Fast bounding procedures for large instances of the Simple Plant Location Problem

Adam N. Letchford *, Sebastian J. Miller
Department of Management Science, Lancaster University, Lancaster LA1 4YX, United Kingdom

A R T I C L E   I N F O

Available online 29 June 2011

Keywords:
Combinatorial optimisation
Facility location
Dual ascent

A B S T R A C T

Some new, simple and extremely fast bounding procedures are presented for large-scale instances of the Simple Plant Location Problem. The lower-bounding procedures are based on dual ascent. The fastest of them runs in $O(mn \log m)$ time, where $m$ and $n$ are the number of locations and clients, respectively. The upper-bounding procedures are based on iteratively dropping facilities, and the fastest of them runs in $O(mn + \log m)$ time. Extensive computational results show that, in practice, the procedures give very good bounds extremely quickly.

© 2011 Elsevier Ltd. All rights reserved.

1. Introduction

In the Simple Plant Location Problem (SPLP) we have a set $I$ of locations and a set $J$ of clients. For any location $i \in I$, the fixed cost of opening a facility at $i$ is $f_i$. For any location $i \in I$ and any client $j \in J$, the cost of serving client $j$ from an open facility at location $i$ is $c_{ij}$. The task is to decide where to open facilities, and to assign each client to exactly one open facility, such that the total cost is minimised.

The SPLP is a well-known $NP$-hard combinatorial optimisation problem that has received a great deal of attention. A survey of early work on the SPLP (still relevant today) is given by Krarup and Puzan [20]. More recent surveys include Cornuéjols et al. [10] and Labbé and Louveaux [23]. We remark that, in some papers, the SPLP is called the Uncapacitated Facility Location Problem or UFLP.

The SPLP is normally formulated as the following Zero–One Linear Program (0–1 LP):

$$
\text{min } \sum_{i \in I} f_i y_i + \sum_{i \in I, j \in J} c_{ij} x_{ij} \\
\text{s.t. } \sum_{i \in I} x_{ij} = 1 \quad (\forall j \in J) \\
y_{i} - x_{ij} \geq 0 \quad (\forall i \in I, j \in J) \\
x_{ij} \in \{0, 1\} \quad (\forall i \in I, j \in J) \\
y_{i} \in \{0, 1\} \quad (\forall i \in I).
$$

Here, $y_{i}$ is a binary variable, taking the value 1 if and only if a facility is assigned to a location $i$, and $x_{ij}$ is a binary variable, taking the value 1 if and only if a facility is opened at location $j$.

A key feature of this 0–1 LP is that its LP relaxation is typically quite tight. Moreover, the dual of the LP relaxation can be solved to near-optimality very quickly using dual ascent [5], dual adjustment [11] or Lagrangian relaxation [4]. Even today, these dual-based procedures remain the methods of choice.

Now, let $m$ denote the number of locations and $n$ the number of clients. It is not hard to show (see Section 3.1) that the dual ascent method runs in $O(m^2n)$ time. This is of course polynomially bounded, but it can be excessively high when $m$ or $n$ is large. In this paper, we show how a simple modification to the algorithm can significantly improve its speed in practice. We also present an alternative dual ascent algorithm, which runs in $O(mn \log m)$ time, yet produces bounds of a similar quality.

In addition, we present a new upper-bounding procedure – based on iteratively dropping facilities in non-increasing order of reduced costs – that runs in $O(mn + \log m)$ time. Because it is so fast, we can safely call it several times, and keep the best upper bound generated. We ensure, however, that it is called no more than $O(\log m)$ times, to keep the running time small.

We remark that there exist some other effective lower- and upper-bounding procedures for the SPLP, which we review in Section 2. The emphasis in this paper is on procedures that are extremely fast (in both theory and practice), conceptually simple, and easy to implement.

The structure of the paper is as follows. In Section 2, the relevant literature is reviewed. In Section 3, the running time of the classical dual ascent procedure is analysed, and two modified versions are presented. In Section 4, the new upper-bounding procedure is presented, along with an analysis of its running time. In Section 5, some encouraging computational results are presented. Finally, concluding remarks are given in Section 6.

2. Literature review

We now review the literature. We cover lower bounds in Section 2.1, heuristics in Section 2.2, and exact methods in Section 2.3.
2.1. Lower bounds

After some simplification, the LP relaxation of the above 0–1 LP can be written in the following form:

\[
\begin{align*}
\min & \sum_{i \in I} y_i + \sum_{i \in I, j \in J} c_{ij} x_{ij} \\
\text{s.t.} & \sum_{i \in I} x_{ij} \geq 1 \quad (v_j \in J) \\
& y_i - x_{ij} \geq 0 \quad (v_i \in I, j \in J) \\
& y_i \geq 0 \quad (v_i \in I, j \in J) \\
& y_i \geq 0 \quad (v_i \in I).
\end{align*}
\]

The lower bound from this relaxation is typically very tight (e.g., Jain and Vazirani [17], Mahdian et al. [24], Byrka and Aardal [6]). Also, there is an extensive literature on approximation algorithms, i.e., heuristics that have a proven \textit{a priori} performance guarantee (e.g., Jain and Vazirani [17], Mahdian et al. [24], Byrka and Aardal [6]).

A completely different class of primal heuristics consists of those that attempt to exploit good dual solutions. Suppose that a feasible dual vector \( v^* \in Z^d \) has been obtained by one of the dual heuristics mentioned in the previous section. Observe that the quantity \( s_i = f_i - \sum_{j \in J} \max(0, v^*_j - c_{ij}) \), called the ‘slack’ in the dual context, can be viewed as an estimate of the reduced cost of the variable \( y_i \) in the primal. This led Bilde and Krarup [5] and Erlenkotter [11] to propose the following simple primal heuristic: temporarily open a facility at every blocked location, assign each client to the closest open facility, and then close any open facility that does not have any client assigned to it.

When this primal heuristic is applied to small instances, the resulting upper bound is typically quite good. Moreover, if multiple dual solutions are available (for example, if a sequence of dual adjustments is made), then multiple primal solutions can be obtained, thus potentially leading to even better upper bounds [11,19]. A more sophisticated variant of this primal–dual scheme can be found in Hansen et al. [16]. See also Beasley [4] for a similar heuristic scheme, adapted to the Lagrangian setting.

2.2. Heuristics

A wide variety of primal heuristics (i.e., heuristics for producing feasible integral solutions to the 0–1 LP) have been proposed in the literature. In the early literature, the heuristics were all of a simple greedy nature, in which facilities were either iteratively added or dropped. Examples include the ‘Add’ and ‘Bump-and-Shift’ heuristic of Kuehn and Hamburger [22], the local search heuristic of Manne [25], and the ‘Drop’ heuristic of Feldman et al. [12]. Later on, some meta-heuristic approaches were explored, such as genetic algorithms [21] and tabu search [29]. Also, there is an extensive literature on approximation algorithms, i.e., heuristics that have a proven \textit{a priori} performance guarantee (e.g., Jain and Vazirani [17], Mahdian et al. [24], Byrka and Aardal [6]).

The starting point of the algorithm is to sort the \( c_{ij} \) values, for each client \( j \), in non-decreasing order. For \( j = 1, \ldots, n \) and \( k = 1, \ldots, m \), we let \( c_{ij}^k \) denote the \( k \)th cost in the \( j \)th sorted list. We also use the convention \( c_{ij}^{n+1} = \infty \) for \( j = 1, \ldots, n \). For a given dual solution \((v_1, \ldots, v_n)\), we let \( s_i \) denote the slack of the \( i \)th constraint of the form (1). We also let \( k(j) \) denote, for each \( j \), the current minimum value of \( k \) such that \( v_j \leq c_{ij}^k \). A location \( i \) is called \textit{blocked} if \( s_i = 0 \). A client \( j \) is called \textit{blocked} if there exists a blocked location \( i \) such that \( v_j \geq c_{ij} \). If client \( j \) is blocked, then the dual value \( v_j \) cannot be increased.

A high-level description of the algorithm is as follows:

For each client \( j \), do:

Sort the \( c_{ij} \) values in non-decreasing order.

Set \( v_j := c_{ij}^1 \) and \( k(j) := 1 \).

For each location \( i \):

Set \( s_i := f_i \).

Repeat the following until all clients are blocked:

For each unblocked client \( j \), do:

Let \( A_j \) be the minimum of \( s_i \) over all \( i \) for which \( v_j \geq c_{ij} \).

If \( A_j = 0 \), client \( j \) is blocked.
to his approach were suggested by Van Roy and Erlenkotter [32] and Körkel [19]. The most effective of these were an improved branching rule, the generation of more primal solutions, and some rules for eliminating variables on the basis of estimated reduced costs.

A similar scheme was proposed by Beasley [4], but using Lagrangian relaxation to compute lower and upper bounds instead of dual ascent and dual adjustment.

Finally, we mention that some authors have explored exact solution methods based on the idea of converting the SPLP into a pseudo-Boolean optimisation problem [13–15].

This approach seems to work best when applied to very sparse instances. (An SPLP instance is said to be sparse if $c_{ij} = \infty$ for the majority of pairs $i,j$.)

### 3. New dual-based procedures

#### 3.1. Motivation

To our knowledge, an explicit analysis of the running time of the classical dual ascent procedure has not appeared in the literature. Fortunately, the analysis is straightforward, as shown in the following lemma and proposition:

**Lemma 1.** The initial sorting of the assignment costs can be performed in $O(mn \log m)$ time.

**Proof.** The sorting of the assignment costs for an individual client can be performed in $O(m \log m)$ time using heapsort [33]. This can be performed for each of the $n$ clients. □

**Proposition 1.** The classical dual ascent procedure runs in $O(m^2 n)$ time.

**Proof.** The number of times that we encounter a positive $A_j$ value is $O(mn)$, and, each time this happens, we have to update $O(m)$ slacks. The updating of the slacks is the bottleneck of the procedure. □

For very large SPLP instances, this can be rather time-consuming, especially if one wishes to embed the procedure within a branch-and-bound framework. We remark that very large values of $m$ and $n$ can arise in real-life applications. Indeed, large values of $n$ can arise when a continuous location problem is discretised (i.e., when a location problem with an infinite number of locations is approximated by an SPLP instance with a large, but finite, set of locations). Large values of $n$ can arise simply because some companies have thousands of clients.

As for the dual adjustment procedure, we have not seen a formal analysis of its running time. We suspect that it can be implemented to run in $O(m^2 n^2)$ time, but we have not managed to prove it. In any case, its running time is unattractive for very large instances.

These considerations led to our search for faster ascent procedures.

#### 3.2. Enhanced ascent procedure

We now present an enhanced version of the classical dual ascent procedure, which works significantly faster in practice.

A key concept in the enhanced procedure is that of the base level. The base level is the largest value of $k$, with $1 \leq k \leq m$, such that it is feasible to set $v_j$ to $c^k_{ij}$ for all $j$. We have observed that, on instances with large $m$, the dual ascent procedure spends well over half the time incrementing the $k(j)$ until they all reach the base level. (A similar observation was made by Körkel [19].)

Fortunately, we have the following result:

**Lemma 2.** Let $k^*$ denote the base level. One can compute $k^*$ in $O(nk^* \log k^*)$ time.

**Proof.** First, let $k$ be a fixed integer, with $1 \leq k \leq m$. Simply by checking the constraints (1), one can check in $O(nk)$ time whether it is feasible to set the dual variable $v_j$ to $c^k_{ij}$ for each client $j \in J$.

Now, starting with $k=1$ and iteratively doubling $k$, one can find in $O(nk^* \log k^*)$ time a value $r$ such that $2^r \leq k^* \leq \min(m, 2^{r+1})$.

Next, by performing binary search over the interval $[2^r \cdot \min(m, 2^{r+1})]$, one can determine the exact value of $k^*$ in $O(nk^* \log k^*)$ time. □

Since $k^* \leq m$, the time taken to compute the base level is dominated by the initial sorting of the assignment costs (Lemma 1).

Once the base level has been computed by binary search, one can immediately set all dual values to the base level, and then use standard dual ascent to complete the ascent process. This is what we call the enhanced ascent procedure. Note that the time taken after the base level has been found can still be significant, so that the running time of the enhanced procedure remains $O(m^2 n)$. Nevertheless, as we will see, the running time is frequently substantially reduced in practice.

#### 3.3. Fast ascent procedure

When $m$ and $n$ are extremely large, even the enhanced ascent procedure can be impractical. In this section, we present a new ascent procedure, called fast ascent, that runs in $O(m n \log m)$ time. This is the same time as that taken for the initial sorting.

We will need the following lemma:

**Lemma 3.** Suppose that we are given a feasible solution $v^*$ to the condensed dual, and we have already computed the corresponding slacks $s_i$ for all $i \in I$. For any given client $j \in J$, we can compute in $O(m)$ time the maximum amount by which $v_j$ can be increased, while maintaining feasibility.

**Proof.** It suffices to compute, for all $i \in I$, the maximum amount by which $v_i$ can be increased, while maintaining non-negativity of the slack $s_i$. □

Thus, to obtain the desired running time bound, it suffices to ensure that each client’s dual value is increased no more than $O(\log m)$ times. To achieve this, we perform a kind of binary search.

We are now ready to present the fast ascent procedure:

For each client $j$, do:

1. Sort the $c_{ij}$ values in non-decreasing order using heapsort.
2. Compute the base level $k^*$ by binary search, as described above.
3. Set $v_j = c^k_{ij}$ and $k(j) = k^*$ for all $j$, and update the slacks $s_i$ accordingly.
4. Repeat the following until all clients are blocked:
   1. For each unblocked client $j$, do:
      1. Let $A_j$ be the largest amount by which $v_j$ can be increased.
      2. If $A_j = 0$, client $j$ is blocked.
      3. If $v_j + A_j$ is greater than $c^k_{ij}$, find the largest value $k(j)$ such that $c^k_{ij}$ is no larger than $v_j + A_j$.
      4. Set $k(j) = \lceil \log_k + k^* \rceil$.
      5. Set $A_j$ to the difference between $c^k_{ij}$ and $v_j$.
      6. If $A_j > 0$ increase $v_j$ by $A_j$.
      7. Update the slacks $s_i$.

As mentioned above, the time taken for the sorting and for computing the base level is $O(mn \log m)$. Now consider the
remainder of the algorithm. For each client \( j \), the quantity \( v_j + A_j \) is non-increasing, which implies that \( k'(j) \) is non-increasing as well. Since we repeatedly set \( k(j) \) to the mean of \( k(j) \) and \( k'(j) \), the total number of times \( v_j \) is updated is \( O(\log m) \) as desired.

**Remark 1.** Instead of replacing \( k(j) \) with \( \lfloor (k(j) + k'(j))/2 \rfloor \), as described above, we can replace it with

\[
\left\lfloor \frac{(t-1)k(j) + k'(j)}{t} \right\rfloor,
\]

where \( t > 1 \) is an arbitrary constant. The running time remains \( O(mn \log m) \), but the logarithm is to the base \( t/(t-1) \). As \( t \) increases, the fast ascent routine becomes more and more similar to the enhanced routine described in the previous subsection, but the running time increases.

### 4. A new scheme for producing primal solutions

Now we turn our attention to primal heuristics, i.e., heuristic procedures for producing good integer feasible solutions to the SPLP. Since we are dealing with very large instances of the SPLP, we are interested in heuristics that run extremely quickly.

Recall from Section 2.2 that a primal heuristic based on opening ‘blocking’ facilities was described in [5,11]. It is not hard to show that this heuristic, when applied to a single given dual solution, can be implemented to run in \( O(mn \log m) \) time. So, we considered simply applying this heuristic to the dual solution obtained from one of our dual ascent procedures. Our preliminary computational experiments revealed, however, that this approach usually performs very poorly.

We propose to use instead the following simple heuristic scheme, which is based on iteratively ‘dropping’ facilities:

1. Temporarily open a facility at every location (i.e., set \( y^s_i = 1 \) for all \( i \in I \)).
2. Assign each client to its nearest open facility.
3. Sort each location in the sorted list:
   - Evaluate the effect on the cost of closing the facility at that location.
   - If closing the facility would lead to a cost saving, close it.

The following proposition shows that this ‘drop heuristic’ can be implemented so that it runs very quickly:

**Proposition 2.** Suppose that the initial sorting of assignment costs has already been performed (i.e., for each client \( j \), the locations have already been sorted in non-decreasing order of \( c_{ij} \)). Suppose also that the criterion for sorting the locations has been specified in advance. Then the drop heuristic can be implemented to run in \( O(mn + m \log m) \) time.

**Proof.** The first step is to sort the locations according to the specified criterion, which can be performed in \( O(m \log m) \) time.

Now, we construct an array of length \( n \), called \( \text{nearest} \), with the following interpretation. At any stage of the algorithm, if \( \text{nearest}(j) = k \), it means that the closest open facility to client \( j \) is the \( k \)th nearest facility to client \( j \). We also construct another array of length \( n \), called \( \text{second_nearest} \), which is similar to \( \text{nearest} \), but stores the level of the \( \text{second} \) nearest open facility to each client.

At the start of the heuristic, all facilities are open, and each client is assigned to the nearest facility. Therefore we initialise \( \text{nearest}(j) = 1 \) and \( \text{second_nearest}(j) = 2 \) for all \( j \in J \).

We now scan through the list of facilities. For each facility \( i \), we scan through the list of clients. For each client such that \( \text{nearest}[j] = i \), if any, we evaluate the effect on the cost of reassigning client \( j \) to the second closest open facility. If we then decide to close facility \( i \), we scan through the clients again, and update the \( \text{nearest} \) and \( \text{second_nearest} \) arrays.

Now, the largest possible value of any entry in the two arrays is \( m \), and the value of each entry can only increase, not decrease. Therefore the total amount of work spent in scanning and updating arrays is \( O(mn) \).

The crucial choice to be made, of course, is the criterion by which the locations are to be sorted. We have experimented with three different criteria:

1. Sort in non-increasing order of fixed cost \( f_i \).
2. Sort in non-increasing order of \( s_i \), where \( s \) is the vector of slacks obtained at the end of the fast ascent procedure described in the previous subsection.
3. Sort in non-increasing order of \( s_i \), where \( s \) is the vector of slacks obtained at the base level \( k^* \), i.e., when \( v_j = c_{ij}^* \) for all \( j \).

We call these approaches the standard, fast and base drop heuristics, respectively. The results given in the next section indicate that fast drop usually produce better upper bounds than standard and base drop, but that there is no clear winner among standard and base drop.

We have also experimented with the following more sophisticated approach, which we call the multi-drop heuristic: after each major iteration of the fast ascent procedure, the facilities are sorted in non-increasing order of their current slack values, and the drop heuristic is invoked. The best upper bound is then taken over all major iterations. By ‘major iteration’, we mean a single loop through all the clients. The number of major iterations is \( O(mn \log m) \), which ensures that the total time taken by multi-drop is \( O(mn \log m + m \log^2 m) \).

Note that the upper bound given by multi-drop is guaranteed to be at least as good as the best of the upper bounds given by fast and base drop. We will see in the next section that, in fact, all drop heuristics give better upper bounds than the heuristic of [5,11], and multi-drop in particular gives excellent bounds.

### 5. Computational experiments

In this section, we report the results of some computational experiments.

We began by testing the procedures on the three largest instances in the OR library [3], which are taken from Beasley [2]. For these instances, the assignment costs are distances between random points in the plane, but with small random perturbations. All three instances have \( m=100 \) and \( n=1000 \), and the optimal solutions are known for all of them [3].

Table 1 reports, for each instance, the percentage gap between various lower bounds and optimum. The lower bounds were obtained using four different dual ascent procedures: classical, enhanced, fast with \( t=2 \) and fast with \( t=10 \). (Recall that the classical and enhanced ascent procedures yield the same dual solution, and therefore the same lower bound.)

We see that the lower bounds from classical/enhanced ascent are very good, whereas the lower bounds from fast ascent are competitive only when the larger value for the parameter \( t \) is selected.

The running times for the ascent procedures were negligible (less than 15 ms) in every case.

Table 2 reports the results obtained when applying the various primal heuristics to the OR-Lib instances. For each instance, we display the instance name and the average percentage gaps
Table 1
Percentage gaps obtained when applying dual ascent routines to OR-Lib instances with $m=100$ and $n=1000$.

<table>
<thead>
<tr>
<th>Name</th>
<th>% gap of lower bound</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Class/enh.</td>
</tr>
<tr>
<td>a</td>
<td>0.37</td>
</tr>
<tr>
<td>b</td>
<td>1.13</td>
</tr>
<tr>
<td>c</td>
<td>1.11</td>
</tr>
<tr>
<td>Mean</td>
<td>0.87</td>
</tr>
</tbody>
</table>

Table 2
Percentage gaps obtained when applying primal heuristics to OR-Lib instances with $m=100$ and $n=1000$.

<table>
<thead>
<tr>
<th>Name</th>
<th>Block S-drop B-drop F-drop2 F-drop10 M-drop2 M-drop10</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>9.10 12.57 1.11 1.11 0.00 0.00 0.00</td>
</tr>
<tr>
<td>b</td>
<td>14.01 5.55 7.91 6.13 2.82 2.82 1.08</td>
</tr>
<tr>
<td>c</td>
<td>3.78 4.52 3.64 3.72 0.20 0.03 0.03</td>
</tr>
<tr>
<td>Mean</td>
<td>8.96 7.55 4.22 3.65 1.01 0.95 0.37</td>
</tr>
</tbody>
</table>

Table 3
Running times and percentage gaps obtained when applying ascent routines to new large instances.

<table>
<thead>
<tr>
<th>m−n</th>
<th>Running time (s)</th>
<th>% gap of lower bound</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Classical</td>
<td>Enhanced</td>
</tr>
<tr>
<td>500</td>
<td>0.076</td>
<td>0.061</td>
</tr>
<tr>
<td>1000</td>
<td>0.411</td>
<td>0.321</td>
</tr>
<tr>
<td>1500</td>
<td>1.150</td>
<td>0.916</td>
</tr>
<tr>
<td>2000</td>
<td>2.530</td>
<td>1.998</td>
</tr>
<tr>
<td>2500</td>
<td>5.724</td>
<td>4.756</td>
</tr>
<tr>
<td>3000</td>
<td>9.978</td>
<td>8.227</td>
</tr>
<tr>
<td>Mean</td>
<td>3.312</td>
<td>2.714</td>
</tr>
</tbody>
</table>

Table 4
Percentage gaps obtained when applying primal heuristics to new large instances.

<table>
<thead>
<tr>
<th>m−n</th>
<th>Block S-drop B-drop F-drop2 F-drop10 M-drop2 M-drop10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20.06  7.96  7.68  2.21  1.77  1.29  0.52</td>
</tr>
<tr>
<td>1000</td>
<td>21.40  8.25  11.08 1.78  2.05  0.81  0.79</td>
</tr>
<tr>
<td>1500</td>
<td>25.33  9.11  13.40 3.09  2.59  1.21  0.62</td>
</tr>
<tr>
<td>2000</td>
<td>27.62  8.33  11.90 2.71  1.94  1.37  0.68</td>
</tr>
<tr>
<td>2500</td>
<td>27.73  8.74  17.79 2.58  1.40  1.28  0.62</td>
</tr>
<tr>
<td>3000</td>
<td>26.84  9.36  14.19 2.59  1.80  1.42  0.72</td>
</tr>
<tr>
<td>Mean</td>
<td>24.83  8.63  12.67 2.49  1.93  1.23  0.66</td>
</tr>
</tbody>
</table>

Table 5
Running times and percentage gaps obtained when testing larger instances.

<table>
<thead>
<tr>
<th>Size</th>
<th>Running time (s)</th>
<th>Total % gap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Classical</td>
<td>Enhanced</td>
</tr>
<tr>
<td>5000</td>
<td>31.853</td>
<td>23.269</td>
</tr>
<tr>
<td>7500</td>
<td>98.304</td>
<td>72.954</td>
</tr>
<tr>
<td>10000</td>
<td>192.779</td>
<td>140.466</td>
</tr>
<tr>
<td>12500</td>
<td>318.058</td>
<td>234.707</td>
</tr>
<tr>
<td>15000</td>
<td>494.387</td>
<td>353.172</td>
</tr>
<tr>
<td>Mean</td>
<td>227.076</td>
<td>164.914</td>
</tr>
</tbody>
</table>
hand, we think it is promising that, using our new methods, one can obtain lower and upper bounds differing by under 4% in a couple of minutes even when \( m \) and \( n \) are as large as 15,000.

Finally, we remark that we would have liked to run our algorithms on even larger instances, but ran into memory limitations.

6. Conclusion

Large-scale SPLP instances can arise when there are thousands of clients, or when a continuous location problem is discretised. We have shown that it is possible to compute quickly reasonably good lower bounds, and very good upper bounds, for such instances. In a subsequent paper, we will show how to embed our fast lower- and upper-bounding procedures in a sophisticated scheme for solving large-scale instances to proven (near-)optimality.

We end the paper by making some remarks about sparsity (see Section 2.3 for a definition). For a given client \( j \), let \( d_j \) denote the number of locations for which \( c_{ij} \) is finite. Also define:

\[
\sigma = \sum_{j \in J} d_j \quad \text{and} \quad \overline{d} = \max_{j \in J} d_j.
\]

It is not hard to implement the classical and fast ascent algorithms so that they run in \( O(\sigma^2) \) and \( O(\sigma \log \overline{d}) \) time, respectively. Moreover, sparse instances consume significantly less memory. For this reason, we believe that our new algorithms could be used to tackle sparse instances of extremely large size.

Acknowledgements

The first author was supported in part by the Engineering and Physical Sciences Research Council under Grant EP/D072662/1. Thanks are due to an anonymous referee for some helpful suggestions.

References