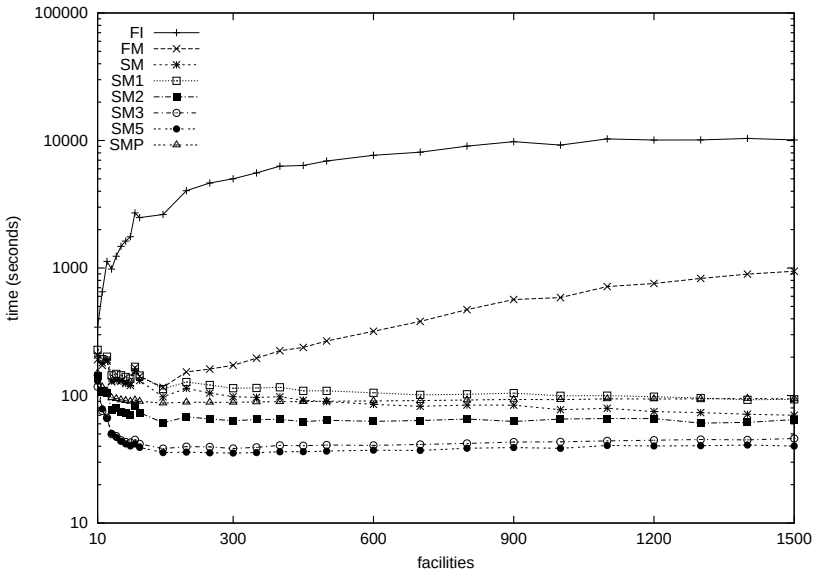

Figure 8: Instance rl5934: dependency of running times on p for different methods. Times are in logarithmic scale and include preprocessing where applicable.

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 7, and both are much faster than SM (which keeps no list at all).

As a final note, we observe that, because 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 $p \leq m/2$, and negative from this point on, as Figure 9 shows. This correlates well with the total number of solutions: there are $\binom{m}{p}$ solutions of size p, and this expression is maximized for $p = m/2$.

6.2.5 Profile

The results for SMP show that the series of modification proposed in this paper can, together, results in significant acceleration. How much further can we go? Can additional modifications to the algorithm make it even faster?

These are open questions. However, we argue that small modifications are unlikely to lead to major gains. As mentioned before, the algorithm has three potential bottlenecks: calls to `updateClosest`, `updateStructures` (and

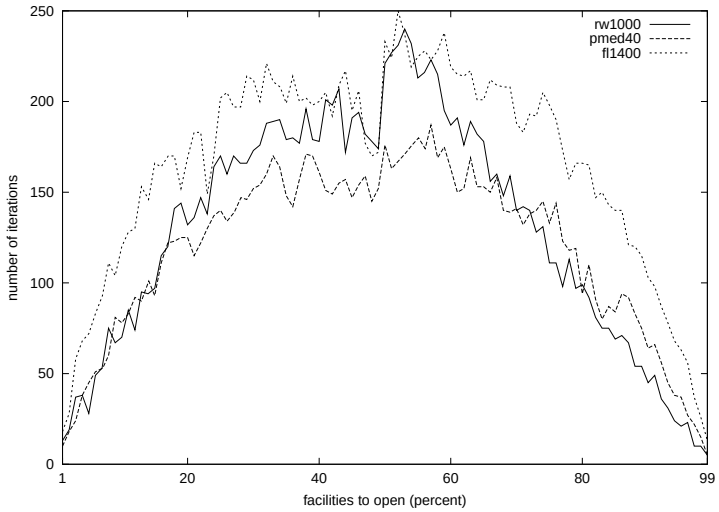

Figure 9: Number of iterations of the local search procedure as a function of p, starting from a greedy solution. One instance from each series is represented.

Table 5: Execution profile for method SMP: percentage of time spent on each of the potential bottlenecks (only the largest instance in each class is shown). Preprocessing times are not considered.

INSTANCE			UPDATE	UPDATE	BEST	OTHER
NAME	n, m	p	CLOSEST	STRUCT.	NEIGH.	OPER.
pmed40	900	90	9.7	36.4	25.5	28.4
rw1000	1000	500	10.1	49.1	27.3	13.5
rl5934	5934	1500	17.7	35.0	42.9	4.4

`undoUpdateStructures`), and `bestNeighbor`. For the largest instance in each series, we measured the fraction of time spent on each of these operations, and Table 5 shows that neither clearly dominates. No component took more than 50% of the running time. Therefore, even if we could make a component run in virtually no time, the algorithm would be at most twice as fast. A decent speedup, but not at all comparable to 750, the factor we were able to achieve in this paper. To obtain better factors, it seems necessary to work on all three bottlenecks, or to come up with a different strategy altogether.

7 Concluding Remarks

We have presented a new implementation of the swap-based local search for the p-median problem introduced by Teitz and Bart. We combine several techniques (using a matrix to store partial results, a compressed representation for this matrix, and preprocessing) 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 by a factor not greater than two.

An important test to the algorithms proposed here would be to apply them within more elaborate metaheuristics. We have done that in [15]. That paper describes a multistart heuristic for the p-median problem that relies heavily on local search and path-relinking, both implemented according to the guidelines detailed in this paper. The algorithm has proved to be very effective in practice, obtaining remarkably good results (in terms of running times and solution quality) when compared to other methods in the literature.

A possible extension of the work presented here would be to apply the methods and ideas presented here to problems beyond p-median and facility location. Swap-based local search is a natural operation to be performed on problems such as maximum cover, for example.

References

[1] V. Arya, N. Garg, R. Khandekar, A. Mayerson, K. Munagala, and V. Pandit. Local search heuristics for k-median and facility location problems. In *Proc. 33rd ACM Symposium on the Theory of Computing*, 2001.

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

[3] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[4] G. Cornuejols, M. L. Fisher, and G. L. Nemhauser. Location of bank accounts to optimize float: An analytical study of exact and approximate algorithms. *Management Science*, 23:789–810, 1977.

[5] 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.

[6] F. Glover, M. Laguna, and R. Martí. Fundamentals of scatter search and path relinking. *Control and Cybernetics*, 39:653–684, 2000.

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

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

[9] O. Kariv and L. Hakimi. An algorithmic approach to nework location problems, Part II: The p-medians. *SIAM Journal of Applied Mathematics*, 37(3):539–560, 1979.

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

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

[12] 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.

[13] 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/.*

[14] M. G. C. Resende and R. F. Werneck. On the implementation of a swap-based local search procedure for the p-median problem. Technical Report TD-5E4QKA, AT&T Labs Research, 2002.

[15] 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.

[16] 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.

[17] 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.

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

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

[20] 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.

[21] 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.

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

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