A Computational Geometry-based Local Search Algorithm for Planar Location Problems

Hadrien Cambazard, Deepak Mehta, Barry O'Sullivan, and Luis Quesada

CTVR,Cork Constraint Computation Centre, University College Cork, Ireland
{h.cambazard|d.mehta|b.osullivan|l.quesada}@4c.ucc.ie

Abstract. Constraint-based local search is an important paradigm in the field of constraint programming, particularly when considering very large optimisation problems. In this paper we are motivated by applications in areas such as telecommunications network design, warehouse location and other problems in which we wish to select an optimal set of locations from a two dimensional plane. The problems we are interested in are so large that they are ideal candidates for constraint-based local search methods. In an optimisation setting it is important that a local search algorithm has access to an incremental way of evaluating the value of its local moves. In the case of two dimensional plane problems, we can often achieve incrementality by exploiting computational geometry. In this paper we present a novel approach to solving a class of placement problems for which Voronoi cell computation can provide an efficient form of incrementality.

1 Introduction

We are motivated by applications in areas such as telecommunications network design, warehouse location and other problems in which we wish to select an optimal set of locations from a two dimensional plane. Local search algorithms have proved very efficient in this area and we are interested in the design of efficient incremental mechanisms involving closest points problems. We investigate a restriction of this class of problems in which the cost of allocating a client to a facility is assumed to be proportional to Euclidean distance. This assumption comes from an application focused on the design of resilient long-reach passive optical networks [**?**]. The core problem in this application is referred to as the Single Coverage Problem and defined as follows:

Definition 1 (The Single Coverage problem). *Given a constant κ and a set $S = \{p_i | 1 \leq i \leq m\}$ of points in a two dimensional space where each point p_i is associated with a weight b_i, the Single Coverage problem (SC) is to decide whether there exists a set $W \subset S$ of cardinality p such that the weighted sum of the distances from the points in $S - W$ to their closest points in W is less than or equal to κ, i.e., $\sum_{p_i \in S-W} b_i \times \min_{p_j \in W} d_{ij} \leq \kappa$.*

S is referred to as the set of clients and W as the set of facilities. Moreover, d_{ij} denotes the Euclidean distance between the points p_i and p_j. The single coverage problem is strongly related to location problems in a plane such as the uncapacitated warehouse location problem (UWLP) [?]. Typically, the SC differs from the UWLP in three respects: the transportation cost is proportional to the Euclidean distance (although this is usually the case in practice, UWLP does not make any assumption on the transportation costs); there is no fixed cost of opening a warehouse; and the number of opened warehouses is bounded by p. Another, and even closer, problem is the p-median problem. This latter problem mainly differs from the SC in that it is not limited to 2 dimensions. The set of points or observations are usually described using more than two attributes. The similarity or dissimilarity measure used by p-median problems is not restricted to the Euclidean distance. A vast literature deals with location problems in the plane, the survey [?] presents spatial clustering problems.

State of the art algorithms for solving large-scale p-median (k-medoid) or uncapacitated warehouse location problems rely on local search. Genetic algorithms [?] and Tabu search in particular have been very successful [?]. We describe in Sections **??** and **??** a local search algorithm for the single coverage problem directly inspired by this previous work. In particular the incrementality of the neighborhood has been described in details by [?,?,?,?]. In Section **??**, we present new ideas for achieving a better complexity for the incremental algorithms presented in the previous section when the cost is proportional to Euclidean distance.

2 The Tabu Search

We denote the set of current facilities by W and the current set of clients or nodes by $C = S - W$. We use i as an index for nodes/clients whereas j refers to facilities.

Neighborhood. The neighborhood is defined by moving a facility from one location to another. This move is performed in two steps, a facility is closed first and another one is opened. Closing a facility involves removing a point p_j from W. Opening a facility involves adding a point p_i from C to W. The objective function is evaluated by ensuring that clients are always connected to their closest facilities. This invariant is maintained throughout the search. This neighborhood is used by the main algorithm for k-medoid, namely the Partitioning Around Medoids (PAM) algorithm [?]. However, PAM does not include an incremental evaluation of the moves. Therefore we distinguish the two steps (opening and closing) to develop an incremental scheme using [?] for each of these steps separately. This incremental evaluation of the moves is achieved by maintaining the following data structures (Section **??** presents the algorithm):

- Δ_i^+ is the variation of cost due to adding p_i in W (opening a facility).
- Δ_i^- is the variation of cost due to removing p_i from W (closing a facility).

Search. The initial p facilities are chosen randomly. The tabu mechanism is very simple. It prevents a point that was a facility in the last t iterations (where t is the length of the tabu-list) from becoming a facility again. The tabu-list is denoted T in Algorithm **??**. If there are several equally good improving moves, then the move to perform is picked randomly. If no improving move exists, the facility to close is chosen randomly and the new best location for this facility is opened.

Algorithm **??** presents the general scheme of the tabu search. It assumes that two methods are available for opening and closing a facility (resp. OpenFacility and CloseFacility) while maintaining incrementally the value of the objective function (denoted obj) and Δ^+. It is not useful to maintain Δ^- incrementally for this specific neighborhood where all opened nodes are closed to evaluate each move (Line 8 is only using Δ^+). Δ^- would be useful in a more general context when the closing and opening operations can be done independently, e.g., warehouse location.

Algorithm 1 TabuSearch()

1. `Initialize` W `randomly,` $C = S - W$`,`
3. **While** `(end condition not reached)`
4. $p_j^* = -1$`,` $bestDelta = \infty$`,` $cobj = obj$
5.   **For each** $p_j \in W - T$
6.   `CloseFacility(`p_j`) // this updates obj and all` Δ_i^+ `incrementally`
7. $p_{ibest} = \arg\min_{\{p_i \in C-T\}}(\Delta_i^+)$
8.   **If** $(\Delta_{ibest}^+ + (cobj - obj)) < bestDelta$
9. $p_j^* = p_j$`,` $bestDelta = \Delta_{ibest}^+ + (cobj - obj)$
10.   `OpenFacility(`p_j`) // this updates obj and all` Δ_i^+ `incrementally`
11.   **If** `(bestDelta > 0)`
12. p_j^* `= a random point in` $W - T$
13.   `CloseFacility(`p_j^*`)`
14.   `OpenFacility` $(\arg\min_{\{p_i \in C-T\}}(\Delta_i^+))$
15.   `update tabu list` T

This algorithm is the PAM algorithm enhanced with the incremental mechanisms, and the tabu metaheuristic, introduced in warehouse location for a similar neighborhood. We believe it is the best starting point for our application and purpose.

3 Incremental Neighborhood

Maintaining the objective function incrementally is a key element for the efficiency of local search algorithms [**?**]. When moving a facility from one location to another, only a small subset of the clients are reallocated. Clients that have lost their current closest facilities and clients that have gained a new closest facility. The cost is not affected by the other clients. Optimal incremental algorithms have been published for opening and closing operations in the context of warehouse location [?,?,?]. We present this approach in detail as we will build upon it. The data structures needed to develop the incremental approach and maintain Δ^+ and Δ^- are the following:

1. a_i^1 is the closest facility to client i so that $a_i^1 = \arg\min_{p_j \in W} d_{ij}$.
2. a_i^2 is the second closest facility to client i so that $a_i^2 = \arg\min_{p_j \in W | p_j \neq a_i^1} d_{ij}$.
3. Q_i is a priority queue storing all the current facilities W ordered by increasing distance from i. Consequently a_i^1 and a_i^2 are the first two elements of this queue.

The variations of the objective function due to closing and opening a facility i are initialized as follows:

$$\Delta_i^- = \sum_{p_j \in S | a_j^1 = p_i} b_j \times (d_{j,a_j^2} - d_{ji}) \tag{1}$$

$$\Delta_i^+ = -\sum_{p_j \in C} b_j \times \max(0, d_{j,a_j^1} - d_{ji}) \tag{2}$$

When closing a facility i, we need to add to the objective function the cost of disconnecting each point connected to p_i and re-connecting them to their second closest facility. Therefore, we add d_{j,a_j^2} and remove d_{ji}. Similarly when opening a facility i, each point p_j of C that is closer to this new facility than to its current closest facility ($d_{j,a_j^1} > d_{ji}$) needs to be de-connected and re-connected decreasing the objective function by the quantity $d_{j,a_j^1} - d_{ji}$. Notice that Δ_i^+ is at most zero (opening never degrades the overall objective function) and Δ_i^- is at least zero (closing always degrades the overall objective function). In what follows we will refer to $d_{j,a_j^2} - d_{ji}$ as the contribution of p_j by which Δ_i^- increases. Similarly, we will say that $d_{j,a_j^1} - d_{ji}$ is the contribution of p_j by which Δ_i^+ decreases. For the sake of clarity we will assume that all the b_i are equal to 1 in the following, the algorithms presented remain identical with general weights. It is simply a matter of multiplying the distance by the weight.

3.1 Closing a Facility

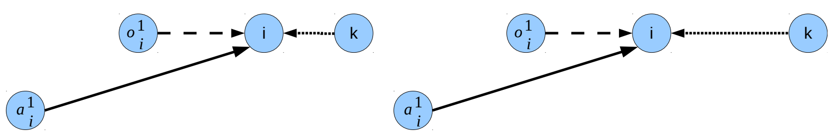
Algorithm **??** shows the incremental maintenance of the data structures (in particular Δ^+ and Δ^- which are used to select the best move) when closing a given facility p_j.

For each client p_i of C, the priority queue, Q_i, is maintained (Line 2). The previous values of the closest and second closest facilities of p_i are saved in o_i^1 and o_i^2 respectively (Line 3). The closest and second closest facilities of p_i are then updated in a_i^1 and a_i^2 using Q_i respectively (Line 4). Lines 5 to 11 deal with the update of Δ^+ and Δ^-. When a facility p_j is closed either the closest facility of p_i can change, or the second closest facility of p_i can change or none

Algorithm 2 CLOSEFACILITY(p_j)

1. **For each** $p_i \in S$ **do**
2.   `remove` p_j `from` Q_i
3. $o_i^1 = a_i^1$, $o_i^2 = a_i^2$
4. $a_i^1 = Q_i.getFirst()$, $a_i^2 = Q_i.getSecond()$
5.   **If** $(o_i^1 \neq a_i^1) \vee (o_i^2 \neq a_i^2)$ **do**
6. $\Delta^-_{o_i^1} = \Delta^-_{o_i^1} - (d_{i,o_i^2} - d_{i,o_i^1})$
7. $\Delta^-_{a_i^1} = \Delta^-_{a_i^1} + (d_{i,a_i^2} - d_{i,a_i^1})$
8.   **If** $(o_i^1 \neq a_i^1)$ **do**
9.   **For each** $p_k \in S$ `such that` $d_{i,k} < d_{i,a_i^1}$ **do**
10.   **If** $(d_{i,k} < d_{i,o_i^1})$ $\Delta_k^+ = \Delta_k^+ - (d_{i,a_i^1} - d_{i,o_i^1})$
11.   **Else** $\Delta_k^+ = \Delta_k^+ - (d_{i,a_i^1} - d_{i,k})$
12. $W = W - \{p_j\}$, $C = C \cup \{p_j\}$

of them changes. Only the points p_i which have a new closest or second closest facility can trigger the changes of the values of Δ^-. Line 6 simply removes from Δ^- the previous contribution of p_i to its old closest facility and Line 7 adds the new contribution of p_i to its new closest facility.



(a) i was already contributing to Δ_k^+ (b) i was not contributing to Δ_k^+

Fig. 1. The two scenarios for a node i that contributes to Δ_k^+ in Algorithm **??**. The old association is drawn with a dashed line, the new one is drawn with a continuous line and the potential association is drawn with a dotted line.

Lines (8–11) update Δ_k^+ with respect to the contribution of i. From Equation (**??**) recall that the contribution of i for Δ_k^+ can change only when the closest facility of i changes, i.e., when $o_i^1 \neq a_i^1$ (Line 8) and when $d_{i,k} < d_{i,a_i^1}$ (Line 9). Therefore, the iteration is performed on a pre-computed list of points k sorted by distance from i as long as the criteria $d_{i,k} < d_{i,a_i^1}$ holds. If k is closer to i than o_i^1 (i.e., $d_{i,k} < d_{i,o_i^1}$) as shown in Figure **??** then it follows that the contribution of i to Δ_k^+ is non-zero. Therefore, the previous contribution, $d_{i,o_i^1} - d_{i,k}$, should be replaced by the new contribution $d_{i,a_i^1} - d_{i,k}$, which is effectively the difference between $d_{i,a_i^1} - d_{i,o_i^1}$ (Line 10). If k is not closer to i than o_i^1 as shown in Figure **??** then the contribution of i to Δ_k^+ is 0. Therefore, Δ_k^+ is updated with the new contribution of i (Line 11).

We now consider the complexity of Algorithm **??** for closing a facility. Updating one priority queue is done in $\mathcal{O}(\log(p))$ (implementation based on heaps) and this has to be done for all points thus Lines 1-2 imply a $\mathcal{O}(m \log(p))$ complexity. Updating Δ^- is then done in constant time whereas updating Δ^+ is achieved in time linear in the number of points p_i whose closest facility has changed. This complexity is optimal as it is necessary to consider all the updated points and they cannot cancel out since d_{i,a_i^1} is always increasing ($d_{i,a_i^1} \geq d_{i,o_i^1}$). The pre-computed lists of points sorted by distance from any other points (Line 9) requires $O(m^2)$ space which can be an issue when solving very large problems. In practice [**?**] reports that the cost is dominated by the update of the priority queues. The update of Δ^+ is costly but only done on a small subset of S whereas the priority queues have to be updated for the $m - p$ points.

To help a reader understand the correctness of the algorithm, we remark that the following invariant is always maintained: Δ_i^- *does not decrease when closing a facility* $j \neq i$. If o_i^1 is equal to a_i^1 then it means that p_j was the second closest facility of p_i, which is now changed to a_i^2. The same entry of Δ^- is updated in Lines 6 and 7 by removing the previous contribution of p_i and adding the new contribution of p_i respectively. As $(d_{i,o_i^2} - d_{i,o_i^1})$ is at most $(d_{i,a_i^2} - d_{i,a_i^1})$ the invariant is maintained. If o_i^1 is not equal to a_i^1 then the invariant is trivially maintained since o_i^1 is equal to the closing facility p_j. Notice that Δ_j^- is set to zero after the execution of the for loop since the contributions of all those p_i that were contributing to Δ_j^- are removed in Line 6 when o_i^1 is equal to j (i.e., when o_i^1 is not equal to a_i^1).

3.2 Opening a Facility

Algorithm **??** shows the incremental maintenance of the data structures (in particular Δ^+ and Δ^- which are used to select the best move) when opening a given facility p_j. This algorithm is included here to give a complete picture and for the sake of reproducibility. Notice that in the case of opening a facility we always have $d_{i,o_i^1} \geq d_{i,a_i^1}$.

Algorithm 3 OPENFACILITY(p_j)

1. **For each** $p_i \in S$ **do**
2.   add p_j to Q_i
3 $\rightarrow$ 7.   identical to Algorithm ??
8.   **If** $(o_i^1 \neq a_i^1)$ **do**
9.   **For each** $p_k \in S$ such that $d_{i,k} < d_{i,o_i^1}$ **do**
10.   **If** $(d_{i,k} < d_{i,a_i^1})$ $\Delta_k^+ = \Delta_k^+ + (d_{i,o_i^1} - d_{i,a_i^1})$
11.   **Else** $\Delta_k^+ = \Delta_k^+ + (d_{i,o_i^1} - d_{i,k})$
12. $W = W \cup \{p_j\}$, $C = C - \{p_j\}$

The update of Δ^- is identical to Algorithm **??**. The update of Δ^+ is very similar. As mentioned before the contribution of p_i to Δ^+ only needs to be updated when d_{i,a_i^1} is updated (i.e., when $(o_i^1 \neq a_i^1)$). However, in this case the contribution of p_i to a given Δ_k^+ is reduced either partially or completely since a node is being opened. Line 10 refers to the case where p_i remains as a contributor. In this case we just update its contribution by taking into account that a_i^1 is $d_{i,o_i^1} - d_{i,a_i^1}$ closer than d_{i,o_i^1}. In Line 11 we remove the contribution of p_i completely. Finally, in Line 12, W and C are updated accordingly.

4 A New Incremental Algorithm

The incremental algorithm presented in the previous section is dominated by the $O(m \log(p))$ cost of updating the priority queues. In practice very few points of S are likely to have a new closest or second closest facility. The left part of Figure **??** shows an example of opening a new facility p_j. Facilities are indicated by plain circles and points by crosses. The points for which p_j is the new closest facility are shown in squares whereas the points for which p_j is the new second closest facility are shown in circles. Only a very small number of points of the m points of S are affected.

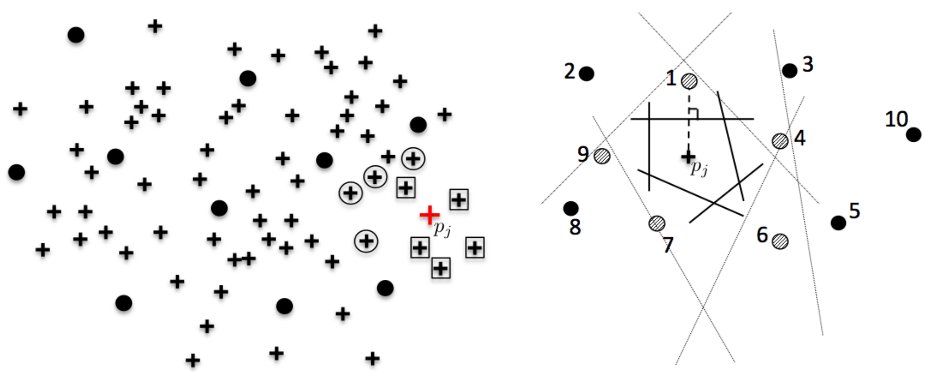


Fig. 2. On the left: Example of opening a facility p_j. Facilities are shown as plain circles, points as crosses and the points having p_j as their closest (resp. second closest) facility are shown in a square (resp. a circle). On the right: Example of the Voronoi cell of p_j ($\mathcal{V}(p_j)$). The boundary of the cell is indicated by the dashed nodes so $\mathcal{B}(p_j) = \{1, 4, 6, 7, 9\}$.

In this paper we focus on approaches that do not maintain the priority queues Q_i. The set of points for which a_i^1 and a_i^2 need to be maintained, is computed directly using computational geometry techniques.

We start with a simple approach. We define the radius r_j of a facility j as the maximum distance between the facility and any of its points that it covers.

The radius of each facility can be maintained easily. If a new facility j is opened then the closest and the second closest of only those points i that are within the reach of $\max_{j \in W}(r_j)$ may change. Using the sorted list of nodes i by increasing distance from node j, we only have to iterate over those points i for which $d_{i,j} \leq \max_{j \in W}(r_j)$ rather the complete set S.

This approach already takes advantage of Euclidean distance and we will see in the following how the space and time complexities of Algorithms **??** and **??** can be improved with computational geometry techniques. Closest point problems are common in computational geometry [**?**,**?**]. A strongly related work is [**?**] which relies on triangulation to speed up the PAM algorithm but does not present complexity results and ignore the optimal incremental schemes that have been developed to improve PAM [**?**,**?**,**?**,**?**]. A more relevant reference is [**?**] which proposes to improve the k-means algorithm by using geometric reasoning based on kd-trees to speed-up the allocation of each point to its closest cluster. Our work is dedicated to the p-median/warehouse location problem rather than k-means and the proposed method tries to build upon known optimal incremental algorithms to improve them in the context of Euclidean distances.

4.1 The Closest Points to a New Facility

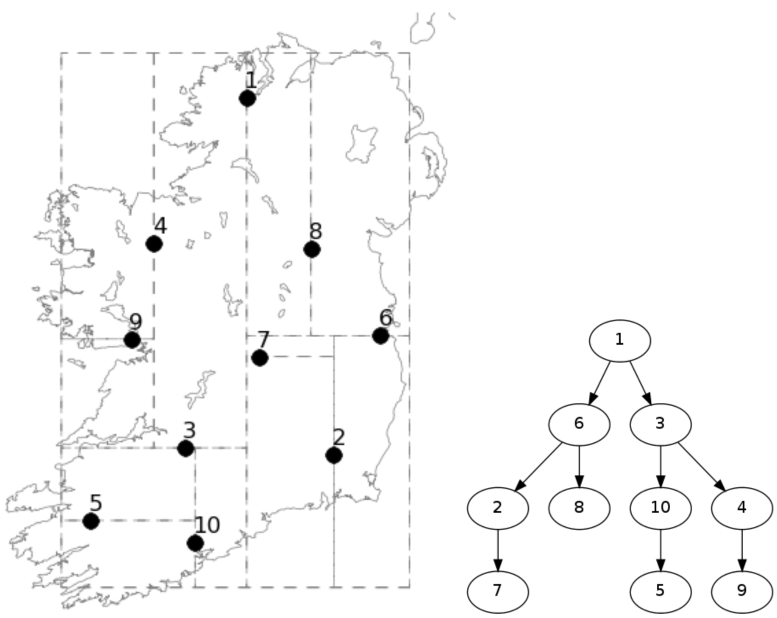
Firstly we focus on updating a_i^1 when opening a new facility. The question we would like to answer efficiently is: *determine the points in S which are closer to a given point p_j (the facility we would like to open) than to any other points of a given specific set (the facilities W).* This set is precisely characterized by the *Voronoi cell* [**?**] of p_j regarding W denoted $\mathcal{V}(p_j)$. A point q lies in $\mathcal{V}(p_j)$ if and only if $dist(q, p_j) < dist(q, p_k)$ for all $p_k \in W$. The right part of Figure **??** shows how a Voronoi cell is built. For any two points p_j and p_k we can define the bisector (see [**?**] chapter 7) as the perpendicular bisector of the line segment $\overline{p_j p_k}$. This bisector splits the plane into two half-planes: one containing the point p_j and another containing the point p_k. $\mathcal{V}(p_j)$ can be seen as the intersection of the half-planes containing p_j obtained by bisecting $\overline{p_j p_k}$ for all $p_k \in W$.

Definition 2 (Boundary of p_j). *The boundary of p_j, $\mathcal{B}(p_j)$, is the set of facilities p_k such that the bisector of $\overline{p_j p_k}$ coincides with one of the line segments of the Voronoi cell of p_j.*

Computing one Voronoi cell is based on computing the intersection of p half-planes which can be done in $O(p\,log(p))$ [**?**]. This however does not give us the actual points of S contained in the cell. We propose two approaches to compute the actual points of S, the first one is very simple but requires $O(m^2)$ space complexity while the second one remains in $O(m)$ in space.

Approach based on the radius: The first approach does not require any special data structure. It is based on the upper bound on the distance between the newly opened facility j and the nodes which will have j as their facility. The Voronoi cell of p_j is a convex polygon which is associated with a set of (corner) points. The minimum and the maximum distances between j and any of the corner points of the Voronoi cell is denoted by r_{min} and r_{max} respectively. Any point whose distance from j is less than r_{min} will definitely have j as its new new facility. Any point whose distance from j is more than r_{max} will not be affected by the new facility. Any point whose distance from j is between r_{max} and r_{min} could possibly be affected. Therefore one has to iterate over all the points i whose distance from j is less than or equal to r_{max}. This is easy if we have access to the points sorted in the increasing distance from j which requires $O(m^2)$ space.

Approach based on a kd-tree: The second approach is based on the use of a common data structure in computational geometry, namely, a kd-tree [**?**]. A kd-tree for two dimensional data points is a binary tree data structure where at each node the space is partitioned horizontally or vertically. The two children of each node correspond to two regions of the space. An example of such a tree is shown on Figure **??**. The kd-tree is built once and contains the points of S, the subdi-



(a) kd tree partition (b) kd tree

Fig. 3. Example of a kd tree based on a distribution of telecommunications exchange sites across the Irish territory.

vision of the space is made alternatively on the x and y coordinates of a point of S and continues until all regions contain no more than one point. A balanced kd-tree (a tree such that each leaf node is more or less at the same distance to the root) can be easily built by inserting the points in a specific order (simply by choosing at each step the median of the points sorted on the corresponding coordinate). In a balanced kd-tree, obtaining one point of S contained in a rectangular area (a range query parallel to the axis used for partitioning the space in the kd-tree) can be done in $O(\sqrt{m})$ and finding all points of S contained in a rectangular area costs $O(\sqrt{m}+k)$, where k is the number of points in the corresponding area [**?**]. The tree is traversed as long as the region of a node intersects the area. When a leaf is reached, the corresponding point is added in the list of points to return. Similarly, when the area fully contains the region of a node, all points contained in the subtree rooted at this node are added in the answer to the query.

This algorithm applies if the area is a convex polygon such as the Voronoi cell. In this case, checking the intersection with the rectangular region of a node is done in $O(h)$ where h is the size of the boundary of the cell so $h = |\mathcal{B}(p_j)|$. However, in this latter case, the $O(\sqrt{m})$ is not guaranteed. For expressing the complexity, we will consider the enclosing rectangle of the Voronoi cell as the query. So let k be the number of points in the cell, the Voronoi cell can be enclosed in a rectangle containing $k^{'}$ points ($k^{'} > k$) in which case the overall complexity is $O(plog(p) + \sqrt{m} + k^{'})$. In practice we apply the algorithm using the Voronoi cell itself to obtain more pruning in the tree.

4.2 Updating the Two Closest Points when Opening a Facility

We now focus on updating a_i^1 and a_i^2 when opening a new facility. We extend the previous idea to find the set of points which have either a new closest or a new second closest facility. The question we would like to answer efficiently is now to *determine the points in S for which a given point, p_j (the facility we would like to open), is one of their two closest neighbors regarding a given specific set (the facilities W).* Determining such a set exactly is slightly harder since the points of the set may not necessarily be enclosed in a convex polygon. Characterizing such a set will involve the computation of the Voronoi cell of each facility of $\mathcal{B}(j)$, which will increase the complexity.

We generalize the previous ideas so that the same scheme applies by replacing the concept of Voronoi cell with a set $\mathcal{V}^{'}(p_j)$ containing the set of points q for which p_j is closer than their second closest neighbor in W. In order to do so we suggest a simple convex approximation based on the concept of *Extended Voronoi cell.*

Definition 3 (Extended Voronoi cell). *Given a point p_j, the extended Voronoi cell $\mathcal{V}_2(p_j)$ is defined as the Voronoi cell of p_j associated with the set of facilities $W - \mathcal{B}(p_j)$.*

Figure **??** illustrates an extended Voronoi cell. Similarly the concept of boundary can be extended and we will denote $\mathcal{B}_2(p_j)$ the boundary of the extended Voronoi cell of p_j.

Lemma 1 ($\mathcal{V}^{'}(p_j) \subseteq \mathcal{V}_2(p_j)$).

Proof. Consider a point q outside of $\mathcal{V}_2(p_j)$. q is closer to a facility $p_k \in W - \mathcal{B}(p_j)$ than to p_j because $\mathcal{V}_2(p_j)$ is the Voronoi cell regarding $W - \mathcal{B}(p_j)$. q is also necessarily closer to a point of $\mathcal{B}(p_j)$ than to p_j since p_j does not belong to $\mathcal{B}(p_j)$. Thus p_j cannot be one of the two closest neighbors of q. $\square$

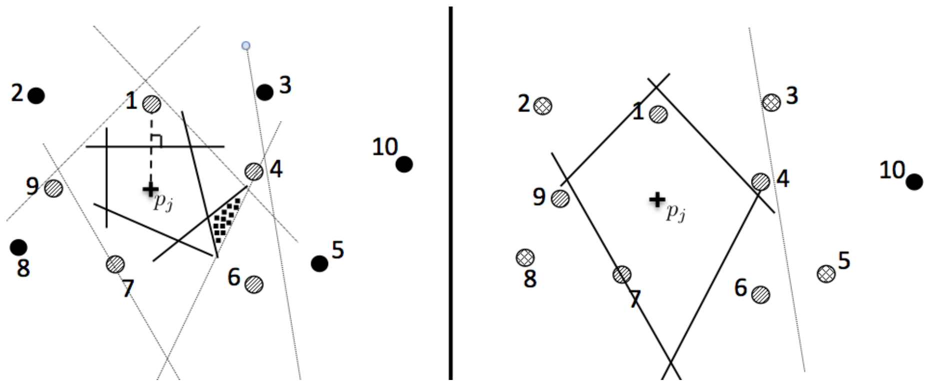


Fig. 4. On the left : Example of $\mathcal{V}(p_j)$ the Voronoi cell of p_j, the boundary of cell is defined by the dashed nodes so $\mathcal{B}(p_j) = \{1, 4, 6, 7, 9\}$. On the right: Example of $\mathcal{V}_2(p_j)$ the extended Voronoi cell of p_j whose boundary $\mathcal{B}_2(p_j) = \{2, 3, 5, 8\}$.

Notice that $\mathcal{V}^{'}(p_j) \neq \mathcal{V}_2(p_j)$. For example in Figure **??** (left), the area paved with squares within $\mathcal{V}_2(p_j)$ contains points that are closer to 4 and 6 than to p_j. Using an exact characterization of $\mathcal{V}^{'}(p_j)$ should be possible but would change the algorithmic significantly.

4.3 Updating the Two Closest Points when Closing a Facility

We examine in this part how to update a_i^1 and a_i^2 when closing a facility p_j. Similar to the previous case, the set of points which have p_j as their closest or second closest facility can be computed using the extended Voronoi cell of p_j. In this case however we can assume that we maintain the set of points connected to p_j in a dedicated data structure, e.g. a list. When closing p_j, the closest or second closest facility of these points have to be updated. A simple solution would be to iterate over the current opened facilities W to find the two closest. Alternatively, this is exactly a 2-nearest neighbors problem. One seeks to quickly identify the two nearest neighbors in W of the points that were connected to p_j. The k-nearest neighbors is a classic problem in machine learning [**?**] and efficient implementations rely on the use of kd-trees [**?**,**?**]. Assuming that we maintain a kd-tree of the set W, finding the two nearest neighbors of a given point can be done efficiently in a balanced kd-tree. The worst-case complexity remains $O(p)$ as it is easy to construct examples where all the leaves of the tree will have to be checked. The complexity analysis presented in [**?**] reports that the expected number of nodes inspected in the tree is in $O(log(p))$. The nearest neighbor algorithm proceeds in two steps. Firstly, an initial candidate point is found in the leaf containing the target point p_i (by diving in the tree). The closest point has to be in a circle centered in p_i and with a radius defined by the distance to the initial candidate found. Secondly, we backtrack in the tree by pruning all regions not intersecting the circle (the radius of the circle is updated each time a new candidate is found). The initial dive is in $O(log(p))$ and the amount of backtracking required in the tree is then independent of p thus an overall $O(log(p))$ expected complexity [**?**].

4.4 Updating Algorithms ?? and ??

The complexity reported for the following algorithms does not include the complexity due to maintaining Δ^- and Δ^+ which is optimal [**?**] and linear in the number of changes of closest or second closest.

We introduce three additional data structures:

1. S_j, corresponding to the list of nodes for which facility p_j is either the closest or second closest facility.
2. KW is a kd-tree of the set W of facilities. KW is therefore dynamic and must be updated when closing/opening facilities.
3. KS is a kd-tree of the set S of nodes. KS is static and pre-computed initially.

Algorithm **??** is the new version of Algorithm **??** taking advantage of the ideas based on computational geometry. The extended Voronoi cell of the facility opened is computed first (Line 1) and the points contained in the cell (S_2) are extracted using the kd-tree KS of S (Line 2). The loop over all the points of S is replaced by a loop over the points contained in S_2. The closest or second-closest facility of p_i might now be p_j but this update takes constant time. So does the update of the S_j data structure (S_j is useful for Algorithm **??** when closing a facility). Finally, the incremental maintenance of Δ^- and Δ^+ remain unchanged.

Algorithm 4 OpenFacility2(p_j)

1. compute $\mathcal{V}_2(p_j)$
2. compute the set of points S_2 in $\mathcal{V}_2(p_j)$ using the kd-tree KS of S
3. **For each** $p_i \in S_2$ **do**
4. $\quad o_i^1 = a_i^1$, $o_i^2 = a_i^2$
5. $\quad a_i^1 = \arg\min_{p_k \in \{o_i^1, p_j\}} d_{i,k}$, $a_i^2 = \arg\min_{p_k \in \{o_i^1, o_i^2, p_j\} - \{a_i^1\}} d_{i,k}$
6. $\quad$ **If** ($o_i^1 \neq a_i^1 \vee o_i^2 \neq a_i^2$) $S_{p_j} = S_{p_j} \cup \{p_i\}$, $S_{o_i^2} = S_{o_i^2} - \{p_i\}$

7 $\rightarrow$ 14. identical to Lines 5-12 Algorithm **??**

15. add p_j in the kd-tree KW of the facilities

Line 1 takes $O(plog(p))$, Line 2 (assuming we are using the enclosing rectangle of the Voronoi cell) takes $O(\sqrt{m}+k^{'})$. $k^{'}$ is the number of points in the enclosing rectangle so it is greater than the number of points contained in the cell ($k^{'} \geq k$). Finally Line 5 is performed in $O(1)$ and the update of KW (line 15) is done in $O(log(p))$. The complexity of Algorithm **??** is overall $O(plog(p) + \sqrt{m} + k^{'})$. We recall that the complexity of the previous incremental algorithm is dominated by the $O(mlog(p))$ factor which involves examining systematically all the m points. Algorithm **??** does not have this drawback as m does not appear directly in the complexity but only in a worst case where $k^{'} = m$. In practice, we expect $k^{'}$ to be much smaller than m in average.

Similarly, Algorithm **??** is the new version of Algorithm **??**. The list S_j is used to iterate over the points which had p_j as their closest or second closest. The only difference is that the update of the a_i^1 and a_i^2 is done by using the kd-tree of the facilities KW since we no longer maintain the priority queues. The worst-case complexity of the nearest neighbors search in a balanced kd-tree is in $O(p)$ but its expected complexity is $O(log(p))$. Note that KW has to be re-balanced from time to time to guarantee this complexity. The update of the kd-tree KW is done in Line 1 and takes $O(log(p))$ so that the overall expected complexity for closing a facility is $O(klog(p))$ with $|S_j| = k$.

Algorithm 5 CLOSEFACILITY2(p_j)

1. `remove` p_j `from` KW
2. **For each** $p_i \in S_j$ **do**
3. $o_i^1 = a_i^1$, $o_i^2 = a_i^2$
4. `update` a_i^1, a_i^2 `using a 2-nearest neighbors search in` KW
5. **If** ($o_i^1 \neq a_i^1 \wedge o_i^2 \neq a_i^2$) $S_{a_i^1} = S_{a_i^1} \cup \{p_i\}$
6. **Else** ($o_i^2 \neq a_i^2$) $S_{a_i^2} = S_{a_i^2} \cup \{p_i\}$

7 $\rightarrow$ 13. `identical to Lines 5-12 of Algorithm` **??**

14. $S_j = \emptyset$

Lines 9 of Algorithms **??** and **??** (which are identical to lines 9 of Algorithms **??** and **??**) requires $O(m^2)$ space to store the list of points sorted by distance from any points of S. To avoid this space complexity we can use KS to retrieve the points contained in circle of the corresponding radius (d_{i,a_i^1} for Algorithms **??** and d_{i,o_i^1} for Algorithm **??**).

We can distinguished three different approaches:

- BL (Base Line): the approach proposed in [**?**] corresponding to Algorithms **??** and **??** that iterates over all points, updates the priority queues as well as the closest and second closest facility of each point when needed.
- LIBL (Less Incremental Base Line): this approach is a simple modification of the BL that does not use geometry. It simply ignores the priority queues in Algorithm **??** as closest and second-closest can be updated in constant time

when opening a facility and it is based on Algorithm **??** for closing but do not use the kd-tree KW to update the closest and second-closest. It simply iterates over W to update the two closest facilities.
- GEO (Geometric): the new approach proposed based on Algorithm **??** and **??**.

We summarize the complexity of the three approaches in Table **??** where m is the number of nodes, p the number of facilities, k the number of nodes which have p_j as a closest or second closest and $k^{'}$ is an upper bound on k useful to express the complexity (it is the number of points contained in the enclosing rectangle of the extended Voronoi cell) as we still have $k^{'} \leq m$; ignoring the linear update of Δ^- and Δ^+ which is identical and optimal in the three cases.

Table 1. Summary of time complexities of the different schemes

operation	BL	LIBL	GEO
open a facility p_j	$O(mlog(p))$	$O(m)$	$O(plog(p) + \sqrt{m} + k^{'})$
close a facility p_j	$O(mlog(p))$	$O(kp)$	$expected: O(klog(p))$, worst-case: $O(kp)$

5 Conclusion

We have presented new ideas to achieve efficient and incremental evaluation of the neighborhood of local search algorithms for facility location problems. These ideas apply when the cost for allocating a client to a facility is proportional to the Euclidean distance. We showed how to use computational geometry to efficiently maintain the closest or second closest client to a facility. We also showed how this can be integrated within existing state of the art local search techniques for this class of problems. Any neighborhood involving the maintenance of closest and second closest could benefit from these ideas and the techniques presented make sense for a constraint-based local search framework where this type of incrementality is needed for spatial location problems.

This work remains preliminary and its experimental evaluation still needs to be completed. Many improvements are also possible as computational geometry is a very rich and active domain. For example, the use of range trees [**?**] instead of kd-trees would lead to a $O(log^2(m) + k)$ complexity (instead of $O(\sqrt{(}m) + k)$ for kd-tree) for a small increase of the space complexity to $O(mlog(m))$ (instead of $O(m)$ for the kd-tree). Besides the experimental evaluation, we will explore the generalization of this work to other metric distances or general distance matrices.

Acknowledgement. This work is supported by Science Foundation Ireland Grant No. 10/CE/I1853.