



Transportation Science

Publication details, including instructions for authors and subscription information:
<http://pubsonline.informs.org>

A New Algorithm for the Single Source Weber Problem with Limited Distances

Giovanni Righini

To cite this article:

Giovanni Righini (2021) A New Algorithm for the Single Source Weber Problem with Limited Distances. Transportation Science

Published online in Articles in Advance 27 Aug 2021

. <https://doi.org/10.1287/trsc.2021.1083>

Full terms and conditions of use: <https://pubsonline.informs.org/Publications/Librarians-Portal/PubsOnLine-Terms-and-Conditions>

This article may be used only for the purposes of research, teaching, and/or private study. Commercial use or systematic downloading (by robots or other automatic processes) is prohibited without explicit Publisher approval, unless otherwise noted. For more information, contact permissions@informs.org.

The Publisher does not warrant or guarantee the article's accuracy, completeness, merchantability, fitness for a particular purpose, or non-infringement. Descriptions of, or references to, products or publications, or inclusion of an advertisement in this article, neither constitutes nor implies a guarantee, endorsement, or support of claims made of that product, publication, or service.

Copyright © 2021, INFORMS

Please scroll down for article—it is on subsequent pages



With 12,500 members from nearly 90 countries, INFORMS is the largest international association of operations research (O.R.) and analytics professionals and students. INFORMS provides unique networking and learning opportunities for individual professionals, and organizations of all types and sizes, to better understand and use O.R. and analytics tools and methods to transform strategic visions and achieve better outcomes.

For more information on INFORMS, its publications, membership, or meetings visit <http://www.informs.org>

A New Algorithm for the Single Source Weber Problem with Limited Distances

Giovanni Righini^a

^aDepartment of Computer Science, University of Milan, Milan 20122, Italy

Contact: giovanni.righini@unimi.it,  <https://orcid.org/0000-0001-9830-7454> (GR)

Received: April 16, 2020

Revised: January 26, 2021; April 10, 2021

Accepted: June 3, 2021

Published Online in *Articles in Advance*:

August 27, 2021

<https://doi.org/10.1287/trsc.2021.1083>

Copyright: © 2021 INFORMS

Abstract. The single source Weber problem with limited distances (SSWPLD) is a continuous optimization problem in location theory. The SSWPLD algorithms proposed so far are based on the enumeration of all regions of \mathbb{R}^2 defined by a given set of n intersecting circumferences. Early algorithms require $O(n^3)$ time for the enumeration, but they were recently shown to be incorrect in case of degenerate intersections, that is, when three or more circumferences pass through the same intersection point. This problem was fixed by a modified enumeration algorithm with complexity $O(n^4)$, based on the construction of neighborhoods of degenerate intersection points. In this paper, it is shown that the complexity for correctly dealing with degenerate intersections can be reduced to $O(n^2 \log n)$ so that existing enumeration algorithms can be fixed without increasing their $O(n^3)$ time complexity, which is due to some preliminary computations unrelated to intersection degeneracy. Furthermore, a new algorithm for enumerating all regions to solve the SSWPLD is described: its worst-case time complexity is $O(n^2 \log n)$. The new algorithm also guarantees that the regions are enumerated only once.

Supplemental Material: The online appendices are available at <https://doi.org/10.1287/trsc.2021.1083>.

Keywords: Weber problem • depth-first-search

1. The Problem

The single source Weber problem with limited distances (SSWPLD), also known as the facility location problem with limited distances, is a continuous optimization problem in location theory. A set $\mathcal{N} = \{1, 2, \dots, n\}$ of circumferences in \mathbb{R}^2 is given. For each circumference $i \in \mathcal{N}$, a center in position O_i , a radius $r_i \geq 0$, and a weight $w_i \geq 0$ are given. The problem consists of optimally locating a point X in \mathbb{R}^2 minimizing the weighted sum of cost terms depending on its distances from the centers of the circumferences. The cost term for each circumference $i \in \mathcal{N}$ is the minimum of the distance between X and O_i and the radius r_i . The objective function is as follows:

$$z = \min_{X \in \mathbb{R}^2} \left\{ \sum_{i \in \mathcal{N}} w_i \min\{d(O_i, X), r_i\} \right\},$$

where $d()$ indicates the Euclidean distance in \mathbb{R}^2 .

Drezner, Mehrez, and Wesolowsky (1991) proposes an algorithm to solve the SSWPLD as an unrestricted single-source location problem for each region of the partition of \mathbb{R}^2 induced by the circumferences. A region is defined by the subset of circumferences including it. Hence, the objective can be restated as follows:

$$z = \min_{Q \subseteq \mathcal{N}, X \in \mathbb{R}^2} \left\{ \sum_{i \in Q} w_i d(O_i, X) + \sum_{i \notin Q} w_i r_i : d(O_i, X) \leq r_i \ \forall i \in Q \right\}.$$

The constraint $d(O_i, X) \leq r_i \ \forall i \in Q$ can be dropped because any solution $(Q, X) : \exists i \in Q, d(O_i, X) > r_i$ is dominated by another solution (Q', X) with $Q' = Q \setminus \{i\}$.

Indicating with R the set of regions of \mathbb{R}^2 induced by the circumferences, the SSWPLD can be reformulated as

$$z = \min_{Q \in R, X \in \mathbb{R}^2} \left\{ \sum_{i \in Q} w_i d(O_i, X) + \sum_{i \notin Q} w_i r_i \right\}.$$

If an algorithm is available to compute the optimal location $X^*(Q)$ for each region $Q \in R$ with the corresponding optimal value $z^*(Q)$, then the problem is

$$z = \min_{Q \in R} \left\{ z^*(Q) + \sum_{i \notin Q} w_i r_i \right\},$$

and it can be solved by enumerating the regions in R , as suggested by Drezner, Mehrez, and Wesolowsky (1991).

The single-source optimal location problem, or one-median problem, can be solved by the classical infinite algorithm proposed by Weiszfeld (1937) or one of its variations (e.g., Ostresh 1978).

In this paper, the focus is on the complexity of the region enumeration algorithm, building upon the papers by Drezner, Mehrez, and Wesolowsky (1991) and Venkateshan (2020).

The algorithm proposed by Drezner, Mehrez, and Wesolowsky (1991) relies upon a theorem stating that n circumferences in \mathbb{R}^2 induce up to $2n(n-1)$ distinct regions. Therefore, the single-source optimal location algorithm must be executed a quadratic number of times to find the optimum of the SSWPLD. The enumeration algorithm of Drezner, Mehrez, and Wesolowsky (1991) is based on the observation that each intersection point between two circumferences is adjacent to four regions. For each intersection point P between two distinct circumferences $i \in \mathcal{N}$ and $j \in \mathcal{N}$, the set S_P of circumferences different from i and j that cover P is computed in $O(n)$. Then, a set R_P of four regions is generated: $R_P = \{S_P, S_P \cup \{i\}, S_P \cup \{j\}, S_P \cup \{i, j\}\}$. This procedure, repeated for all intersection points, that is, $O(n^2)$ times, generates the whole set of regions $R = \bigcup_P R_P$ in $O(n^3)$ time.

Unfortunately, this algorithm does not work correctly with “pathological” instances. One possible reason is the presence of circumferences entirely included in one another or disjoint from all the others. Aloise, Hansen, and Liberti (2012) show how to correct the algorithm in order to cope with instances with this structure. The complexity of their algorithm is $O(n^3)$ like that of Drezner, Mehrez, and Wesolowsky (1991).

More recently, Venkateshan (2020) pointed out the need for a further correction that is needed because of instances in which more than two circumferences pass through the same intersection point. In the Venkateshan (2020) algorithm, given an intersection point P between circumferences, a subset S_P is defined as the subset of circumferences *strictly* covering P , and a subset T_P is defined as the subset of circumferences passing through P . A “small enough” neighborhood is constructed around P , and the intersections of the circumferences in T_P with the frontier of the neighborhood are computed. Then, following the frontier of the neighborhood, one can correctly enumerate the set of all relevant subsets of T_P , that correspond to the regions with a vertex in P . The construction and analysis of the neighborhood requires $O(n^4)$, suggesting that the need to take into account the possible occurrence of degenerate intersections increases the complexity of the region enumeration problem.

In this paper, in Section 2, it is shown that this is not the case, because the same result obtained by Venkateshan’s (2020) method can be achieved with better computational complexity without actually constructing the neighborhoods but just distinguishing the two sides of the circumferences in T_P , that is, the interior and the exterior, and sorting the directions of their

tangent lines accordingly. In this way, the enumeration of all relevant subsets takes $O(n^2 \log n)$. However, after this improvement, the bottleneck of the overall enumeration algorithm is still the computation of all subsets S_P , that requires $O(n^3)$ in all algorithms devised so far.

In Section 3, a new region enumeration algorithm is illustrated: it does not require computing the subsets S_P , and it allows enumerating all regions in $O(n^2 \log n)$.

It must be remarked that the true bottleneck in the solution of the SSWPLD is the need of running the single-source optimal location algorithm for as many times as the number of regions that are enumerated. A remarkable feature of the new algorithm is that it guarantees to enumerate all regions only once.

2. An Improvement to Existing Algorithms

Preprocessing: A generic SSWPLD instance can be preprocessed for at least two purposes: (i) merging pairs of circumferences $i \in \mathcal{N}$ and $j \in \mathcal{N}$ with $O_i = O_j$ and $r_i = r_j$ in a unique circumference with the same center, the same radius, and weight $w_i + w_j$; (ii) eliminating circumferences with radius $r = 0$ or weight $w = 0$ because they have no effect on the value of any solution. Such preprocessing takes $O(n \log n)$ and it is not a computational complexity bottleneck.

In the remainder, the term “multiple intersection point” (m.i.p.) is used to indicate a point in \mathbb{R}^2 at which two or more circumferences intersect.

Given an m.i.p. P and the corresponding subset T_P of circumferences that intersect in P , the enumeration method proposed in Venkateshan (2020) is based on the construction of a small enough circular neighborhood of P such that there is no intersection other than P between the circumferences of T_P within the neighborhood.

A neighborhood with this property certainly exists because there are no two circumferences in T_P with the same center and the same radius owing to preprocessing.

Given an m.i.p. P and a small enough neighborhood of P with the preceding property, let us indicate its radius with ρ_P and its frontier with F_P . For the definition of small enough neighborhood and because ρ_P is guaranteed to be strictly positive, the following observation holds.

Observation 1. Given an m.i.p. P and a small enough neighborhood of P with frontier F_P , the intersection points of the circumferences in T_P with F_P are all distinct.

Assume scanning F_P according to an arbitrary orientation (e.g., counterclockwise) starting from an

arbitrary direction (e.g., the positive x semiaxis). Then, there exists a unique cyclic order in which the intersection points with the circumferences in T_P are encountered along F_P . By cyclic order, we mean a sequence in which the successor of the last element is the first one and the predecessor of the first element is the last one. Two cyclic orders are defined to be equal when they contain the same elements and each element has the same predecessor and successor in both.

Let us indicate by \bar{e}_i and \bar{l}_i the directions from P to the intersection points between F_P and each circumference $i \in T_P$ as shown in Figure 1. Assuming scanning F_P counterclockwise, the intersection point corresponding to \bar{e}_i is encountered when “entering” circumference $i \in \mathcal{N}$, and the intersection point corresponding to \bar{l}_i is encountered when “leaving” it.

Obviously, the cyclic order of the intersection points is equal to the cyclic order of the corresponding directions \bar{e} and \bar{l} . We indicate such a cyclic order by \bar{L}_P . Note that \bar{L}_P does not depend on ρ_P although the position of the intersection points on F_P does because, by definition, any small enough neighborhood does not contain intersections between the circumferences in T_P apart from P .

The cyclic order of the intersection points along F_P is the piece of information needed to correctly enumerate the regions around P as shown by Venkateshan (2020). Here, we observe that the same cyclic order can be computed in a slightly different way just by sorting the directions of the tangent lines in P .

Let us call g_i the direction from P to the center O_i of each circumference $i \in T_P$. We can easily obtain the directions of the lines tangential to circumference i in P corresponding to entering (e_i) and leaving (l_i) the

circumference when F is scanned counterclockwise: $e_i = g_i - \frac{\pi}{2}$ and $l_i = g_i + \frac{\pi}{2}$, where all angles are computed modulo 2π .

Because F_P is continuous and the circumferences are continuous, when ρ_P tends to zero the intersection points on F_P tend to P , and then \bar{e}_i tends to e_i and \bar{l}_i tends to l_i for each $i \in T_P$. Therefore, there exists a cyclic order L_P of the directions e and l that coincides with \bar{L}_P , that is, these two properties hold: (i) L_P can be obtained from \bar{L}_P by replacing \bar{e}_i with e_i and \bar{l}_i with l_i for each $i \in T_P$; (ii) L_P is one of the possible cyclic orders in which directions e and l can be sorted counterclockwise.

Ties do not exist in the cyclic order of directions \bar{e} and \bar{l} by Observation 1, but they can occur in cyclic orders of directions e and l because distinct circumferences in T_P can have coincident tangent lines. This can occur only when $g_i = g_j \pm \pi$ or when $g_i = g_j$. When ties occur, the unique cyclic order of the tangent lines that corresponds to \bar{L}_P must be determined. This is obtained by two simple tie-break criteria.

Tie-break criterion 1: For any $i \neq j \in T_P$ such that $e_i = l_j$ and $e_j = l_i$, l_j precedes e_i and l_i precedes e_j .

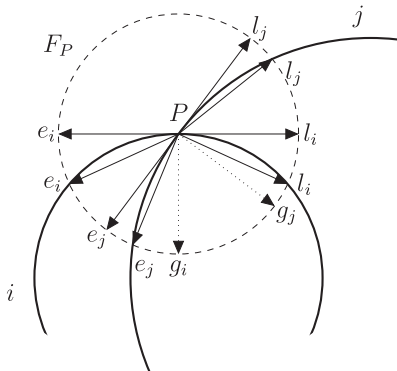
Tie-break criterion 2: For any $i \neq j \in T_P$ such that $l_i = l_j$ and $e_i = e_j$ with $r_i > r_j$, l_j precedes l_i and e_i precedes e_j .

Both criteria rely upon basic properties of tangent circumferences, illustrated in Figure 2. Tie-break criterion 1 solves ties occurring when $g_i = g_j \pm \pi$, and it is illustrated in Figure 2 on the left; tie-break criterion 2 solves ties occurring when $g_i = g_j$, and it is illustrated in Figure 2 on the right. The two criteria allow sorting the directions e and l in a uniquely defined cyclic order L_P equal to the unique cyclic order \bar{L}_P of the directions \bar{e} and \bar{l} . In turn, this allows computing L_P without computing \bar{L}_P and obtaining from L_P the same piece of information that can be obtained from \bar{L}_P .

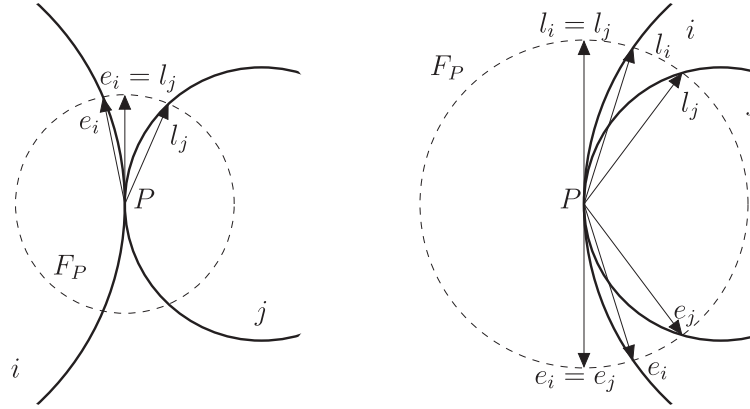
Venkateshan (2020) describes an algorithm to obtain the set of distinct regions around an m.i.p. P once the cyclic order \bar{L}_P of the intersection points on its neighborhood frontier has been obtained. Here, an alternative algorithm is described to achieve the same result from L_P . This enumeration algorithm is outlined in Algorithm 1. The procedure **Scan** is called for each m.i.p. P ; its input is a sequence V_P , defined hereafter, and the set S_P of circumferences strictly covering P as defined in Venkateshan (2020). Its effect is to call the single-source optimal location algorithm **Evaluate** for each region around P . For this purpose, the cyclic order L_P of the directions from P is represented as a sequence V_P of $2c$ elements with $c = |T_P|$. The sequence is obtained starting from an arbitrary element of L_P and replacing e_i with $+i$ and l_i with $-i$ for each $i \in T_P$.

In **Scan**, the sequence V_P is scanned twice. At any point during the execution, a current subset Q of circumferences in T_P is kept. Q is initialized at the empty set (line 2). When V_P is scanned for the first time (lines

Figure 1. An Intersection Point P Between Two Circumferences; Its Neighborhood; the Intersections Along Its Frontier F_P ; and the Directions \bar{e} , \bar{l} , e , and l



Note. In this example, $\bar{L}_P = \{\bar{l}_j, \bar{e}_i, \bar{e}_j, \bar{l}_i\}$.

Figure 2. Tie-Break Criteria

Notes. Tie-break criterion 1 (left): When leaving a circumference j and entering a circumference i with $g_i = g_j \pm \pi$, direction \bar{l}_j is encountered before \bar{e}_i . Tie-break criterion 2 (right): When leaving circumferences i and j with $g_i = g_j$ and $r_i > r_j$, direction \bar{l}_j is encountered before \bar{l}_i ; when entering them, \bar{e}_i is encountered before \bar{e}_j .

3–8), for each element, a circumference index in $\{1, \dots, n\}$ is inserted in Q or deleted from Q : when the entering point of circumference i is encountered (line 4), then i is inserted in Q (line 5); when the leaving point of circumference i is encountered, if $i \in Q$ (line 7), then it is deleted (line 8). It is trivial to prove that, at the end of the first scan, Q contains all indices $i \in T_P$ for which $-i$ precedes $+i$ in V_P and no index $i \in T_P$ for which $+i$ precedes $-i$ in V_P . Therefore, Q correctly represents the subset of circumferences in T_P covering the points of F_P between the last element of V_P and the first one. This provides a correct initialization for the second scan. During the second scan (lines 9–14), the same insertion/deletion rule is applied with the guarantee that $i \in Q$ whenever $-i$ is encountered. In this way all regions around P are correctly identified, and a single-source optimal location problem is solved for each of them.

Algorithm 1 (The Enumeration Algorithm to Be Executed for Each m.i.p. P)

```

1: procedure Scan( $V_P, S_P$ )
2:    $Q \leftarrow \emptyset$ 
3:   for  $t = 1, \dots, |V_P|$  do
4:     if  $V_P[t] > 0$  then
5:        $Q \leftarrow Q \cup \{V_P[t]\}$ 
6:     else
7:       if  $(-V_P[t] \in Q)$  then
8:          $Q \leftarrow Q \setminus \{-V_P[t]\}$ 
9:   for  $t = 1, \dots, |V_P|$  do
10:    if  $V_P[t] > 0$  then
11:       $Q \leftarrow Q \cup \{V_P[t]\}$ 
12:    else
13:       $Q \leftarrow Q \setminus \{-V_P[t]\}$ 
14:    Evaluate( $Q \cup S_P$ )

```

Three examples are provided in Online Appendix A to show how the algorithm works in full detail.

2.1. Computational Complexity

To establish the asymptotic worst-case time complexity of the region enumeration algorithm, it is necessary to distinguish three main steps. In step 1, one must compute the intersection points for all pairs of distinct circumferences, and one must detect when some of them coincide; the output is a list of m.i.p. In step 2, for each m.i.p. P , one must compute the set S_P of circumferences that strictly cover P . In step 3, all regions with a vertex in P are enumerated for each m.i.p. P , and a single-source optimal location algorithm is run for each detected region. Hereafter, the worst-case time complexity of each of these three steps is analyzed.

2.1.1. Step 1. The set of intersection points between circumferences can be computed in $O(n^2)$. To detect coincident intersections, intersection points can be sorted so that coincident intersection points turn out to be consecutive in the ordering. For instance, one can sort the intersection points in lexicographical order according to the x value, using the y value as a secondary criterion in case of identical x values. The intersection points are $O(n^2)$, and ordering a list of $O(n^2)$ elements requires $O(n^2 \log n)$ time. After that, all subsets T_P for each m.i.p. P can be identified in $O(n^2)$ by scanning the ordered list of $O(n^2)$ elements and iteratively merging consecutive elements of the list when their positions coincide. Each merge operation takes $O(1)$ because it requires checking whether two circumferences already belong to T_P and inserting them if they are not already in T_P ; insertion takes constant time if subsets are represented by their binary characteristic vectors. Hence, the asymptotic worst-case time complexity of step 1 is $O(n^2 \log n)$.

2.1.2. Step 2. For each m.i.p. P , listing the subset S_P of circumferences strictly covering it requires $O(n)$; therefore, step 2 has asymptotic worst-case time complexity $O(n^3)$. This is indeed the complexity of the region enumeration algorithms proposed by Drezner, Mehrez, and Wesolowsky (1991) and Aloise, Hansen, and Liberti (2012).

2.1.3. Step 3. This is the step on which we focus our study because it is the bottleneck step in Venkateshan's (2020) algorithm. We show that its complexity can be reduced from $O(n^4)$ to $O(n^2 \log n)$.

Consider an m.i.p. P and the corresponding subset T_P of c circumferences intersecting in P . Computing all directions g_i from P to $O_i \forall i \in T_P$ takes $O(c)$. Computing all directions e_i and l_i takes $O(c)$. Sorting the sequence L_P with $2c$ angle values takes $O(c \log c)$. Scanning L_P to enumerate all regions around P with Algorithm 1 takes $O(c)$ because insertion and deletion operations on lines 5, 8, 11, and 13 of Scan can be implemented as $O(1)$ operations on a binary array (whose initialization takes $O(c)$), and the number of iterations of the loops in Scan is bounded by $2c$.

Procedure Scan must be repeated for all m.i.p. The number of m.i.p. grows as $O(n^2)$. Therefore, the asymptotic worst-case complexity of step 3, based on sorted tangent lines, is not worse than $O(n^3 \log n)$, which is already an improvement with respect to the $O(n^4)$ complexity of the algorithm based on the explicit construction of the neighborhood.

However, it is also possible to further refine the complexity analysis of step 3 to prove a better bound. In an m.i.p. P in which $c > 2$ circumferences intersect, a number of intersections coincide. This number is the triangular number $\sum_{h=1}^{c-1} h = c(c-1)/2$. Therefore, degeneration actually *decreases* the computational complexity of the region enumeration problem because a quadratic number of intersection points is treated in a single point at the expense of a less-than-quadratic overhead. To express this formally, we need to establish the following theorem.

Theorem 1. Consider the multigraph $\mathcal{M} = (\mathcal{V}, \mathcal{E})$ defined by n intersecting circumferences, in which \mathcal{V} is the set of m.i.p. and \mathcal{E} is the set of circumference arcs between them. Then, $|\mathcal{E}|$ grows as $O(n^2)$.

Two proofs are given.

Proof 1. Consider the multigraph $\tilde{\mathcal{M}} = (\tilde{\mathcal{V}}, \tilde{\mathcal{E}})$ obtained by a small perturbation of the circumferences at the m.i.p. at which more than two circumferences intersect so that no degenerate intersections occur in $\tilde{\mathcal{M}}$. Then, all vertices in $\tilde{\mathcal{M}}$ have degree four. The number of pairs of distinct circumferences is $n(n-1)/2$, and for each pair, at most two intersection points exist. Hence, the number of vertices in $\tilde{\mathcal{M}}$ is not larger than $n(n-1)$. Because in $\tilde{\mathcal{M}}$ all vertices have degree four,

the total degree in $\tilde{\mathcal{M}}$ is bounded by $4n(n-1)$. Because each edge has two endpoints, then $|\tilde{\mathcal{E}}| \leq 2n(n-1)$. By construction, all edges of \mathcal{M} have a counterpart in $\tilde{\mathcal{M}}$ although the converse does not hold: hence, $|\mathcal{E}| \leq |\tilde{\mathcal{E}}|$. Therefore, the number of edges in \mathcal{M} is also bounded above by $2n(n-1)$.

Proof 2. For any given planar multigraph $\mathcal{M} = (\mathcal{V}, \mathcal{E})$ inducing a set of regions \mathcal{R} in \mathbb{R}^2 , Euler formula holds: $|\mathcal{E}| + 2 = |\mathcal{V}| + |\mathcal{R}|$. By the Drezner, Mehrez, and Wesolowsky (1991) theorem, $|\mathcal{R}|$ is $O(n^2)$. Because $|\mathcal{V}|$ is also $O(n^2)$, then $|\mathcal{E}|$ is $O(n^2)$.

Corollary 1. The total degree of the vertices in \mathcal{V} grows as $O(n^2)$.

This immediately follows from Theorem 1 because the total degree is twice the number of edges.

The asymptotic worst-case time complexity of step 3 is given by $O(\sum_{k=1}^K c_k \log c_k)$, where K indicates the number of m.i.p. and c_k the number of circumferences intersecting in each m.i.p. $k = 1, \dots, K$. Because $c_k \leq n \forall k = 1, \dots, K$, and hence, $\log c_k \leq \log n \forall k = 1, \dots, K$, a valid worst-case bound is $O(\log n \sum_{k=1}^K c_k)$. The sum $\sum_{k=1}^K c_k$ is half the total degree of the vertices of the multigraph \mathcal{M} defined earlier. For Corollary 1, such a total degree grows as $O(n^2)$. Therefore, an asymptotic worst-case bound for step 3 is $O(n^2 \log n)$.

The main conclusion of this complexity analysis is that degenerate intersections in the SSWPLD can be dealt with without worsening the $O(n^3)$ worst-case time complexity of the enumeration algorithms proposed so far that did not take degeneracy into account. The computational complexity bottleneck in the enumeration is not a result of degenerate intersections (affecting steps 1 and 3), but rather of the need of checking whether each given circumference covers each m.i.p. in step 2. All algorithms proposed so far require $O(n^3)$ time complexity for this crucial step. The next section describes a new enumeration algorithm that does not require this step and has $O(n^2 \log n)$ complexity.

3. A New Algorithm

A set of intersecting circumferences induces one or more planar multigraphs in \mathbb{R}^2 . Their *vertices* are m.i.p., that is, subsets of intersection points between pairs of circumferences. When two or more intersection points coincide, they belong to the same vertex. We call *edges* the circumference arcs between adjacent vertices. We further remark that the multigraphs induced by the circumferences are planar by definition, that is, there is no other intersection between edges apart from vertices.

The new algorithm runs in four steps. In step 1, all intersection points are enumerated, and they are

sorted to find coincident intersections; they are the vertices of a set of planar multigraphs. In step 2, the vertices occurring along each circumference are sorted according to a given orientation, and this allows identifying all edges of the multigraphs and computing the star of each vertex. In step 3, the circumference arcs incident to each vertex are sorted so that the star of each vertex can be scanned according to a given orientation. In step 4, each planar multigraph is visited with a depth-first-search algorithm, and all regions are enumerated.

3.1. Step 1: Enumeration of Vertices

First of all, in order to compute the planar multigraphs mentioned, it is necessary to find their vertices, that is, all subsets of coincident intersection points. For the sake of clarity, the description of this step is broken into three substeps.

3.1.1. Step 1.1: Enumeration of Intersection Points.

The first substep of the algorithm is the enumeration of all intersection points between pairs of distinct circumferences, and it is described in Intersections in Algorithm 2.

Algorithm 2 (The Algorithm that Enumerates All Intersection Points and All Enclosing Circumferences)

```

1: procedure Intersections IN:  $O, r$ . OUT:  $\Omega, f, \Lambda$ 
2:   for  $i = 1, \dots, n$  do
3:      $f(i) \leftarrow \text{false}$ 
4:      $\Omega(i) \leftarrow \emptyset$ 
5:    $\Lambda \leftarrow \emptyset$ 
6:   for  $i = 1, \dots, n - 1$  do
7:     for  $j = i + 1, \dots, n$  do
8:       if  $(d(O_i, O_j) < |r_i - r_j|)$  then
9:         if  $r_i > r_j$  then
10:           $\Omega(j) \leftarrow \Omega(j) \cup \{i\}$ 
11:        else
12:           $\Omega(i) \leftarrow \Omega(i) \cup \{j\}$ 
13:        else
14:          if  $(d(O_i, O_j) \leq r_i + r_j)$  then
15:             $[P(i, j), P(j, i)] \leftarrow \text{Intersect}(i, j)$ 
16:             $f(i) \leftarrow \text{true}$ 
17:             $f(j) \leftarrow \text{true}$ 
18:             $t\Lambda \leftarrow \Lambda \cup \{[i, j, x(P(i, j)), y(P(i, j))], [j, i, x(P(j, i)), y(P(j, i))]\}$ 

```

Algorithm Intersections has three main effects: first, a subset $\Omega(i)$ of enclosing circumferences is computed for each circumference $i \in \mathcal{N}$; second, a flag $f(i)$ is set for each circumference $i \in \mathcal{N}$, stating whether the circumference intersects at least another one or it is isolated; third, a list Λ of all intersection points is produced by considering all pairs of distinct circumferences.

All sets Ω are initially empty. When the test on line 8 succeeds, then one of the two circumferences i and j is strictly enclosed in the other; then, the subset Ω of the smallest circumference is updated to include the largest circumference, and no intersection point is computed.

All flags f are initially set to false. If the test on line 8 fails and the test on line 14 succeeds, then circumferences i and j have two (possibly coincident) intersection points; therefore, their flags $f(i)$ and $f(j)$ are set to true. The two intersection points are identified as $P(i, j)$ and $P(j, i)$ for each pair of circumferences i and j with $i < j$. Assume all circumferences are followed counterclockwise. Then, as shown in Figure 3, $P(i, j)$ is where circumference i enters circumference j and circumference j leaves circumference i , and the converse occurs in $P(j, i)$. The coordinates of the two intersection points are computed in constant time by a suitable function `Intersect()` (line 15). Then, they are added to the list Λ of all intersection points (line 18). Each element of Λ is a record with four fields $[i, j, x, y]$, representing the entering circumference, the leaving circumference, and the coordinates of the intersection point.

If both tests fail, then circumferences i and j are disjoint, and no update occurs to Ω, f , and Λ .

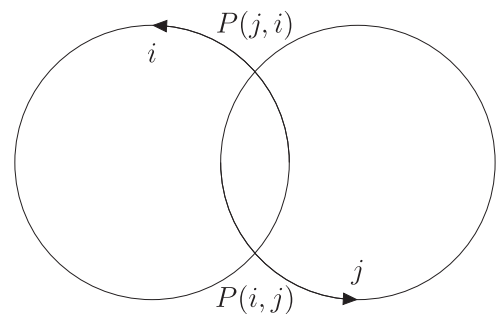
The complexity of Intersections is $O(n^2)$ because of the two nested loops (lines 6 and 7) that contain $O(1)$ operations.

3.1.2. Step 1.2: Ordering the Intersection Points.

Step 1.2 is quite simple to describe, but it turns out to be a bottleneck of the whole region enumeration algorithm. It consists of ordering the list Λ of the intersection points according to any arbitrary criterion so that coincident points occur in consecutive positions in the ordered list. In this way, it is possible to enumerate the vertices of the multigraphs induced by the intersecting circumferences.

For instance, the points in Λ can be sorted by non-decreasing values of x , and ties can be broken by sorting them by nondecreasing values of y .

Figure 3. Intersection Points Between Two Circumferences



Because $|\Lambda|$ is $O(n^2)$, the complexity of sorting the intersection points is $O(n^2 \log n)$.

3.1.3. Step 1.3: Enumeration of Vertices. For notational convenience, we assume here that the sorted list Λ is transformed into an array. The sorted array Λ is scanned to find the vertices; this is done by FindVertices, illustrated in Algorithm 3. Indices t' and t'' are used to find the first and last positions of the elements in each subset of coincident points. The integer v indicates the number of vertices found.

Algorithm 3 (The Algorithm that Enumerates the Vertices of the Multigraphs)

```

1: procedure FindVertices. IN:  $\Lambda$ . OUT:  $x, y, T, v$ 
2:    $v \leftarrow 0$ 
3:    $t' \leftarrow 1$ 
4:   while  $t' \leq |\Lambda|$  do
5:      $t'' \leftarrow t' + 1$ 
6:     while  $(t'' \leq |\Lambda|) \wedge (\Lambda[t''].x = \Lambda[t'].x) \wedge (\Lambda[t''].y = \Lambda[t'].y)$  do
7:        $t'' \leftarrow t'' + 1$ 
8:      $v \leftarrow v + 1$ 
9:      $x(v) \leftarrow \Lambda[t'].x$ 
10:     $y(v) \leftarrow \Lambda[t'].y$ 
11:     $T(v) \leftarrow \emptyset$ 
12:    for  $h = t', \dots, t'' - 1$  do
13:       $T(v) \leftarrow T(v) \cup \{\Lambda[h].i, \Lambda[h].j\}$ 
14:     $t' \leftarrow t''$ 

```

For each vertex $k = 1, \dots, v$, $x(k)$ and $y(k)$ are its coordinates, and $T(k)$ is the set of all circumferences passing through it. Each set $T(k)$ can be implemented as a balanced tree: in this way, duplicates can be detected so that each circumference appears only once in it. This implies that inserting an element in $T(k)$ (line 13 of FindVertices) has $O(\log n)$ complexity.

Because $|\Lambda|$ is $O(n^2)$, the complexity of FindVertices is $O(n^2 \log n)$.

Therefore, the overall worst-case time complexity of step 1 is $O(n^2 \log n)$.

3.2. Step 2: Enumeration of Edges

In step 2, vertices are sorted according to the order in which they are encountered when moving along each circumference counterclockwise. For the sake of clarity, the description of step 2 is broken into three substeps.

3.2.1. Step 2.1: Enumeration of the Vertices Along Each Circumference.

For each circumference $i \in \mathcal{N}$, a set $W(i)$ of vertices is computed. This is done by EnumerateVertices, illustrated in Algorithm 4. The list of all vertices $k = 1, \dots, v$ is scanned: for each circumference $i \in \mathcal{N}$ that occurs in $T(k)$, an element k is inserted in the subset $W(i)$.

Algorithm 4 (The Algorithm that Enumerates All Vertices Along Each Circumference)

```

1: procedure EnumerateVertices. IN:  $T, v$ . OUT:  $W$ 
2:   for  $i \in \mathcal{N}$  do
3:      $W(i) \leftarrow \emptyset$ 
4:     for  $k = 1, \dots, v$  do
5:       for  $i \in T(k)$  do
6:          $W(i) \leftarrow W(i) \cup \{k\}$ 

```

Every time a vertex is found to belong to a circumference, it contributes by an amount of two to the total degree of the multigraphs. Because the total degree of the multigraphs is $O(n^2)$, the insertion on line 6 is done $O(n^2)$ times, and therefore, the time complexity of EnumerateVertices is also $O(n^2)$.

3.2.2. Step 2.2: Sorting the Vertices Along Each Circumference.

Each circumference $i \in \mathcal{N}$ is examined separately. For each vertex k in $W(i)$, the direction from O_i to the point of coordinates $(x(k), y(k))$ is considered, and the corresponding angle $\alpha(i, k)$ is computed. Function $\arctan()$ is assumed to return a value in $[0, 2\pi)$ computed counterclockwise from the positive x semiaxis (line 4). Then, the subset $W(i)$ is sorted by increasing values of α . No tie can occur in the order because, by construction, all vertices are distinct, and distinct points along a circumference are guaranteed to produce distinct values of α . Step 2.2 is executed by SortVertices illustrated in Algorithm 5.

Algorithm 5 (The Algorithm That Sorts the Vertices Along Each Circumference)

```

1: procedure SortVertices. IN:  $W$ . OUT:  $W$ 
2:   for  $i = 1, \dots, n$  do
3:     for  $k \in W(i)$  do
4:        $\alpha(i, k) \leftarrow \arctan(O_i, (x(k), y(k)))$ 
5:      $W(i) \leftarrow \text{Sort}(W(i))$ 

```

As already shown in Section 3.2.1, the number of (i, k) pairs in the multigraphs is $O(n^2)$. Therefore, the number of calls to $\arctan()$ is $O(n^2)$. The time complexity bottleneck is given by the sorting operation: sorting the vertices takes $O(|W(i)| \log |W(i)|)$ for each circumference $i \in \mathcal{N}$. Because $|W(i)| \leq 2(n-1) \forall i \in \mathcal{N}$ and $\sum_{i=1}^n |W(i)| \leq 2n(n-1)$, the time complexity of SortVertices is $O(n^2 \log n)$. As for step 1.2, this is a computational complexity bottleneck of the new enumeration algorithm.

3.2.3. Step 2.3: Building Vertex Stars.

The structure of the multigraphs is finally produced by connecting the vertices with circumference arcs. Once the list of vertices along each circumference has been sorted in step 2.2, this information is used to build a suitable data-structure H for each vertex, representing the star of the vertex, that is, the ordered set of edges with an endpoint in that vertex. For a generic vertex k , each

element in its star $H(k)$ is a triplet (i, γ, h) , where i is the index of a circumference passing through the vertex, γ is a bit representing “counterclockwise” with one and “clockwise” with zero, and h is the index of the vertex that is reached from vertex k following circumference i in direction γ .

The pseudo-code of BuildStar is shown in Algorithm 6. The set $H(k)$ is initialized to the empty set for each vertex k (line 3). Then, each circumference is considered, and each pair of consecutive vertices k' and k'' is considered along it, scanning $W(i)$ as a circular list so that also the last element and the first one form a consecutive pair (line 6). Finally, the edge between k' and k'' is inserted in $H(k')$ as a counterclockwise edge entering k'' and in $H(k'')$ as a clockwise edge entering k' . As a special case, it is possible that $W(i)$ contains a single vertex k . In this case, two edges are inserted in $H(k)$ with opposite directions γ and with the second endpoint equal to k .

Algorithm 6 (The Algorithm that Builds the Star of Each Vertex)

```

1: procedure BuildStar. IN:  $W$ . OUT:  $H$ 
2:   for  $k = 1, \dots, v$  do
3:      $H(k) \leftarrow \emptyset$ 
4:     for  $i \in \mathcal{N}$  do
5:       for  $k' \in W(i)$  do
6:          $k'' \leftarrow \text{succ}(k')$ 
7:          $H(k') \leftarrow H(k') \cup \{(i, 1, k'')\}$ 
8:          $H(k'') \leftarrow H(k'') \cup \{(i, 0, k')\}$ 

```

The complexity for scanning all the W subsets is $O(n^2)$ as already observed. The total number of elements in subsets H is twice the total number of edges in the multigraphs because each edge is inserted in two stars. Hence, the time complexity of BuildStar is $O(n^2)$.

3.3. Step 3: Sorting Vertex Stars

The subsets H computed in step 2 indicate which edges of the multigraphs are incident to each vertex. The aim of step 3 is to sort the stars so that consecutive edges belong to the frontier of a same region, owing to the planarity of the multigraphs. This step is necessary to enumerate the regions while visiting the multigraphs.

For the sake of clarity, the description of step 3 is broken into two substeps.

3.3.1. Step 3.1: Computing Edge Directions. A direction $\beta(k, i, \gamma)$ is associated with each edge along a circumference $i \in \mathcal{N}$ and belonging to $H(k)$ for some vertex $k = 1, \dots, v$: it is the direction of the line tangent to the circumference i in vertex k , oriented from the vertex in direction γ . The tangent certainly exists because preprocessing guarantees that all circumferences have a strictly positive radius. As before, angles are computed counterclockwise starting from the

direction of the positive x semiaxis. The computation is done by ComputeDirections, shown in Algorithm 7. The effect of this procedure is to add a fourth field β to the three-field records (i, γ, h) in the subset $H(k) \forall k = 1, \dots, v$ (line 8).

Algorithm 7 (The Algorithm that Computes a Direction for Each Edge in Each Vertex Star)

```

1: procedure ComputeDirections. IN:  $H, O, (x, y)$ . OUT:  $\beta$ 
2:   for  $k = 1, \dots, v$  do
3:     for  $(i, \gamma, h) \in H(k)$  do
4:       if  $\gamma = 1$  then
5:          $\beta \leftarrow (\arctan(O_i, (x(k), y(k))) + \pi/2) \bmod 2\pi$ 
6:       else
7:          $\beta \leftarrow (\arctan(O_i, (x(k), y(k))) - \pi/2) \bmod 2\pi$ 
8:       Replace  $(i, \gamma, h)$  with  $(i, \gamma, h, \beta)$ 

```

Because the total number of elements in the subsets H is $O(n^2)$, the complexity of ComputeDirections is $O(n^2)$.

3.3.2. Step 3.2: Sorting the Edges. For each vertex $k = 1, \dots, v$, its star $H(k)$ is sorted counterclockwise, according to the values of the angle β of each incident edge. However, ties may occur because it may happen that two or more circumferences have the same tangent lines in their intersection points. In these cases, the following criteria are used to break ties.

Tie-break criterion 3: Given a tie between two edges $(i, 0, \beta)$ and $(j, 1, \beta)$, $(i, 0, \beta)$ must precede $(j, 1, \beta)$ in $H(k)$.

Tie-break criterion 4: (a) Given a tie between two edges $(i, 0, \beta)$ and $(j, 0, \beta)$ with $r_i < r_j$, $(i, 0, \beta)$ must precede $(j, 0, \beta)$ in $H(k)$. (b) Given a tie between two edges $(i, 1, \beta)$ and $(j, 1, \beta)$ with $r_i < r_j$, $(j, 1, \beta)$ must precede $(i, 1, \beta)$ in $H(k)$.

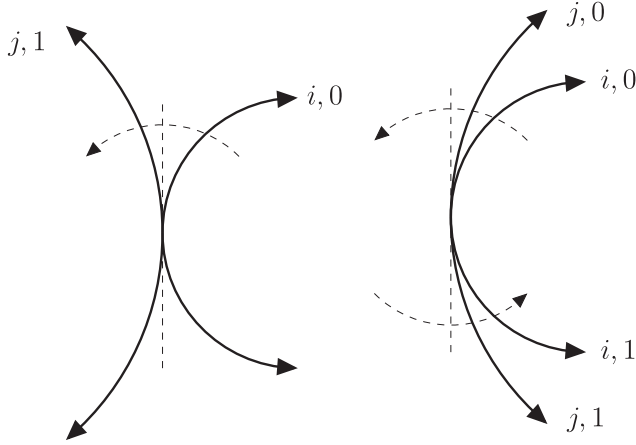
These criteria are quite similar to those illustrated in Section 2, and they have the same meaning. Their justification is trivial, and it is illustrated in Figure 4.

The resulting sorted list $H(k)$ for each $k = 1, \dots, v$ is managed as a circular array so that the successor of the last element is the first one.

The effect of step 3 is to sort the edges in $H(k)$ in the same order as they would be encountered moving counterclockwise along the frontier of a small enough neighborhood of vertex k . Consequently, consecutive edges in $H(k)$ belong to the contour of a same region.

There are $O(n^2)$ vertices and $|H(k)|$ edges in each vertex star. The complexity for sorting all vertex stars is $O(\sum_{k=1}^v |H(k)| \log |H(k)|)$. Because $|H(k)| \leq 2n \forall k = 1, \dots, v$ and $\sum_{k=1}^v |H(k)| \leq 4n(n-1)$, the time complexity of step 3.2 is $O(n^2 \log n)$. Together with the sorting steps 1.2 and 2.2, this is the third computational complexity bottleneck of the new algorithm.

Figure 4. Sorting Edges with the Same Tangent Line in a Vertex Star



Notes. Left: Tie-break criterion 3 is applied to break ties between $(i, 0, \beta)$ and $(j, 1, \beta)$. Right: Tie-break criterion 4 is applied to break ties between $(i, 0, \beta)$ and $(j, 0, \beta)$ and between $(i, 1, \beta)$ and $(j, 1, \beta)$.

3.4. Step 4: Region Enumeration

Besides multigraphs, the set of given circumferences of an SSWPLD instance may also contain isolated circumferences, that is, circumferences with no intersections with others. It is also possible that some multigraphs or isolated circumferences are completely enclosed in other multigraphs or isolated circumferences. In order to correctly deal with all these cases, we need some preliminary observations.

We define a *connected component* to be either a multigraph or an isolated circumference. Each circumference belongs to exactly one connected component. We indicate by $\phi(i) \subseteq \mathcal{N}$ the connected component of circumference i ; it exists, and it is unique for each $i \in \mathcal{N}$. With this definition, an isolated circumference is just a special case of a connected component made of a single circumference.

As shown in the introduction, in order to correctly solve the SSWPLD, it is necessary and sufficient to enumerate all regions of \mathbb{R}^2 induced by all connected components. The set of points not enclosed in any connected component is of no interest because it cannot contain the optimal solution. Actually, it is the set of the worst solutions of the SSWPLD, in which z attains its maximum value $\sum_{i \in \mathcal{N}} w_i r_i$.

Nonoverlapping connected components induce disjoint sets of regions that can be enumerated independently. Their union is the whole set of regions that must be enumerated.

Step 4 visits the whole set of input circumferences, one connected component at a time. If a connected component is a multigraph, then it is completely visited, and all its internal regions are enumerated. The pseudo-code of *Components* is shown in Algorithm 8.

Algorithm 8 (The Algorithm that Enumerates All Connected Components)

```

1. procedure Components
2.    $N \leftarrow \text{SortCircles}(\mathcal{N})$ 
3.    $\bar{\mu} \leftarrow 0$ 
4.   for  $k \in 1, \dots, v$  do
5.      $\mu(k) \leftarrow 0$ 
6.   while  $N \neq \emptyset$  do
7.      $i^* \leftarrow \text{Rightmost}(N)$ 
8.      $Q \leftarrow \Omega(i^*)$ 
9.     if  $f(i^*)$  then
10.      /* Multigraph */
11.       $k \leftarrow W(i^*)_1$ 
12.       $\bar{\mu} \leftarrow \bar{\mu} + 1$ 
13.       $\mu(k) \leftarrow \bar{\mu}$ 
14.       $\text{ScanStar}(k, i^*, 1)$ 
15.   else
16.     /* Isolated circumference */
17.      $\text{Flip}(i^*)$ 
18.      $\text{Evaluate}(Q)$ 
19.      $N \leftarrow N \setminus \{i^*\}$ 

```

A set N , implemented as a binary array, is initialized to the set of all given circumferences \mathcal{N} . Then, every time a connected component is examined, all its circumferences are deleted from N as soon as they are encountered during the visit (see line 19 of *Components* and line 6 of *ScanStar*). The algorithm terminates when N is empty (line 6 of *Components*). This guarantees that all connected components are examined once. Algorithm *SortCircles* (line 2 of *Components*) initially sorts the circumferences in N by nonincreasing value of the abscissa of their rightmost point. In case of ties, the circumferences are sorted by decreasing values of their radius. In the case of a further tie, the selection is done at random. At each iteration of the loop on lines 6–19 of *Components* a rightmost unvisited circumference $i \in N$ is selected by the function *Rightmost* (line 7 of *Components*), and its connected component is examined.

In this paragraph, we indicate by C_i the subset made of a single circumference $i \in \mathcal{N}$, that is, $C_i = \{i\}$. To indicate that a circumference encloses another one or a whole connected component, we use the symbol \sqsupset . By “enclosing,” we mean “strictly covering.” For a circumference C and a connected component ϕ , $C \sqsupset \phi$ if and only if all points of ϕ are within C and no point of C belongs to ϕ . Recall that $\Omega(i) \subset \mathcal{N}$, computed in step 1, is the subset of circumferences that strictly enclose each circumference $i \in \mathcal{N}$, that is, $\Omega(i) = \{j \in \mathcal{N} : C_j \sqsupset C_i\} \forall i \in \mathcal{N}$. With these definitions, the set of all circumferences strictly enclosing the connected component $\phi(i)$ is determined by the following property.

Theorem 2. *If circumference $i \in \mathcal{N}$ is the rightmost circumference in its connected component $\phi(i)$, then $\Omega(i)$ is the set of circumferences that strictly enclose $\phi(i)$:*

$$j \in \Omega(\text{Rightmost}(\phi)) \Leftrightarrow C_j \supset \phi.$$

Proof.

i. Assume $j \in \Omega(\text{Rightmost}(\phi))$. Then, $C_j \supset C_i \forall i \in \phi$. Let $i^* = \text{Rightmost}(\phi)$. So, in particular, $C_j \supset C_{i^*}$. By contradiction, assume that $j \in \phi$. Then, $C_j \supset C_{i^*}$ implies that j precedes i^* in N , that is, i^* cannot be $\text{Rightmost}(\phi)$. Hence, it is proven by contradiction that $(j \in \Omega(\text{Rightmost}(\phi))) \Rightarrow j \in \phi$. If $j \in \phi$ and $C_j \supset C_{i^*}$ with $i^* = \text{Rightmost}(\phi)$, then $C_j \supset \phi$ because ϕ is connected, and it cannot intersect circumference j .

ii. Assume $C_j \supset \phi$. Then, $C_j \supset C_i \forall i \in \phi$. In particular, $C_j \supset C_{i^*}$ for $i^* = \text{Rightmost}(\phi)$. Therefore, $j \in \Omega(\text{Rightmost}(\phi))$.

Theorem 2 justifies the initialization of the current region Q (more on it later) on line 8 of **Components**: Q is initialized as the set of circumferences enclosing the current connected component $\phi(i^*)$ with $i^* = \text{Rightmost}(N)$. They belong to all regions enumerated while the connected component $\phi(i^*)$ is visited.

Once a rightmost circumference $i^* \in N$ has been detected, two cases may occur, depending on whether the circumference has intersections or not: this is indicated by the flag $f(i^*)$ (line 9 of **Components**) computed in step 1. If circumference i^* has intersections, then its multigraph is visited (lines 11–14 of **Components**); if circumference i^* has no intersections, then it is directly processed (lines 17–19 of **Components**).

Multigraphs: In order to visit multigraphs, an additional data structure is needed. An integer $\mu(k)$ is associated with each vertex $k = 1, \dots, v$ to indicate the sequence in which the vertices are visited. It is initialized at zero (meaning “not visited”) in the loop on lines 4 and 5 of **Components**. A counter $\bar{\mu}$ of visited vertices is kept. It starts from zero (line 3 of **Components**); every time a vertex k is visited for the first time $\bar{\mu}$ is increased by one and $\mu(k)$ is set to $\bar{\mu}$ (see lines 12 and 13 of **Components** and lines 9 and 10 of **ScanStar**).

When the test on line 9 of **Components** succeeds, a depth-first-search algorithm is initialized. By construction, the first vertex in the sorted subset $W(i)$ is the first vertex that is encountered moving along circumference i counterclockwise starting from its rightmost point. This vertex is indicated by $W(i^*)_1$ on line 11 of **Components**. The multigraph is visited by recursive calls to the procedure **ScanStar**, shown in Algorithm 9. The initial call for each multigraph occurs on line 14 of **Components**. The depth-first-search algorithm that visits a multigraph is described in Section 3.4.1.

Isolated circumferences: When $\phi(i^*)$ consists of an isolated circumference, the internal region is computed by adding element i^* to subset Q (line 17 of **Components**) by flipping its component i^* as explained in the remainder. Then, the internal region is enumerated, that is, the single-source optimal location algorithm **Evaluate** is called (line 18); finally, i^* is deleted from N (line 19).

3.4.1. Depth-First-Search Visit to Multigraphs. In the circular array $H(k)$, representing the sorted star of each vertex $k = 1, \dots, v$, each edge incident to k has a successor (see line 4 of **ScanStar**). Exploiting this ordering, it is possible to visit all edges of the planar multigraphs induced by a set of intersecting circumferences. The visit is done with a depth-first-search algorithm. Every time a vertex is reached for the first time, its star is scanned counterclockwise starting from the successor of the edge from which the vertex has been reached. Each edge in the star of the vertex is traversed. If and only if the other endpoint of the edge has not yet been visited, then a recursive call is made to scan its star. This guarantees that each star is scanned at most once, and therefore, each edge is traversed at most twice.

The recursive procedure **ScanStar** uses three parameters: the first parameter, k , indicates the vertex whose star must be scanned; the second parameter, i , indicates the circumference of the edge traversed to reach vertex k ; the third parameter, γ , indicates the direction in which circumference i has been traversed to reach vertex k : one stands for counterclockwise, and zero stands for clockwise. These three parameters are passed by value; that is, they are local to each instance of **ScanStar**, which means that a copy is created for each call to **ScanStar**.

When **ScanStar** is called the first time in **Components** on line 14, the second parameter is the circumference with the rightmost point of the multigraph, the first parameter is the first vertex along it (the *start vertex* in the remainder), and the third parameter indicates counterclockwise.

Algorithm 9 (The Recursive Procedure that Scans the Star of a Vertex of a Multigraph)

```

1: procedure ScanStar( $k, i, \gamma$ )
2:    $t \leftarrow \text{FindEdge}(k, i, 1 - \gamma)$ 
3:   for  $p = 1, \dots, |H(k)| - 1$  do
4:      $t \leftarrow t \bmod |H(k)| + 1$ 
5:      $(j, \gamma', h) \leftarrow H(k)[t]$ 
6:      $N \leftarrow N \setminus \{j\}$ 
7:     if  $\mu(h) = 0$  then
8:       /* Forward edge */
9:        $\bar{\mu} \leftarrow \bar{\mu} + 1$ 
10:       $\mu(h) \leftarrow \bar{\mu}$ 
11:      ScanStar( $h, j, \gamma'$ )

```

```

12:   else
13:     /* Backtrack edge */
14:     Flip(j)
15:     if  $(\mu(h) < \mu(k)) \vee ((\mu(h) = \mu(k)) \wedge (\gamma' = 1))$  then
16:       /* First traversal */
17:       Evaluate(Q)
    
```

When $\text{ScanStar}(k, i, \gamma)$ is executed, the circular array $H(k)$ is searched with procedure **FindEdge** (line 2) to find the position t that corresponds to the edge that has been traversed to reach vertex k : it is the edge leaving vertex k along circumference i in a direction opposite to γ . Such an edge certainly exists and is unique because, by construction, $H(k)$ contains exactly two records $(i, 0, *)$ and $(i, 1, *)$ for each circumference i passing through vertex k .

Then, all the other $|H(k)| - 1$ edges in the star of vertex k are sequentially scanned in the loop on lines 3–17. Each edge in $H(k)$ is represented by a triple (j, γ', h) , where j is the index of the circumference to which the edge belongs, γ' indicates the direction along which the edge is traversed from vertex k to the other endpoint, and h is the index of the other endpoint (line 5).

Three cases can occur. If $\mu(h) = 0$ (line 7), then vertex h has not yet been visited; in this case, **ScanStar** is recursively called to scan the star of vertex h (line 11). Otherwise, the algorithm backtracks from h to k , and the current region Q is updated as explained in the remainder (line 14). If the test on line 15 succeeds, then the edge from k to h has been traversed for the first time (as explained later); in this case, a region is enumerated (line 17). Otherwise, the edge had already been traversed before, and the second traversal has no effect.

In all cases, when the algorithm backtracks to node k , it proceeds to the next edge in the star of vertex k counterclockwise (line 4).

3.4.1.1. Traversing the Edges. The following observations characterize some useful properties of the depth-first-search algorithm that traverses the edges of a multigraph.

Observation 2. Because vertex stars are completely scanned and because each multigraph is, by definition, connected, all vertices in the multigraphs are visited, and all edges in the multigraphs are traversed.

Observation 3. The depth-first-search algorithm defines an orientation of the edges that indicates the direction in which each edge is traversed the first time. Because all edges are traversed, all edges are oriented.

Consider a directed multigraph defined by the orientation of its edges, and let us distinguish between *forward edges* and *backtrack edges*. Consider a generic edge traversed by depth-first-search for the first time.

Let us call k its tail vertex and h its head vertex. If $\mu(h) = 0$, then the edge is a forward edge; otherwise, it is a backtrack edge.

Observation 4. Because all vertices of the multigraph are reached for the first time once, then each vertex has one forward edge entering it with the only exception of the start vertex, which has none.

Observation 5. Forward edges cannot form directed circuits because $\mu(h) > \mu(k)$ for all forward edges from k to h .

Theorem 3. *The set of forward edges forms a spanning arborescence rooted at the starting vertex.*

Proof. The proof directly follows from Observations 4 and 5.

3.4.1.2. Updating the Current Region. Let us indicate with $\text{right}(e)$ and $\text{left}(e)$ the regions on the right and left sides of a generic edge e according to its orientation. “Right” and “left” are well defined owing to the planarity of multigraphs and the unique orientation of all edges.

The algorithm uses a global variable, namely, a subset Q , representing the *current region*. The subset is assumed to be represented by a binary vector so that inserting or deleting an element is done in $O(1)$ by flipping the corresponding bit. This is done by the procedure **Flip** (see line 17 of **Components** and line 14 of **ScanStar**).

Observation 6. Two adjacent regions separated by an edge belonging to circumference j correspond to subsets that differ only by the component j .

Hence, flipping $Q[j]$ corresponds to moving from the region on one side of an edge belonging to circumference $j \in \mathcal{N}$ to the region on the other side.

The algorithm updates the current region Q according to rule 1: $Q[j]$ is flipped if and only if a backtrack occurs on an edge along circumference $j \in \mathcal{N}$ (line 14 of **ScanStar**).

3.4.1.3. Detecting Second Traversals. No attempt is made to traverse forward edges for the second time because the star $H(k)$ of the head vertex k of a forward edge e is scanned only up to the edge preceding e (see line 3 of **ScanStar**).

On the contrary, backtrack occurs twice on each backtrack edge because each backtrack edge is traversed twice by the depth-first-search algorithm. However, for the analysis of the algorithm presented hereafter it is necessary to detect when a backtrack edge is traversed for the first time and when not.

For this purpose, let us define a vertex as *open* once it has been reached by a forward edge and *closed* when its star has been completely scanned. Let us

indicate by *current* vertex the vertex k when an edge from k to h is traversed. By definition of depth-first-search, the following observation holds.

Observation 7. The current vertex is the vertex with maximum value of μ among all open vertices.

Theorem 4. If a backtrack edge is traversed the first time and its orientation is from vertex k to vertex h , then $\mu(h) \leq \mu(k)$.

Proof. When an edge from k to h is traversed the first time, k is the current vertex. Because the edge belongs also to $H(h)$ and it has not yet been traversed from h to k , this implies that h is also open. Therefore, by Observation 7, $\mu(k) \geq \mu(h)$.

As a consequence of Theorem 4, when the two endpoints of a backtrack edge are different, the second traversal of the edge can be easily detected by comparing the μ values of its endpoints: if $\mu(h) < \mu(k)$, then the backtrack edge from k to h is traversed the first time; if $\mu(h) > \mu(k)$, then the backtrack edge from k to h is traversed the second time. When $\mu(h) = \mu(k)$, the edge is a self-loop.

Observation 8. The unique vertex k of a self-loop on a circumference $i \in \mathcal{N}$ cannot be reached from any forward edge within circumference i .

This immediately follows from the observation that the rightmost point of the multigraph cannot be in the circumference. Therefore, when $H(k)$ is scanned counterclockwise, the edge corresponding to traversing the self-loop counterclockwise is always encountered before the edge corresponding to traversing the self-loop clockwise. This is also illustrated by the example shown in Figure B.2 in Online Appendix B. Hence, the test for detecting when a self-loop is traversed the first time is $\gamma' = 1$.

Justified by Theorem 4 and Observation 8, the tests on lines 7 and 15 of ScanStar correspond to rule 2: The current region Q is enumerated if and only if it is on the left side of a backtrack edge traversed the first time (line 17 of ScanStar).

3.4.1.4. Enumerating the Regions. Exploiting the planarity property of the multigraphs, the depth-first-search algorithm transforms the guarantee of traversing all edges into the guarantee of enumerating all regions within them. To prove the properties of the algorithm, we need some preliminary definitions and observations.

Let us define *forward moves* and *backward moves*, occurring, respectively, when the depth-first-search algorithm traverses an edge and when it backtracks along an edge. Let us associate a natural number v with each move corresponding to the order in which moves occur during the visit of the multigraph. Let us

indicate with $e(v)$ the edge along which move v occurs. Let us indicate with $\text{right}(v)$ and $\text{left}(v)$ the regions on the right and left sides with respect to the move. Note that $\text{right}(v) = \text{right}(e(v))$ and $\text{left}(v) = \text{left}(e(v))$ if and only if $e(v)$ is traversed for the first time, according to its orientation, and $\text{right}(v) = \text{left}(e(v))$ and $\text{left}(v) = \text{right}(e(v))$ if and only if $e(v)$ is traversed for the second time, opposite to its orientation.

Let us indicate with $R(v)$ the set of regions enumerated by the depth-first-search algorithm up to move v and by $Q(v)$ the current region when move v is done. For initialization purposes, we introduce $R(0)$ to indicate the region surrounding the current multigraph (for which there is no need to call Evaluate). We observe that $R(v') \subseteq R(v'') \forall v' < v''$ because R is only subject to insertions, not deletions.

Theorem 5. For each forward move v , $Q(v) = \text{right}(v) \in R(v-1)$ (right property). For each backward move v , $Q(v) = \text{left}(v) \in R(v)$ (left property).

Proof. The proof is by induction. We assume that the two properties hold for all moves up to move $v-1$, and we prove that they must hold for move v .

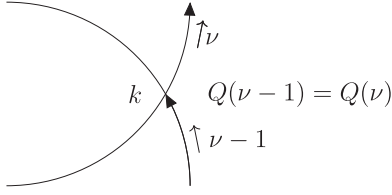
The basis of the induction is that the right property holds for $v = 1$. By the initialization of Q , $Q(1) = \Omega(i)$, where $\Omega(i)$ is the external region surrounding the current multigraph. By construction, the external region is guaranteed to be the region on the right side of the first traversed edge, that is, $\Omega(i) = \text{right}(e(1))$. The edge traversed by move $v = 1$ is certainly traversed for the first time; hence, $\text{right}(1) = \text{right}(e(1))$. By the initialization, $R(0) = \Omega(i)$. Hence, the right property holds for the first forward move.

To prove the induction step, we distinguish four cases, depending on $v-1$ and v being forward or backward moves.

Case I: Move $v-1$ is forward, and move v is forward. In this case, $e(v-1)$ and $e(v)$ belong to the star of a same vertex k , and $e(v-1)$ is the forward edge entering k . Because edge $e(v-1)$ is a forward edge, then k is reached for the first time when it is reached along edge $e(v-1)$. Therefore, edges incident to k are not traversed by any move $v' < v-1$. Then, $e(v-1)$ and $e(v)$ are traversed according to their orientations: $\text{right}(v-1) = \text{right}(e(v-1))$ and $\text{right}(v) = \text{right}(e(v))$. Edge $e(v)$ is the edge next to $e(v-1)$ in $H(k)$ counterclockwise. Hence, $\text{right}(e(v)) = \text{right}(e(v-1))$ (see Figure 5). By the induction hypothesis, $Q(v-1) = \text{right}(v-1) \in R(v-2)$. By rule 1, $Q(v) = Q(v-1)$. By construction, $R(v-2) \subseteq R(v-1)$. The combination of these equations implies $Q(v) = \text{right}(v) \in R(v-1)$. So, the right property holds for the forward move v .

Case II: Move $v-1$ is forward, and move v is backward. In this case, $e = e(v-1) = e(v)$ is a backtrack edge. If e is traversed for the first time, $\text{right}(v-1) = \text{right}(e)$

Figure 5. Case I: Both Moves $\nu - 1$ and ν Are Forward
 Moves: $\text{right}(\nu - 1) = \text{right}(\nu)$ and $Q(\nu - 1) = Q(\nu)$



and $\text{left}(\nu) = \text{left}(e)$. By the induction hypothesis, $Q(\nu - 1) = \text{right}(\nu - 1)$. By rule 1, component e is flipped: hence, $Q(\nu - 1) = \text{right}(e)$ implies $Q(\nu) = \text{left}(e)$. The combination of these equations implies $Q(\nu) = \text{left}(\nu)$ (see Figure 6). By rule 2, if e is traversed for the first time, $Q(\nu)$ is inserted in $R(\nu)$. Hence, $Q(\nu) = \text{left}(\nu) \in R(\nu)$.

If e is traversed for the second time, then $\text{right}(\nu - 1) = \text{left}(e)$, $\text{left}(\nu) = \text{right}(e)$. By the induction hypothesis, the right property holds up to $\nu - 1$, that is, $Q(\nu - 1) = \text{right}(\nu - 1) \in R(\nu - 2)$. By rule 1, component e is flipped: hence, $Q(\nu - 1) = \text{left}(e)$ implies $Q(\nu) = \text{right}(e)$. The combination of these equations implies $Q(\nu) = \text{left}(\nu)$. If $e = e(\nu - 1)$ is visited for the second time, there exists a forward move $\nu' < \nu - 1$ such that $e = e(\nu')$. By the induction hypothesis, $Q(\nu') = \text{right}(\nu') \in R(\nu' - 1)$; moreover, $\text{right}(\nu') = \text{right}(e)$ because ν' is a forward move traversing e for the first time. By rule 2, $R(\nu) = R(\nu - 1)$. Therefore, $Q(\nu) = \text{left}(\nu) = \text{right}(e) = \text{right}(\nu') \in R(\nu' - 1) \subseteq R(\nu)$. So, in both cases, the left property holds for the backward move ν .

Case III: Move $\nu - 1$ is backward, and move ν is forward. In this case, $e(\nu - 1)$ and $e(\nu)$ belong to the star of a same vertex k , and $e(\nu)$ is next to $e(\nu - 1)$ in $H(k)$ counterclockwise. Hence, $\text{left}(\nu - 1) = \text{right}(\nu)$ (independently of the orientation of the edges). For the induction hypothesis, $Q(\nu - 1) = \text{left}(\nu - 1)$, and by rule 1 $Q(\nu) = Q(\nu - 1)$. Hence, $Q(\nu) = Q(\nu - 1) = \text{left}(\nu - 1) = \text{right}(\nu)$ (see Figure 7). For the induction hypothesis, $Q(\nu - 1) \in R(\nu - 1)$. Hence, $Q(\nu) = Q(\nu - 1) \in R(\nu - 1)$. So the right property holds for the forward move ν .

Figure 6. Case II: Move $\nu - 1$ Is Forward and Move ν Is Backward Along the Same Edge e : $\text{left}(\nu - 1) = \text{left}(\nu)$, $\text{right}(\nu - 1) = \text{right}(\nu)$, and $\{e\}$ Is the Symmetric Difference Between $Q(\nu - 1)$ and $Q(\nu)$

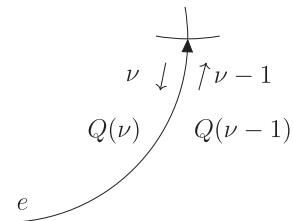
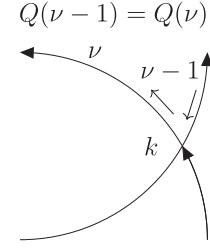


Figure 7. Case III: Move $\nu - 1$ Is Backward and Move ν Is Forward: $\text{left}(\nu - 1) = \text{right}(\nu)$ and $Q(\nu - 1) = Q(\nu)$



Case IV: Move $\nu - 1$ is backward, and move ν is backward. In this case, $e(\nu - 1)$ and $e(\nu)$ belong to the star of a same vertex k , $e(\nu)$ is the forward edge entering k , and it is next to $e(\nu - 1)$ in $H(k)$ counterclockwise. Hence, $\text{left}(\nu - 1) = \text{left}(\nu)$ (independently of the orientation of $e(\nu - 1)$). For the induction hypothesis, $Q(\nu - 1) = \text{left}(\nu - 1) \in R(\nu - 1)$. By rule 1, $Q(\nu) = Q(\nu - 1)$, and hence, $Q(\nu) = \text{left}(\nu)$ (see Figure 8). By rule 2, $R(\nu) = R(\nu - 1)$, and hence, $Q(\nu) \in R(\nu)$. So the left property holds for the backward move ν .

Observation 9. Because forward edges do not form circuits, every region must have at least one backtrack edge along its contour.

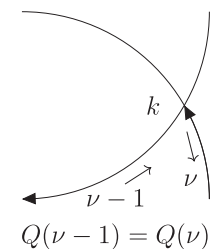
Theorem 6. All regions are enumerated.

Proof. If a region is on the left side of a backtrack edge, then it is enumerated when the backtrack edge is traversed for the first time. If a region is on the right-hand side of a backtrack edge e , then it must also be on the left side of another backtrack edge e' traversed for the first time before e because, for Theorem 5, when e is traversed for the first time the region, $\text{right}(e)$ must have been already enumerated. Because all backtrack edges are traversed, all regions are guaranteed to be enumerated.

Now we can prove that duplicate enumerations do not occur.

Lemma 1. The number of backtrack edges is equal to the number of internal regions of the multigraph.

Figure 8. Case IV: Both Moves $\nu - 1$ and ν Are Backward: $\text{left}(\nu - 1) = \text{left}(\nu)$ and $Q(\nu - 1) = Q(\nu)$



Proof. Let us indicate by E^{fw} the number of forward edges, by E^{bt} the number of backtrack edges, by $|R|$ the number of regions, and by v the number of vertices of a directed planar multigraph. By the Euler formula, $E + 2 = v + |R|$, where $E = E^{fw} + E^{bt}$. By Proposition 3, $E^{fw} = v - 1$. Therefore, $E^{bt} = |R| - 1$. Because R includes the external region that is unique, then $|R| - 1$ is the number of internal regions of the multigraph.

Theorem 7. *Each region is enumerated once.*

Proof. The proof relies on the preceding propositions: (i) all internal regions are enumerated at least once by Theorem 6, (ii) every internal region is enumerated if and only if it is found on the left side of a backtrack edge traversed the first time by rule 2, and (iii) there are as many backtrack edges as the number of internal regions by Lemma 1. Combining (i)–(iii), the theorem follows.

3.4.2. Computational Complexity. To establish the worst-case time complexity of step 4, let us consider Components first. Sorting the n elements of \mathcal{N} with SortCircles takes $O(n \log n)$.

Initializing μ takes constant time for each vertex, that is, $O(n^2)$.

The while loop (lines 6–19) is executed $O(n)$ times because at least one circumference is deleted from N at each iteration. Therefore, all constant-time operations in the loop have an overall $O(n)$ time complexity.

The total time complexity of the executions of Rightmost (line 7) is $O(n^2)$ because a sorted list of cardinality $O(n)$ produced by SortCircles must be scanned $O(n)$ times.

Initializing Q takes $O(n)$ for each component ϕ ; therefore, its overall contribution is $O(n^2)$. The overall contribution of all constant time flip operations needed to update Q (line 17 of Components and line 14 of ScanStar) is trivially bounded by twice the number of edges in the multigraphs, that is, $O(n^2)$.

The overall contribution of deletions (line 19 of Components and line 6 of ScanStar) is $O(n)$ because $|N| = n$, and each deletion takes constant time when N is implemented as a binary array.

Finally, let us examine the time complexity of ScanStar.

The time complexity of all executions of FindEdge is $O(\sum_{k=1}^v |H(k)|)$, that is, $O(n^2)$ because FindEdge is executed only once for each vertex when the vertex is reached for the first time. Therefore, the total number of steps required by FindEdge is bounded by the total degree of the multigraphs, which is $O(n^2)$ (see Theorem 1).

The total number of iterations of the loop (lines 3–17) in all executions of ScanStar is also bounded by

the total degree of the multigraphs, that is, $O(n^2)$, and the loop includes only constant time operations.

Therefore, the overall worst-case time complexity of step 4 is $O(n^2)$.

This allows establishing that the overall complexity of the new region enumeration algorithm is $O(n^2 \log n)$. The bottlenecks are the three sorting procedures in Steps 1.2, 2.2, and 3.2. This complexity analysis does not take into account the time taken by $O(n^2)$ calls to Evaluate that imply the execution of a single-source optimal location algorithm. However, the number of calls is exactly equal to the number of regions to be enumerated with no duplicates.

4. Conclusions

The possible occurrence of coincident intersection points requires correcting the SSWPLD algorithm proposed by Drezner, Mehrez, and Wesolowsky (1991) and a similar algorithm devised by Aloise, Hansen, and Liberti (2012) as pointed out by Venkateshan (2020). However, the occurrence of such pathological cases does not increase, but rather decreases the computational effort needed to enumerate all the regions induced by n circumferences in \mathbb{R}^2 . Coincident intersection points can be detected and correctly taken into account without increasing the $O(n^3)$ time complexity of the enumeration algorithms (Drezner, Mehrez, and Wesolowsky 1991, Aloise, Hansen, and Liberti 2012).

Furthermore, the computational bottleneck of these algorithms can be eliminated by enumerating the regions in a different way. The new algorithm is based on the depth-first-search visit of a set of (possibly nested) planar multigraphs, whose vertices and edges are identified by suitable sorting procedures. This provides $O(n^2 \log n)$ time complexity for enumerating all regions. Even more important, duplicate enumerations are avoided with no additional complexity, allowing executing the single-source optimal location algorithm a minimum number of times.

The algorithm presented here can be easily extended to enumerate regions induced by closed curves of many other types, such as ellipses, Cartesian ovals, and in general any kind of closed curves for which it is possible to efficiently sort points along the contour (step 2.2) and the tangent line is always defined along the contour (step 3.2).

Some interesting questions remain open for further developments.

4.1. Finite Precision Arithmetics

Devising implementations in finite precision machines, preserving correctness and complexity, is an issue common to almost all geometrical algorithms because they typically require computing and comparing irrational numbers (in equality and inequality

tests). The critical point in the new algorithm, as in previous ones, is the ability to detect when intersection points coincide.

The problem with the algorithm by Drezner, Mehrez, and Wesolowsky (1991), corrected by Venkateshan (2020), occurs when intersection points are found to coincide. Numerical approximations tend to make this unlikely: intersections that would coincide in infinite precision computations may be found noncoincident in finite precision arithmetics. In this case, numerical approximations may cause a degenerate multigraph to be analyzed as a nondegenerate one, playing the same role of small perturbations introduced on purpose as suggested by Venkateshan (2020). This would not produce wrong solutions because no region of the degenerate multigraph would be disregarded. Moreover, it would not affect the $O(n^2 \log n)$ worst-case time complexity of the new algorithm, which is the same for degenerate and nondegenerate multigraphs.

On the other side, it is also possible (although extremely unlikely) that very close but noncoincident intersection points are treated as coincident in finite precision arithmetics. However, all SSWPLD algorithms considered in this paper and its references can be made robust to these occurrences by checking the following transitive property: if two intersection points $P(i, j)$ and $P(j, k)$ coincide, then also $P(k, i)$ must coincide with them. If this does not occur, then a “numerically critical” triple of circumferences (i, j, k) is detected, and further suitable tests (with increased numerical accuracy, for instance) can be done to determine whether they intersect in the same point or not. Anyway, it should be noted that the values of $d(X, O_i)$ for each circumference $i \in \mathcal{N}$ would not be affected by more than the rounding error itself, that is, by a negligible amount.

4.2. Implementations

Implementing the new algorithm to evaluate its computational performances is also a possible topic for future research. This can lead to the development of

further algorithmic ideas. For instance, instead of evaluating the regions in the order they are enumerated, it may be profitable to evaluate them following the reverse order, that is, starting from the innermost to the outermost regions. This is because the innermost regions are more likely to contain the optimal solution than the outermost regions. It is possible to prove that the order in which regions are enumerated by the new algorithm in each multigraph corresponds to a path in the dual multigraph. This property can be exploited to record only the bit to be flipped from one region to the next one in a last-in, first-out stack, allowing for the efficient evaluation of the regions in the reverse order. Another idea is to early terminate the single-source optimal location algorithm, which iteratively updates a current point according to gradient information when the current point leaves the region to be evaluated. A third possible idea is to directly skip some region by computing a corresponding lower bound based on centers, radii, and weights without running the single-source optimal location algorithm.

Acknowledgments

Detailed and constructive comments from two anonymous referees were very useful to improve both the content and the presentation.

References

- Aloise D, Hansen P, Liberti L (2012) An improved column generation algorithm for minimum sum-of-squares clustering. *Math. Programming* 131(1–2):195–220.
- Drezner Z, Mehrez A, Wesolowsky GO (1991) The facility location problem with limited distances. *Transportation Sci.* 25(3):183–187.
- Ostresh LM Jr. (1978) On the convergence of a class of iterative methods for solving the Weber location problem. *Oper. Res.* 26(4):597–609.
- Venkateshan P (2020) A note on “The Facility Location Problem With Limited Distances.” *Transportation Sci.* 54(6):1439–1445.
- Weiszfeld A (1937) Sur le point pour lequel la somme des distances de n points donnés est minimum. *Tohoku Math. J. (2)* 34:355–386.