# A new algorithm for the Single Source Weber Problem with Limited Distances

Giovanni Righini

Dept. of Computer Science, University of Milan, Italy

giovanni.righini@unimi.it

June 5, 2021

**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 $\Re^2$ 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, i.e. 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 intersections 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.

*Keywords: Weber problem, depth-first-search.*

## 1 The problem

The Single Source Weber Problem with Limited Distances (SSWPLD), also known as Facility Location Problem with Limited Distances, is a continuous optimization problem in location theory. A set $\mathcal{N} = \{1, 2, \ldots, n\}$ of circumferences in $\Re^2$ 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 $\Re^2$ 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:

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

where $d()$ indicates the Euclidean distance in $\Re^2$.

In 1991 Drezner et al. [2] proposed an algorithm to solve the SSWPLD as an unrestricted single-source location problem for each region of the partition of $\Re^2$ induced by the circumferences. A region is defined by the subset of circumferences including it. Hence the objective can be restated as follows:

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

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 $Q' = Q \backslash \{i\}$.

Indicating with $R$ the set of regions of $\Re^2$ induced by the circumferences, the SSWPLD can be reformulated as

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

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 et al. [2].

The single-source optimal location problem, or 1-median problem, can be solved by the classical infinite algorithm proposed by Weiszfeld [5] or one of its variations (e.g. Ostresh [3]).

In this paper the focus is on the complexity of the region enumeration algorithm, building upon the papers by Drezner et al. [2] and Venkateshan [4].

The algorithm proposed by Drezner et al. relies upon a theorem stating that $n$ circumferences in $\Re^2$ 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 et al. 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, i.e. $O(n^2)$ 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 et al. [1] showed 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 et al..

More recently, Venkateshan [4] 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 Venkateshan’s algorithm, given an intersection point $P$ between circumferences, a subset $S_P$ is defined as the subset of circumferences *strictly* covering $P$, while 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, since the same result obtained by Venkateshan’s method can be achieved with better computational complexity without actually constructing the neighborhoods, but just distinguishing the two sides of the circumferences in $T_P$, i.e. 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 to compute the subsets $S_P$ and it allows to enumerate 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

**Pre-processing.** A generic SSWPLD instance can be pre-processed 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 $w = 0$, since they have no effect on the value of any solution. Such a pre-processing takes $O(n \log n)$ and it is not a computational complexity bottleneck.

In the remainder the term "multiple intersection point" (abbreviated in m.i.p.) is used to indicate a point in $\Re^2$ where two or more circumferences intersect.

Given a m.i.p. $P$ and the corresponding subset $T_P$ of circumferences that intersect in $P$, the enumeration method proposed in [4] 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 pre-processing.

Given a m.i.p $P$ and given a small enough neighborhood of $P$ with the property above, let us indicate its radius with $\rho_P$ and its frontier with $F_P$. For the definition of small enough neighborhood and since $\rho_P$ is guaranteed to be strictly positive, the following observation holds.

**Observation 1** *Given a 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 to scan $F_P$ according to an arbitrary orientation (e.g. counter-clockwise) 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 $\overline{e}_i$ and $\overline{l}_i$ the directions from $P$ to the intersection points between $F_P$ and each circumference $i \in T_P$, as shown in Figure 1. Assuming to scan $F_P$ counter-clockwise, the intersection point corresponding to $\overline{e}_i$ is encountered when "entering" circumference $i \in \mathcal{N}$ and the intersection point corresponding to $\overline{l}_i$ is encountered when "leaving" it.

Obviously, the cyclic order of the intersection points is equal to the cyclic order of the corresponding directions $\overline{e}$ and $\overline{l}$. We indicate such a cyclic order by $\overline{L}_P$. Note that $\overline{L}_P$ does not depend on $\rho_P$, 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 [4]. Here we observe that

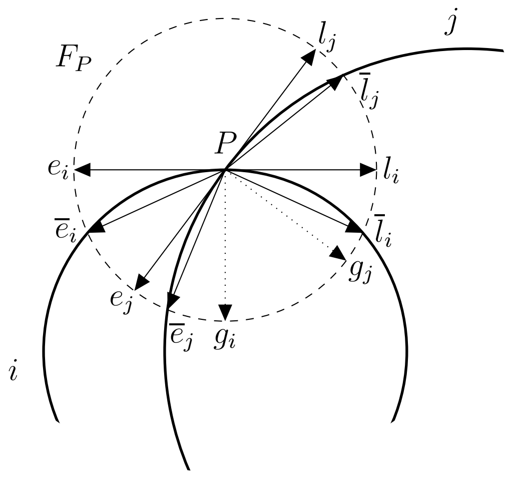


Figure 1: An intersection point $P$ between two circumferences, its neighborhood, the intersections along its frontier $F_P$, the directions $\overline{e}$, $\overline{l}$, $e$ and $l$. In this example $\overline{L}_P = \{\overline{l}_j, \overline{e}_i, \overline{e}_j, \overline{l}_i\}$.

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 tangent to circumference $i$ in $P$ corresponding to "entering" ($e_i$) and "leaving" ($l_i$) the circumference when $F$ is scanned counter-clockwise: $e_i = g_i - \frac{\pi}{2}$ and $l_i = g_i + \frac{\pi}{2}$, where all angles are computed modulo $2\pi$.

Since $F_P$ is continuous and the circumferences are continuous, when $\rho_P$ tends to $0$ the intersection points on $F_P$ tend to $P$ and then $\overline{e}_i$ tends to $e_i$ and $\overline{l}_i$ 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 $\overline{L}_P$, i.e. these two properties hold: (i) $L_P$ can be obtained from $\overline{L}_P$ by replacing $\overline{e}_i$ with $e_i$ and $\overline{l}_i$ 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 counter-clockwise.

Ties do not exist in the cyclic order of directions $\overline{e}$ and $\overline{l}$, 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 $\overline{L}_P$ 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 to sort the directions $e$ and $l$ in a uniquely defined cyclic order $L_P$ equal to the unique cyclic order $\overline{L}_P$ of the directions $\overline{e}$ and $\overline{l}$. In turn, this allows to compute $L_P$ without computing $\overline{L}_P$ and to obtain from $L_P$ the same piece of information that can be obtained from $\overline{L}_P$.

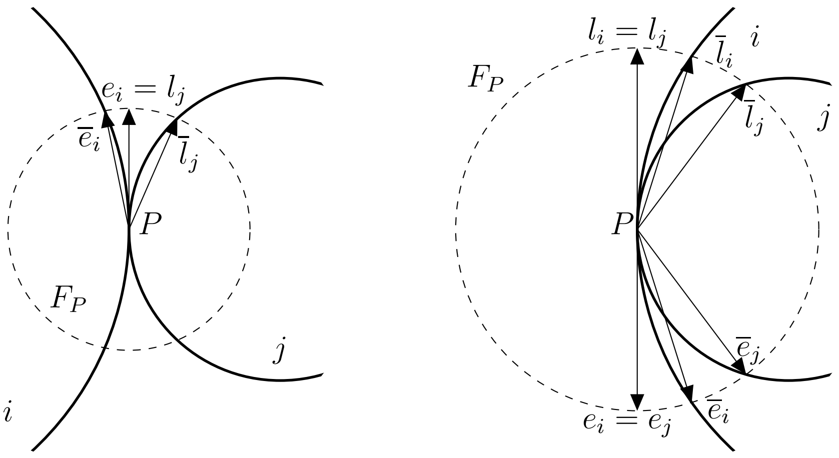


Figure 2: Tie-break criterion 1 (left): when leaving a circumference $j$ and entering a circumference $i$ with $g_i = g_j \pm \pi$, direction $\overline{l}_j$ is encountered before $\overline{e}_i$. Tie-break criterion 2 (right): when leaving circumferences $i$ and $j$ with $g_i = g_j$ and $r_i > r_j$,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

Venkateshan [4] described an algorithm to obtain the set of distinct regions around a m.i.p. $P$, once the cyclic order $\overline{L}_P$ 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 [4]; 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 3-8), for each element a circumference index in $\{1, \ldots, n\}$ 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, then 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 $t = 1, \ldots, |V_P|$ 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:                 $Q \leftarrow Q \backslash \{-V_P[t]\}$
 9:     for $t = 1, \ldots, |V_P|$ do
10:         if $V_P[t] > 0$ then
11:             $Q \leftarrow Q \cup \{V_P[t]\}$
12:         else
13:             $Q \leftarrow Q \backslash \{-V_P[t]\}$
14:         Evaluate($Q \cup S_P$)
```

Three examples are provided in Appendix 1, 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.

**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 to check whether two circumferences already belong to $T_P$ and to insert 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 $O(n^2 \log n)$.

**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 et al. [2] and Aloise et al. [1].

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

Consider a 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 $O(c)$, since 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 a m.i.p. $P$, where $c > 2$ 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, since 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 multi-graph $\mathcal{M} = (\mathcal{V}, \mathcal{E})$, defined by $n$ intersecting circumferences, where $\mathcal{V}$ 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 multi-graph $\tilde{\mathcal{M}} = (\tilde{\mathcal{V}}, \tilde{\mathcal{E}})$ obtained by a small perturbation of the circumferences at the m.i.p. where more than two circumferences intersect, so that no degenerate intersections occur in $\tilde{\mathcal{M}}$. Then, all vertices in $\tilde{\mathcal{M}}$ have degree $4$. 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 $n(n-1)$. Since in $\tilde{\mathcal{M}}$ all vertices have degree $4$, the total degree in $\tilde{\mathcal{M}}$ is bounded by $4n(n-1)$. Since each edge has two endpoints, then $|\tilde{\mathcal{E}}| \leq 2n(n-1)$. By construction, all edges of $\mathcal{M}$ have a counterpart in $\tilde{\mathcal{M}}$, while 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 $2n(n-1)$. $\square$

*Proof 2.* For any given planar multi-graph $\mathcal{M} = (\mathcal{V}, \mathcal{E})$ inducing a set of regions $\mathcal{R}$ in $\Re^2$, Euler formula holds: $|\mathcal{E}| + 2 = |\mathcal{V}| + |\mathcal{R}|$. By Drezner et al. theorem $|\mathcal{R}|$ is $O(n^2)$. Since $|\mathcal{V}|$ is also $O(n^2)$, then is $|\mathcal{E}|$ is $O(n^2)$. $\square$

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

This immediately follows from Theorem 1, since 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. $k = 1, \ldots, K$. Since $c_k \leq n \; \forall k = 1, \ldots, K$, and hence $\log c_k \leq \log n \; \forall k = 1, \ldots, K$, 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 multi-graph $\mathcal{M}$ defined above. For Corollary 1, such a total degree grows as $O(n^2)$. Therefore an aymptotic 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 SSW-PLD 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 due to degenerate intersections (affecting Steps 1 and 3), but rather to 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 multi-graphs in $\Re^2$. Their *vertices* are m.i.p., i.e. 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 multi-graphs induced by the circumferences are planar by definition, i.e. 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 multi-graphs. In Step 2 the vertices occurring along each circumference are sorted according to a given orientation and this allows to identify all edges of the multi-graphs and to compute 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 multi-graph 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 multi-graphs mentioned above, it is necessary to find their vertices, i.e. all subsets of coincident intersection points. For the sake of clarity, the description of this step is broken into three sub-steps.

### 3.1.1 Step 1.1: Enumeration of intersection points

The first sub-step 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 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 $\Lambda$ of all intersection points is produced, by considering all pairs of distinct circumferences.

All sets $\Omega$ 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 $\Omega$ 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 counter-clockwise. Then, as shown in Figure 3, $P(i, j)$ is where circumference $i$ enters circumference $j$ and circumference $j$ leaves circumference $i$, while the converse occurs in $P(j, i)$. The coordinates of the two intersection points are computed in constant time by a suitable function $\mathsf{Intersect}()$ (line 15). Then, they are added to the list $\Lambda$ of all intersection points (line 18). Each element of $\Lambda$ 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 $\Omega$, $f$ and $\Lambda$.

**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:         $f(i) \leftarrow$ 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 $(d(O_i, O_j) \le r_i + r_j)$ then
15:                     $[P(i,j), P(j,i)] \leftarrow$ Intersect$(i,j)$
16:                     $f(i) \leftarrow$ true
17:                     $f(j) \leftarrow$ true
18:                     $\Lambda \leftarrow \Lambda \cup \{[i, j, x(P(i,j)), y(P(i,j))], [j, i, x(P(j,i)), y(P(j,i))]\}$
```

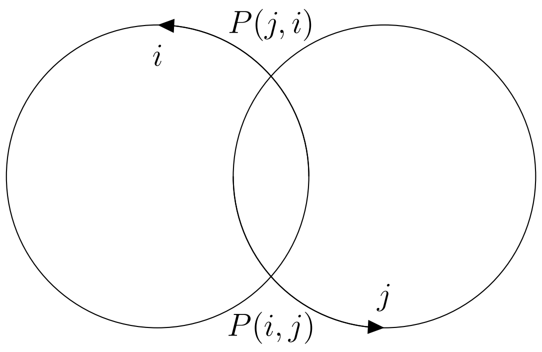


Figure 3: Intersection points between two circumferences.

The complexity of Intersections is $O(n^2)$ due to the two nested loops (lines 6-7) 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 $\Lambda$ 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 multi-graphs induced by the intersecting circumferences.

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

Since $|\Lambda|$ 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 $\Lambda$ is transformed into an array. The sorted array $\Lambda$ 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 position 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 multi-graphs.

```
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 $h = t', \ldots, t'' - 1$ do
13:             $T(v) \leftarrow T(v) \cup \{\Lambda[h].i, \Lambda[h].j\}$
14:         $t' \leftarrow t''$
```

For each vertex $k = 1, \ldots, v$, $x(k)$ and $y(k)$ are its coordinates while $T(k)$ 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.

Since $|\Lambda|$ 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 counter-clockwise. For the sake of clarity, the description of Step 2 is broken into three sub-steps.

### 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 $k = 1, \ldots, v$ 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 $k = 1, \ldots, v$ 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 2 to the total degree of the multi-graphs. Since the total degree of the multi-graphs 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 $\alpha$. 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 $\alpha$. 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 $i = 1, \ldots, n$ do
3:         for $k \in W(i)$ do
4:             $\alpha(i, k) \leftarrow \arctan(O_i, (x(k), y(k)))$
5:         $W(i) \leftarrow \mathsf{Sort}(W(i))$
```

As already shown in Subsubsection 3.2.1, the number of $(i, k)$ pairs in the multi-graphs 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 $i \in \mathcal{N}$. Since $|W(i)| \leq 2(n-1)\ \forall i \in \mathcal{N}$ 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 multi-graphs 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, i.e. 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 $(i, \gamma, h)$, where $i$ is the index of a circumference passing through the vertex, $\gamma$ is a bit representing "counter-clockwise" with $1$ and "clockwise" with $0$, and $h$ is the index of the vertex that is reached from vertex $k$ following circumference $i$ in direction $\gamma$.

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 counter-clockwise edge entering $k''$ and in $H(k'')$ as a clockwise edge entering $k'$. As a special case, it is possible that $W(i)$ contain a single vertex $k$. In this case two edges are inserted in $H(k)$ with opposite directions $\gamma$ 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:             $k'' \leftarrow \mathsf{succ}(k')$
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 above. The total number of elements in subsets $H$ is twice the total number of edges in the multi-graphs since 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 multi-graphs 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 multi-graphs. This step is necessary to enumerate the regions while visiting the multi-graphs.

For the sake of clarity, the description of Step 3 is broken into two sub-steps.

### 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 $k = 1, \ldots, v$: it is the direction of the line tangent to the circumference $i$ in vertex $k$, oriented *from* the vertex in direction $\gamma$. The tangent certainly exists, because pre-processing guarantees that all circumferences have strictly positive radius. As before, angles are computed counter-clockwise 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 $\beta$ to the three-field records $(i, \gamma, h)$ in the subset $H(k)$ $\forall k = 1, \ldots, v$ (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 $k = 1, \ldots, v$ 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 $(i, \gamma, h, \beta)$
```

Since 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 $k = 1, \ldots, v$, its star $H(k)$ is sorted counter-clockwise, according to the values of the angle $\beta$ 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 $H(k)$.

The above 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 $k = 1, \ldots, v$ is managed as a circular array, so that the successor of the last element is the first one.

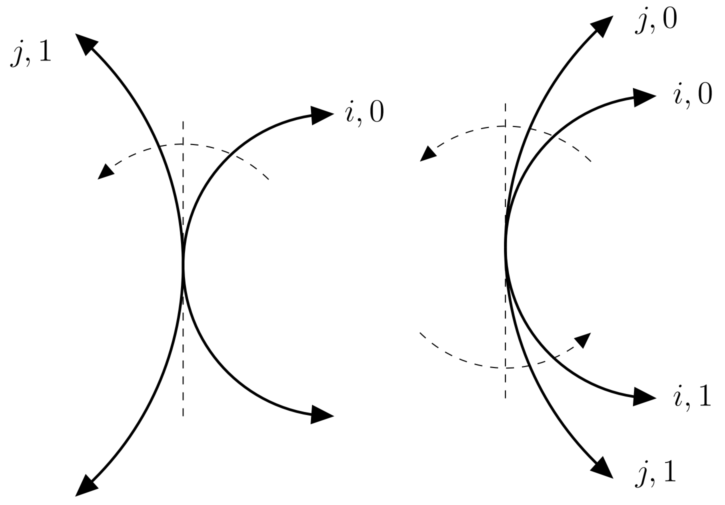


Figure 4: Sorting edges with the same tangent line in a vertex star. Left: Tie-break criterion 3 is applied to break ties between $(i, 0, \beta)$ and $(j, 1, \beta)$. Right: Tie-break criterion 4 is applied to break ties between $(i, 0, \beta)$ and $(j, 0, \beta)$ and between $(i, 1, \beta)$ and $(j, 1, \beta)$.

The effect of Step 3 is to sort the edges in $H(k)$ in the same order as they would be encountered moving counter-clockwise 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 there are $|H(k)|$ edges in each vertex star. The complexity for sorting all vertex stars is $O(\sum_{k=1}^{v} |H(k)| \log |H(k)|)$. Since $|H(k)| \leq 2n \ \forall k = 1, \ldots, v$ 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.

## 3.4 Step 4: Region enumeration

Besides multi-graphs, the set of given circumferences of a SSWPLD instance may also contain isolated circumferences, i.e. circumferences with no intersections with others. It is also possible that some multi-graphs or isolated circumferences are completely enclosed in other multi-graphs 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 multi-graph 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 $\Re^2$ 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, where $z$ attains its maximum value $\sum_{i\in\mathcal{N}} w_i r_i$.

Non-overlapping 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 multi-graph, 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:     $N \leftarrow$ SortCircles($\mathcal{N}$)
 3:     $\overline{\mu} \leftarrow 0$
 4:     for $k \in 1, \ldots, v$ do
 5:         $\mu(k) \leftarrow 0$
 6:     while $N \neq \emptyset$ do
 7:         $i^* \leftarrow$ Rightmost($N$)
 8:         $Q \leftarrow \Omega(i^*)$
 9:         if $f(i^*)$ then
10:             /* Multi-graph */
11:             $k \leftarrow W(i^*)_1$
12:             $\overline{\mu} \leftarrow \overline{\mu} + 1$
13:             $\mu(k) \leftarrow \overline{\mu}$
14:             ScanStar($k, i^*, 1$)
15:         else
16:             /* Isolated circumference */
17:             Flip($i^*$)
18:             Evaluate($Q$)
19:             $N \leftarrow N \backslash \{i^*\}$
```

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 non-increasing value of the abscissa of their rightmost point. In case of ties, the circumferences are sorted by decreasing values of their radius. In case of 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 $i \in \mathcal{N}$, i.e. $C_i = \{i\}$. 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 connnected component $\phi$, $C \sqsupset \phi$ if and only if all points of $\phi$ are within $C$ and no point of $C$ belongs to $\phi$. Recall that $\Omega(i) \subset \mathcal{N}$, computed in Step 1, is the subset of circumferences that strictly enclose each circumference $i \in \mathcal{N}$, i.e. $\Omega(i) = \{j \in \mathcal{N} : C_j \sqsupset C_i\} \; \forall i \in \mathcal{N}$. 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 $\phi(i)$:*

$$j \in \Omega(\mathsf{Rightmost}(\phi)) \Leftrightarrow C_j \sqsupset \phi.$$

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

(ii) Assume $C_j \sqsupset \phi$. Then $C_j \sqsupset C_i \; \forall i \in \phi$. In particular $C_j \sqsupset C_{i^*}$ for $i^* = \mathsf{Rightmost}(\phi)$. Therefore $j \in \Omega(\mathsf{Rightmost}(\phi))$. $\Box$

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 $i^* = \mathsf{Rightmost}(N)$. 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 multi-graph is visited (lines 11-14 of Components); if circumference $i^*$ has no intersections, then it is directly processed (lines 17-19 of Components).

**Multi-graphs.** In order to visit multi-graphs, an additional data-structure is needed. An integer $\mu(k)$ is associated with each vertex $k = 1, \ldots, v$, to indicate the sequence in which the vertices are visited. It is initialized at $0$ (meaning "not visited") in the loop on lines 4-5 of Components. A counter $\overline{\mu}$ of visited vertices is kept. It starts from $0$ (line 3 of Components); every time a vertex $k$ is visited for the first time $\overline{\mu}$ is increased by $1$ and $\mu(k)$ is set to $\overline{\mu}$ (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 $i$ counter-clockwise starting from its rightmost point. This vertex is indicated by $W(redi^*)_1$ on line 11 of Components. The multi-graph is visited by recursive calls to the procedure ScanStar, shown in Algorithm 9. The initial call for each multi-graph occurs on line 14 of Components. The depth-first-search algorithm that visits a multi-graph is described in Subsubsection 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, i.e. 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 multi-graphs

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 multi-graphs 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 counter-clockwise 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, $\gamma$, indicates the direction in which circumference $i$ has been traversed to reach vertex $k$: $1$ stands for "counter-clockwise" and $0$ stands for "clockwise". These three parameters are passed by value, i.e. 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 multi-graph, the first parameter is the first vertex along it (the *start vertex*, in the remainder) and the third parameter indicates "counter-clockwise".

When 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 direction opposite to $\gamma$. 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 $(j, \gamma', h)$, where $j$ is the index of the circumference to which the edge belongs, $\gamma'$ indicates the direction along which the edge

**Algorithm 9** The recursive procedure that scans the star of a vertex of a multi-graph.

```
1: procedure ScanStar($(k, i, \gamma)$)
2:     $t \leftarrow$ 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:         $N \leftarrow N \backslash \{j\}$
7:         if $\mu(h) = 0$ then
8:             /* Forward edge */
9:             $\overline{\mu} \leftarrow \overline{\mu} + 1$
10:            $\mu(h) \leftarrow \overline{\mu}$
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$(Q)$
```

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 $k$ counter-clockwise (line 4).

**Traversing the edges.** The following observations characterize some useful properties of the depth-first-search algorithm that traverses the edges of a multi-graph.

**Observation 2** *Since vertex stars are completely scanned, and since each multi-graph is by definition connected, all vertices in the multi-graphs are visited and all edges in the multi-graphs 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. Since all edges are traversed, all edges are oriented.*

Consider a directed multi-graph 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** *Since all vertices of the multi-graph 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, since $\mu(h) > \mu(k)$ 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 Observation 4 and Observation 5. □

**Updating the current region.** Let us indicate with $\mathsf{right}(e)$ and $\mathsf{left}(e)$ the regions on the right side and the left side of a generic edge $e$ according to its orientation. "Right" and "left" are well-defined owing to the planarity of multi-graphs 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 the following rule.

**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*).*

**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, since 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 $\mu$ 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. Since 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 $\mu$ 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 multi-graph cannot be in the circumference. Therefore, when $H(k)$ is scanned counter-clockwise the edge corresponding to traversing the self-loop counter-clockwise is always encountered before the edge corresponding to traversing the self-loop clock-wise. This is also illustrated by the example shown in Figure 17 in Appendix 2. Hence, the test for detecting when a self-loop is traversed the first time is $\gamma' = 1$.

Justified by the Theorem 4 and Observation 8, the tests on lines 7 and 15 of ScanStar correspond to the following rule.

**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*).*

**Enumerating the regions.** Exploiting the planarity property of the multi-graphs, 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 $\nu$ with each move corresponding to the order in which moves occur during the visit of the multi-graph. Let us indicate with $e(\nu)$ the edge along which move $\nu$ occurs. Let us indicate with $\mathsf{right}(\nu)$ and $\mathsf{left}(\nu)$ the regions on the right side and the left side with respect to the move. Note that $\mathsf{right}(\nu) = \mathsf{right}(e(\nu))$ and $\mathsf{left}(\nu) = \mathsf{left}(e(\nu))$ if and only if $e(\nu)$ is traversed for the first time, according to its orientation, while $\mathsf{right}(\nu) = \mathsf{left}(e(\nu))$ and $\mathsf{left}(\nu) = \mathsf{right}(e(\nu))$ if and only if $e(\nu)$ is traversed for the second time, opposite to its orientation.

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

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

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

Basis of the induction: the right property holds for $\nu = 1$. By the initialization of $Q$, $Q(1) = \Omega(i)$, where $\Omega(i)$ is the external region surrounding the current multi-graph. By construction, the external region is guaranteed to be the region on the right side of the first traversed edge, i.e. $\Omega(i) = \mathsf{right}(e(1))$. The edge traversed by move $\nu = 1$ is certainly traversed for the first time; hence $\mathsf{right}(1) = \mathsf{right}(e(1))$. 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 $\nu - 1$ and $\nu$ being forward or backward moves.

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

Case II: move $\nu - 1$ is forward and move $\nu$ is backward. In this case $e = e(\nu - 1) = e(\nu)$ is a backtrack edge.

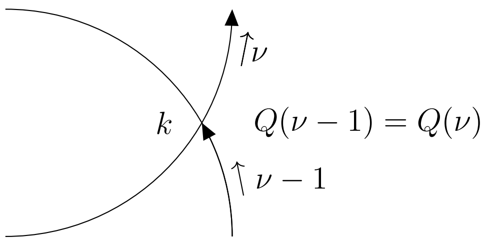


Figure 5: Case I: both moves $\nu - 1$ and $\nu$ are forward moves: $\mathsf{right}(\nu-1) = \mathsf{right}(\nu)$ and $Q(\nu-1) = Q(\nu)$.

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

If $e$ is traversed for the second time, then $\mathsf{right}(\nu-1) = \mathsf{left}(e)$, $\mathsf{left}(\nu) = \mathsf{right}(e)$. By the induction hypothesis, the right property holds up to $\nu - 1$, i.e. $Q(\nu-1) = \mathsf{right}(\nu-1) \in R(\nu-2)$. By Rule 1, component $e$ is flipped: hence $Q(\nu-1) = \mathsf{left}(e)$ implies $Q(\nu) = \mathsf{right}(e)$. The combination of the equations above implies $Q(\nu) = \mathsf{left}(\nu)$. 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, $Q(\nu') = \mathsf{right}(\nu') \in R(\nu'-1)$; moreover $\mathsf{right}(\nu') = \mathsf{right}(e)$, because $\nu'$ is a forward move traversing $e$ for the first time. By Rule 2, $R(\nu) = R(\nu-1)$. Therefore $Q(\nu) = \mathsf{left}(\nu) = \mathsf{right}(e) = \mathsf{right}(\nu') \in R(\nu'-1) \subseteq R(\nu)$.

So, in both cases the left property holds for the backward move $\nu$.

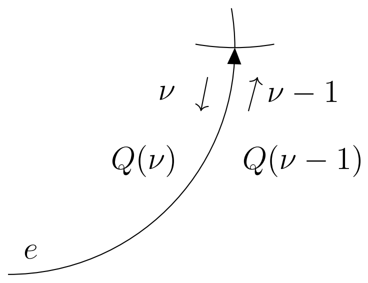


Figure 6: Case II: move $\nu - 1$ is forward and move $\nu$ is backward along the same edge $e$: $\mathsf{left}(\nu-1) = \mathsf{left}(\nu)$, $\mathsf{right}(\nu-1) = \mathsf{right}(\nu)$ and $\{e\}$ is the symmetric difference between $Q(\nu-1)$ and $Q(\nu)$.

Case III: move $\nu - 1$ is backward and move $\nu$ 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 $H(k)$ counter-clockwise. Hence $\mathsf{left}(\nu - 1) = \mathsf{right}(\nu)$ (independently of the orientation of the edges). For the induction hypothesis $Q(\nu - 1) = \mathsf{left}(\nu - 1)$ and by Rule 1 $Q(\nu) = Q(\nu - 1)$. Hence $Q(\nu) = Q(\nu - 1) = \mathsf{left}(\nu - 1) = \mathsf{right}(\nu)$ (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 $\nu$.

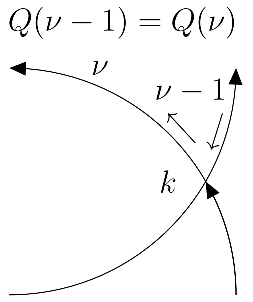


Figure 7: Case III: move $\nu - 1$ is backward and move $\nu$ is forward: $\mathsf{left}(\nu - 1) = \mathsf{right}(\nu)$ and $Q(\nu - 1) = Q(\nu)$.

Case IV: move $\nu - 1$ is backward and move $\nu$ 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 $H(k)$ counter-clockwise. Hence $\mathsf{left}(\nu - 1) = \mathsf{left}(\nu)$ (independently of the orientation of $e(\nu - 1)$). For the induction hypothesis $Q(\nu - 1) = \mathsf{left}(\nu - 1) \in R(\nu - 1)$. By Rule 1, $Q(\nu) = Q(\nu - 1)$ and hence $Q(\nu) = \mathsf{left}(\nu)$ (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 $\nu$. $\square$

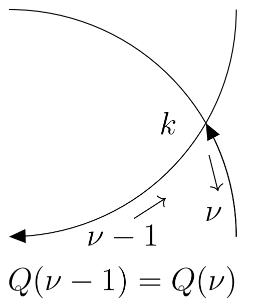


Figure 8: Case IV: both moves $\nu - 1$ and $\nu$ are backward: $\mathsf{left}(\nu - 1) = \mathsf{left}(\nu)$ and $Q(\nu - 1) = Q(\nu)$.

**Observation 9** *Since 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 $\mathsf{right}(e)$ must have been already enumerated. Since 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 multi-graph.*

*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 multi-graph. By Euler formula, $E + 2 = v + |R|$, where $E = E^{fw} + E^{bt}$. By Proposition 3, $E^{fw} = v - 1$. Therefore $E^{bt} = |R| - 1$. Since $R$ includes the external region which is unique, then $|R| - 1$ is the number of internal regions of the multi-graph. □

**Theorem 7** *Each region is enumerated once.*

*Proof.* The proof relies on the propositions above: (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; (iii) there are as many backtrack edges as the number of internal regions, by Lemma 1. Combining (i), (ii) and (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 $\mu$ takes constant time for each vertex, i.e. $O(n^2)$.

The while loop (lines 6-19) is executed $O(n)$ times, since 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 $O(n^2)$, since a sorted list of cardinality $O(n)$, produced by SortCircles must be scanned $O(n)$ times.

Initializing $Q$ takes $O(n)$ for each component $\phi$; 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 multi-graphs, i.e. $O(n^2)$.

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 $O(\sum_{k=1}^{v} |H(k)|)$, i.e. $O(n^2)$, 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 multi-graphs, 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 multi-graphs, i.e. $O(n^2)$ and the loop includes only constant time operations.

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

This allows to establish that the overall complexity of the new region enumeration algorithm is $O(n^2 \log n)$. The bottlenecks are the three sorting procedures in Step 1.2, Step 2.2 and Step 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 to correct the SSWPLD algorithm proposed by Drezner et al. [2] and a similar algorithm devised by Aloise et al. [1], as pointed out by Venkateshan [4]. 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 $\Re^2$. Coincident intersection points can be detected and correctly taken into account without increasing the $O(n^3)$ time complexity of the enumeration algorithms [2, 1].

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 multi-graphs, 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 to execute 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 fur further developments.

**Finite precision arithmetics.** Devising implementations in finite precision machines, preserving correctness and complexity, is an issue common to almost all geometrical algorithms, since they typically require to compute and compare 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 et al., corrected by Venkateshan, 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 non-coincident in finite precision arithmetics. In this case numerical approximations may cause a degenerate multi-graph to be analyzed as a non-degenerate one, playing the same role of small perturbations introduced on purpose, as suggested by Venkateshan [4]. This would not produce wrong solutions, since no region of the degenerate multi-graph 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 non-degenerate multi-graphs.

On the other side, it is also possible (although extremely unlikely) that very close but non-coincident 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, i.e. by a negligible amount.

**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 furher 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, i.e. starting from the innermost to the outermost regions. This is because innermost regions are more likely to contain the optimal solution then outermost regions. It is possible to prove that the order in which regions are enumerated by the new algorithm in each multi-graph corresponds to a path in the dual multi-graph. 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 a 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

[1] Aloise, D., P. Hansen, L. Liberti. 2012. An Improved Column Generation Algorithm for Minimum Sum-of-Squares Clustering. Mathematical Programming. 131.1-2. 195-220.

[2] Drezner, Z., A. Mehrez, G.O. Wesolowsky. 1991. The Facility Location Problem with Limited Distances. Transportation Science. 25.3. 183-187.

[3] Ostresh Jr., L.M. 1978. On the Convergence of a Class of Iterative Methods for Solving the Weber Location Problem. Operations Research. 26.4. 597-609.

[4] Venkateshan P. 2020. A Note on “The Facility Location Problem with Limited Distances”. Transportation Science.

[5] Weiszfeld A. 1937. Sur le point pour lequel la somme des distances de $n$ points donnés est minimum, Tohoku Mathematical Journal 34. 355-386.

# Appendix A Three examples

## A.1 Example 1.

In Example 1 (taken from [4]), illustrated in Figure 9, four circumferences share two m.i.p..

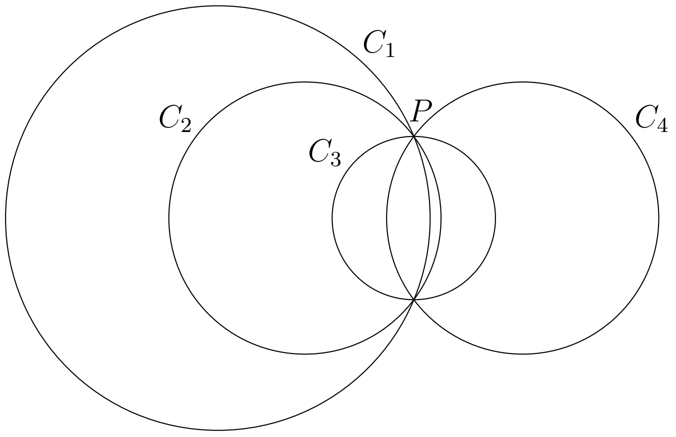


Figure 9: Example 1 [4]: four circumferences sharing two m.i.p..

As pointed out in [4], the algorithms proposed in [2] and [1] would fail to enumerate all regions around $P$. For every pair of circumferences intersecting in $P$, the subset $S_P$, as defined by [2] and [1], would include the other two circumferences; therefore the regions covered by a single circumference ($C_1$ or $C_4$) would be missed in the enumeration. On the contrary, constructing the neighborhood of $P$ shown in Figure 10 on the left and its eight intersection points with $C_1, \ldots, C_4$, one can correctly enumerate all regions around $P$.

Figure 10 on the right shows how the same remedy provided by constructing the neighborhood can be obtained by considering the tangent lines and their orientation. Assume to store the angles corresponding to the directions $e_i$ and $l_i$ for each $i = 1, \ldots, 4$ in a set of $8$ elements and then to sort it, obtaining a cyclic order. In Example 1 we obtain the cyclic order $L = \{l_3, l_4, e_1, e_2, e_3, e_4, l_1, l_2\}$. Now, replacing each $l_i$ with $-i$ and each $e_i$ with $+i$, the vector $V_P = \{-3, -4, +1, +2, +3, +4, -1, -2\}$ is obtained. Then, procedure $\mathsf{Scan}$ shown in Algorithm 1 is executed: after the first scan of $V_P$ it yields $Q = \{3, 4\}$; during the second scan of $V_P$ the following set of regions is found: $\{\{4\}, \emptyset, \{1\}, \{1, 2\}, \{1, 2, 3\}, \{1, 2, 3, 4\}, \{2, 3, 4\}, \{3, 4\}\}$.

Example 1 illustrates a case of a m.i.p. $P$, but it is "easy" in the sense that all tangent lines have distinct angles in $P$, so that there is no need to use tie-break criteria.

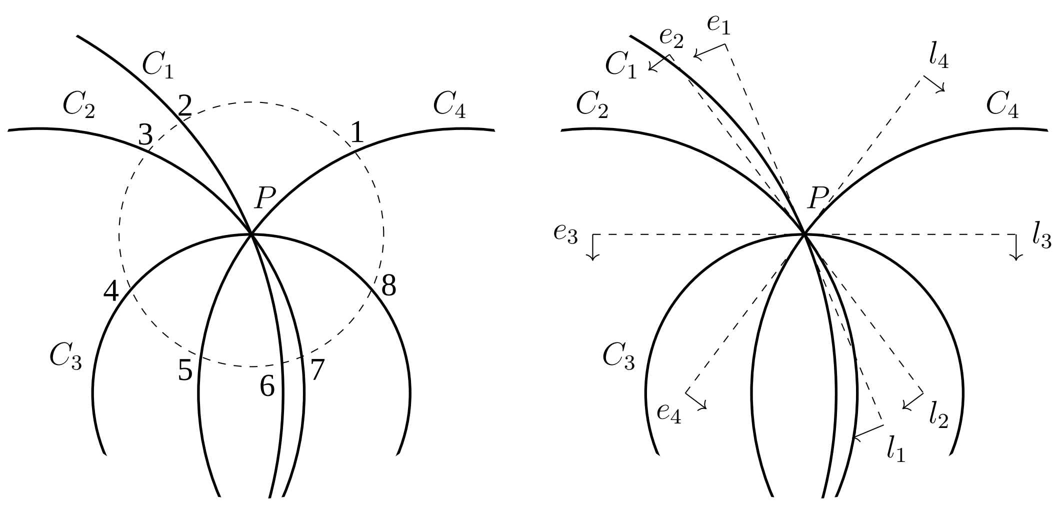


Figure 10: Left: a neighborhood of a m.i.p., as defined in [4]. Right: the tangent lines. For each tangent line two little arrows indicate the side where the center lies. The order in which the intersection points numbered $1, \ldots, 8$ are encountered following the frontier of the neighborhood on the left corresponds to the order in which the directions $e_1, \ldots, e_4$ and $l_1, \ldots, l_4$ on the right appear when they are sorted according to their angles.

## A.2 Example 2.

In Example 2 (taken from [4]), illustrated in Figure 11, four circumferences share a single m.i.p..

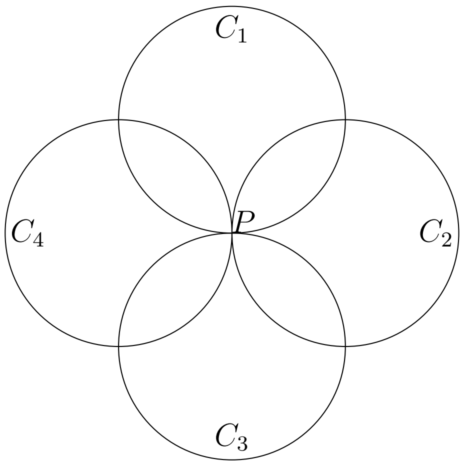


Figure 11: Example 2 [4]: four circumferences sharing a single m.i.p..

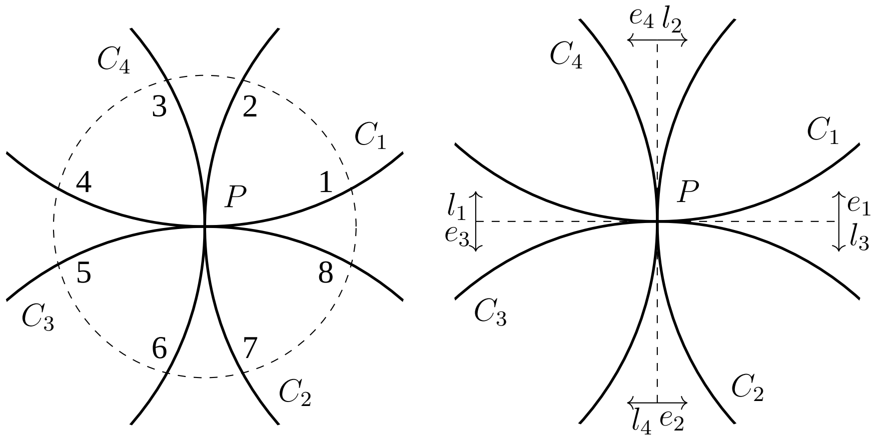


Figure 12: Left: a neighborhood of the m.i.p.. Right: the tangent lines.

Figure 12 compares the actual construction of the neighborhood of the m.i.p. with the more efficient sorting of the directions of the tangent lines. Assume to store the angles corresponding to the directions $e_i$ and $l_i$ for each $i = 1, \ldots, 4$ in a set of $8$ elements and then to sort it obtaining a cyclic order. The cyclic order $L_P = \{[l_3, e_1], [l_2, e_4], [l_1, e_3], [l_4, e_2]\}$ is obtained. Pairs indicated in square brackets have the same value of the angle: tie-break criterion 1 must be used to sort them correctly. Replacing each $l_i$ with $-i$ and each $e_i$ with $+i$ the vector $V_P = \{-3, +1, -2, +4, -1, +3, -4, +2\}$ is obtained. After that, Algorithm 1 is executed: after the first scan of $V_P$ it yields $Q = \{2, 3\}$; during the second scan the following set of regions is found $\{\{2\}, \{1, 2\}, \{1\}, \{1, 4\}, \{4\}, \{3, 4\}, \{3\}, \{2, 3\}\}$.

## A.3 Example 3.

Example 3, illustrated in Figure 13, is even more tricky: four circumferences share a m.i.p. and a same tangent line. However, two of them lie on one side of the line and two of them on the other.

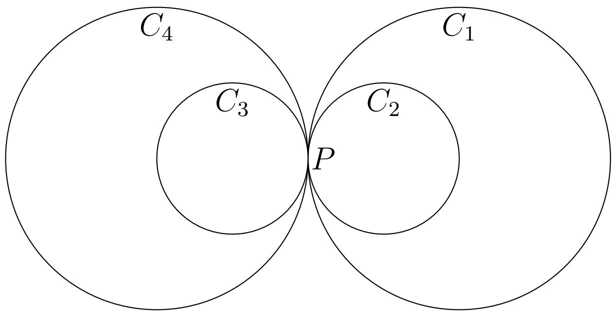
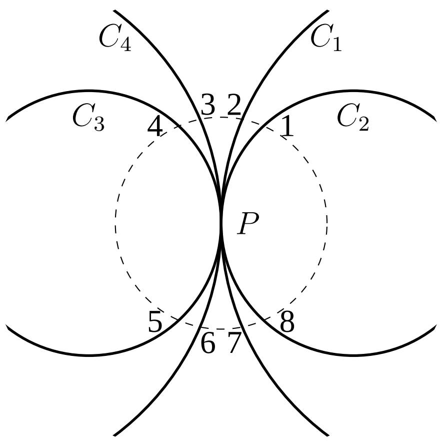


Figure 13: Example 3: four circumferences sharing a single m.i.p. and the same tangent line.



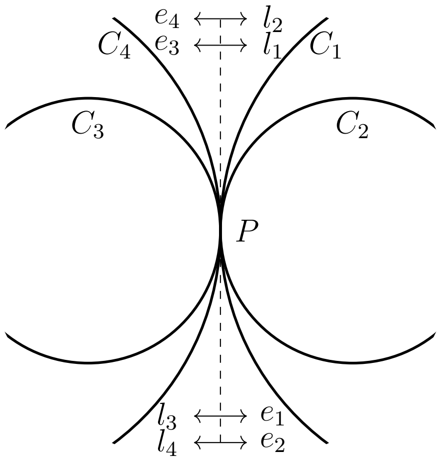

$e_4 \longleftrightarrow l_2$
$e_3 \longleftrightarrow l_1$
$C_4$
$C_1$
$C_3$
$C_2$
$P$
$l_3 \longleftrightarrow e_1$
$l_4 \longleftrightarrow e_2$

Figure 14: Left: the neighborhood of the m.i.p.. Right: the directions of the tangent lines.

Figure 14 shows the correspondence between the intersections along the frontier of the neighborhood of $P$ and the sorted angle values. To correctly sort the angles in this example it is necessary to apply both tie-break criteria. According to tie-break criterion 1, $l_1$ and $l_2$ must precede $e_3$ and $e_4$; for the same reason $l_3$ and $l_4$ must precede $e_1$ and $e_2$. According to tie-break criterion 2, $l_2$ must precede $l_1$, $e_4$ must precede $e_3$, $l_3$ must precede $l_4$ and $e_1$ must precede $e_2$. Therefore the vector $V_P = \{-2, -1, +4, +3, -3, -4, +1, +2\}$ is obtained. When Algorithm 1 is executed, after the first scan of $V_P$ we get $Q = \{1, 2\}$; during the second scan the following (redundant) set of regions is found: $\{\{1\}, \emptyset, \{4\}, \{3, 4\}, \{4\}, \emptyset, \{1\}, \{1, 2\}\}$.

## Appendix B    The three examples rivisited

Hereafter we report the same three examples shown in the Appendix A to show how they are solved by the new algorithm. In Figures 15, 16 and 17 forward edges are indicated by big arrows and backtrack edges by small arrows.

### B.1    Example 1.

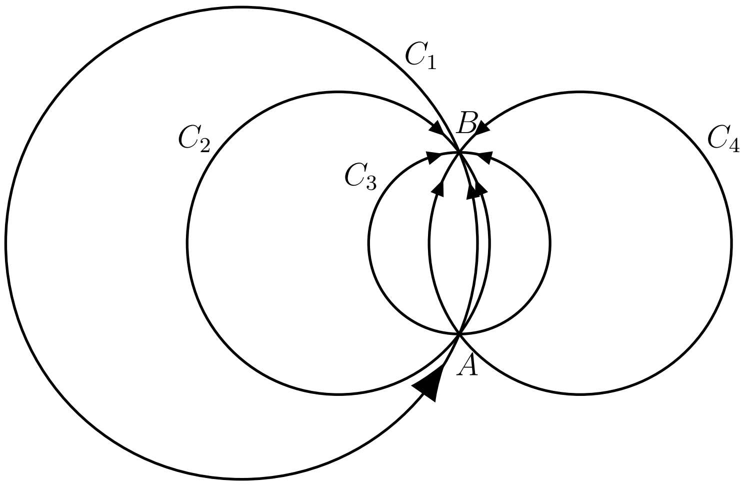


Figure 15: Example 1 [4]: four circumferences sharing two m.i.p..

The steps of the algorithm produce the output reported in Tables 1 to 6.

| $i$ | $f$ | $\Omega$ |
|---|---|---|
| 1 | true | $\emptyset$ |
| 2 | true | $\emptyset$ |
| 3 | true | $\emptyset$ |
| 4 | true | $\emptyset$ |

Table 1: Output of Intersections, Algorithm 2.

| $\Lambda$ | Sorted $\Lambda$ |
|---|---|
| $(1, 2, x_A, y_A)$ | $(1, 2, x_A, y_A)$ |
| $(2, 1, x_B, y_B)$ | $(1, 3, x_A, y_A)$ |
| $(1, 3, x_A, y_A)$ | $(1, 4, x_A, y_A)$ |
| $(3, 1, x_B, y_B)$ | $(2, 3, x_A, y_A)$ |
| $(1, 4, x_A, y_A)$ | $(2, 4, x_A, y_A)$ |
| $(4, 1, x_B, y_B)$ | $(3, 4, x_A, y_A)$ |
| $(2, 3, x_A, y_A)$ | $(2, 1, x_B, y_B)$ |
| $(3, 2, x_B, y_B)$ | $(3, 1, x_B, y_B)$ |
| $(2, 4, x_A, y_A)$ | $(4, 1, x_B, y_B)$ |
| $(4, 2, x_B, y_B)$ | $(3, 2, x_B, y_B)$ |
| $(3, 4, x_A, y_A)$ | $(4, 2, x_B, y_B)$ |
| $(4, 3, x_B, y_B)$ | $(4, 3, x_B, y_B)$ |

Table 2: List of the intersection points before and after sorting.

| $v$ | $T$ |
|---|---|
| $A$ | $\{1, 2, 3, 4\}$ |
| $B$ | $\{1, 2, 3, 4\}$ |

Table 3: Output of FindVertices, Algorithm 3.

| $i$ | $W$ | Sorted $W$ |
|---|---|---|
| 1 | $\{A, B\}$ | $\{B, A\}$ |
| 2 | $\{A, B\}$ | $\{B, A\}$ |
| 3 | $\{A, B\}$ | $\{B, A\}$ |
| 4 | $\{A, B\}$ | $\{B, A\}$ |

Table 4: Output of EnumerateVertices, Algorithm 4, and SortVertices, Algorithm 5.

| $v$ | $H$ |
|---|---|
| $A$ | $\{(1, 0, B), (1, 1, B), (2, 0, B), (2, 1, B), (3, 0, B), (3, 1, B), (4, 0, B), (4, 1, B)\}$ |
| $B$ | $\{(1, 1, A), (1, 0, A), (2, 1, A), (2, 0, A), (3, 1, A), (3, 0, A), (4, 1, A), (4, 0, A)\}$ |

Table 5: Output of BuildStar, Algorithm 6.

| $v$ | $H$ |
|---|---|
| $A$ | $\{(3,1,B),(2,1,B),(1,1,B),(4,0,B),(3,0,B),(2,0,B),(1,0,B),(4,1,B)\}$ |
| $B$ | $\{(3,0,A),(4,0,A),(1,1,A),(2,1,A),(3,1,A),(4,1,A),(1,0,A),(2,0,A)\}$ |

Table 6: Sorted vertex stars.

Table 7 reports the values of the main variables and data-structures during the execution of Step 4. Columns 1 indicates the open vertices. Closed vertices are not indicated explicitly for space reasons; however, a vertex is closed when it disappears from the set of open vertices. Columns 2-5 indicate the edge that is traversed: $k$ is the tail, $i$ is the circumference, $\gamma$ is the direction, $h$ is the head. When $h = k$, the edge is a self-loop. Column 6 indicates whether the edge is a forward edge (fw), a backward edge traversed for the first time (bt1) or a backward edge traversed for the second time (bt2). The values of $\mu$ are reported for all vertices: when $\mu(h) = 0$ then the edge is a forward edge; in the other cases it is a backtrack edge. The last but one column indicates the current region $Q$, that is updated every time a backtrack edge is traversed, by flipping the circumference that the traversed edge belongs to. The last column indicates the region that is evaluated when the single-source optimal location procedure is executed. It coincides with $Q$, but the evaluation only occurs when backtrack edges are traversed for the first time.

| Open | $k$ | $i$ | $\gamma$ | $h$ | fw/bt | $\overline{\mu}$ | $\mu(A)$ | $\mu(B)$ | $Q$ | Evaluated |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | 0 | 0 | 0 | | |
| | | 4 | 1 | $B$ | | 1 | | 1 | | |
| $B$ | $B$ | 1 | 1 | $A$ | fw | 2 | 2 | | | |
| $A,B$ | $A$ | 4 | 1 | $B$ | bt1 | | | | $\{4\}$ | $\{4\}$ |
| $A,B$ | $A$ | 3 | 1 | $B$ | bt1 | | | | $\{3,4\}$ | $\{3,4\}$ |
| $A,B$ | $A$ | 2 | 1 | $B$ | bt1 | | | | $\{2,3,4\}$ | $\{2,3,4\}$ |
| $A,B$ | $A$ | 1 | 1 | $B$ | bt1 | | | | $\{1,2,3,4\}$ | $\{1,2,3,4\}$ |
| $A,B$ | $A$ | 4 | 0 | $B$ | bt1 | | | | $\{1,2,3\}$ | $\{1,2,3\}$ |
| $A,B$ | $A$ | 3 | 0 | $B$ | bt1 | | | | $\{1,2\}$ | $\{1,2\}$ |
| $A,B$ | $A$ | 2 | 0 | $B$ | bt1 | | | | $\{1\}$ | $\{1\}$ |
| $B$ | $B$ | 2 | 1 | $A$ | bt2 | | | | $\{1,2\}$ | |
| $B$ | $B$ | 3 | 1 | $A$ | bt2 | | | | $\{1,2,3\}$ | |
| $B$ | $B$ | 4 | 1 | $A$ | bt2 | | | | $\{1,2,3,4\}$ | |
| $B$ | $B$ | 1 | 0 | $A$ | bt2 | | | | $\{2,3,4\}$ | |
| $B$ | $B$ | 2 | 0 | $A$ | bt2 | | | | $\{3,4\}$ | |
| $B$ | $B$ | 3 | 0 | $A$ | bt2 | | | | $\{4\}$ | |

Table 7: The iterations of ScanStar, Algorithm 9. The effect of Components, Algorithm 8, is indicated above the horizontal line.

## B.2 Example 2.

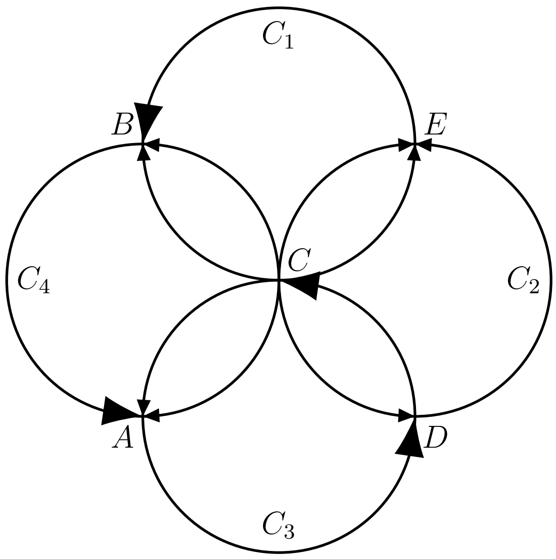


Figure 16: Example 2 [4]: four circumferences sharing a single m.i.p..

The steps of the algorithm produce the output reported in Tables 8 to 13.

| $i$ | $f$ | $\Omega$ |
|---|---|---|
| 1 | true | $\emptyset$ |
| 2 | true | $\emptyset$ |
| 3 | true | $\emptyset$ |
| 4 | true | $\emptyset$ |

Table 8: Output of Intersections, Algorithm 2.

| $\Lambda$ | Sorted $\Lambda$ |
|---|---|
| $(1,2,x_C,y_C)$ | $(4,3,x_A,y_A)$ |
| $(2,1,x_E,y_E)$ | $(1,4,x_B,y_B)$ |
| $(1,3,x_C,y_C)$ | $(1,2,x_C,y_C)$ |
| $(3,1,x_C,y_C)$ | $(1,3,x_C,y_C)$ |
| $(1,4,x_B,y_B)$ | $(3,1,x_C,y_C)$ |
| $(4,1,x_C,y_C)$ | $(4,1,x_C,y_C)$ |
| $(2,3,x_C,y_C)$ | $(2,3,x_C,y_C)$ |
| $(3,2,x_D,y_D)$ | $(2,4,x_C,y_C)$ |
| $(2,4,x_C,y_C)$ | $(4,2,x_C,y_C)$ |
| $(4,2,x_C,y_C)$ | $(3,4,x_C,y_C)$ |
| $(3,4,x_C,y_C)$ | $(3,2,x_D,y_D)$ |
| $(4,3,x_A,y_A)$ | $(2,1,x_E,y_E)$ |

Table 9: List of the intersection points before and after sorting.

| $v$ | $T$ |
|---|---|
| $A$ | $\{3,4\}$ |
| $B$ | $\{1,4\}$ |
| $C$ | $\{1,2,3,4\}$ |
| $D$ | $\{2,3\}$ |
| $E$ | $\{1,2\}$ |

Table 10: Output of FindVertices, Algorithm 3.

| $i$ | $W$ | Sorted $W$ |
|---|---|---|
| 1 | $\{B,C,E\}$ | $\{E,B,C\}$ |
| 2 | $\{C,D,E\}$ | $\{E,C,D\}$ |
| 3 | $\{A,C,D\}$ | $\{D,C,A\}$ |
| 4 | $\{A,B,C\}$ | $\{C,B,A\}$ |

Table 11: Output of EnumerateVertices, Algorithm 4, and SortVertices, Algorithm 5.

| $v$ | $H$ |
|---|---|
| $A$ | $\{(3,1,D),(3,0,C),(4,1,C),(4,0,B)\}$ |
| $B$ | $\{(1,0,E),(1,1,C),(4,1,A),(4,0,C)\}$ |
| $C$ | $\{(1,0,B),(1,1,E),(2,1,D),(2,0,E),(3,1,A),(3,0,D),(4,1,B),(4,0,A)\}$ |
| $D$ | $\{(2,1,E),(2,0,C),(3,1,C),(3,0,A)\}$ |
| $E$ | $\{(1,1,B),(1,0,C),(2,1,C),(2,0,D)\}$ |

Table 12: Output of BuildStar, Algorithm 6.

| $v$ | Sorted $H$ |
|---|---|
| $A$ | $\{(4,1,C),(3,0,C),(4,0,B),(3,1,D)\}$ |
| $B$ | $\{(4,0,C),(1,0,E),(4,1,A),(1,1,C)\}$ |
| $C$ | $\{(3,0,D),(1,1,E),(2,0,E),(4,1,B),(1,0,B),(3,1,A),(4,0,A),(2,1,D)\}$ |
| $D$ | $\{(2,1,E),(3,1,C),(2,0,C),(3,0,A)\}$ |
| $E$ | $\{(2,0,D),(1,1,B),(2,1,C),(1,0,C)\}$ |

Table 13: Sorted vertex stars.

Table 14 reports the values of the main variables and data-structures during the execution of Step 4. The meaning is the same as for Example 1.

| Open | $k$ | $i$ | $\gamma$ | $h$ | fw/bt | $\overline{\mu}$ | $\mu(A)$ | $\mu(B)$ | $\mu(C)$ | $\mu(D)$ | $\mu(E)$ | $Q$ | Evaluated |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|  |  |  |  |  |  | $0$ | $0$ | $0$ | $0$ | $0$ | $0$ |  |  |
|  |  | $2$ | $1$ | $E$ |  | $1$ |  |  |  |  | $1$ |  |  |
| $E$ | $E$ | $1$ | $1$ | $B$ | fw | $2$ |  | $2$ |  |  |  |  |  |
| $B, E$ | $B$ | $4$ | $1$ | $A$ | fw | $3$ | $3$ |  |  |  |  |  |  |
| $A, B, E$ | $A$ | $3$ | $1$ | $D$ | fw | $4$ |  |  |  | $4$ |  |  |  |
| $A, B, D, E$ | $D$ | $2$ | $1$ | $E$ | bt1 |  |  |  |  |  |  | $\{2\}$ | $\{2\}$ |
| $A, B, D, E$ | $D$ | $3$ | $1$ | $C$ | fw | $5$ |  |  | $5$ |  |  | $\{2\}$ |  |
| $A, B, C, D, E$ | $C$ | $1$ | $1$ | $E$ | bt1 |  |  |  |  |  |  | $\{1, 2\}$ | $\{1, 2\}$ |
| $A, B, C, D, E$ | $C$ | $2$ | $0$ | $E$ | bt1 |  |  |  |  |  |  | $\{1\}$ | $\{1\}$ |
| $A, B, C, D, E$ | $C$ | $4$ | $1$ | $B$ | bt1 |  |  |  |  |  |  | $\{1, 4\}$ | $\{1, 4\}$ |
| $A, B, C, D, E$ | $C$ | $1$ | $0$ | $B$ | bt1 |  |  |  |  |  |  | $\{4\}$ | $\{4\}$ |
| $A, B, C, D, E$ | $C$ | $3$ | $1$ | $A$ | bt1 |  |  |  |  |  |  | $\{3, 4\}$ | $\{3, 4\}$ |
| $A, B, C, D, E$ | $C$ | $4$ | $0$ | $A$ | bt1 |  |  |  |  |  |  | $\{3\}$ | $\{3\}$ |
| $A, B, C, D, E$ | $C$ | $2$ | $1$ | $D$ | bt1 |  |  |  |  |  |  | $\{2, 3\}$ | $\{2, 3\}$ |
| $A, B, D, E$ | $D$ | $2$ | $0$ | $C$ | bt2 |  |  |  |  |  |  | $\{3\}$ |  |
| $A, B, E$ | $A$ | $4$ | $1$ | $C$ | bt2 |  |  |  |  |  |  | $\{3, 4\}$ |  |
| $A, B, E$ | $A$ | $3$ | $0$ | $C$ | bt2 |  |  |  |  |  |  | $\{4\}$ |  |
| $B, E$ | $B$ | $1$ | $1$ | $C$ | bt2 |  |  |  |  |  |  | $\{1, 4\}$ |  |
| $B, E$ | $B$ | $4$ | $0$ | $C$ | bt2 |  |  |  |  |  |  | $\{1\}$ |  |
| $E$ | $E$ | $2$ | $0$ | $C$ | bt2 |  |  |  |  |  |  | $\{1, 2\}$ |  |
| $E$ | $E$ | $1$ | $0$ | $C$ | bt2 |  |  |  |  |  |  | $\{2\}$ |  |

Table 14: The iterations of ScanStar, Algorithm 9. The effect of Components, Algorithm 8, is indicated above the horizontal line.

## B.3 Example 3.

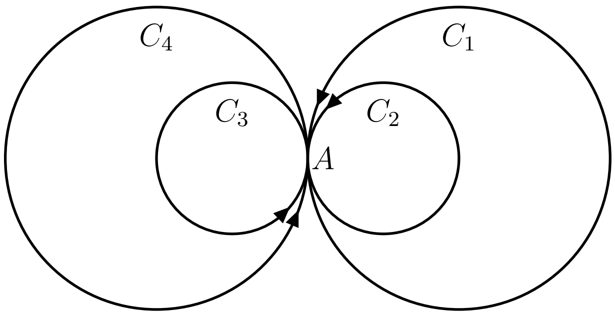


Figure 17: Example 3: four circumferences sharing a single m.i.p. and the same tangent line. In this very special case, the multi-graph includes only one vertex and no forward edge.

The steps of the algorithm produce the output reported in Tables 15 to 20.

| $i$ | $f$ | $\Omega$ |
|---|---|---|
| 1 | true | $\emptyset$ |
| 2 | true | $\emptyset$ |
| 3 | true | $\emptyset$ |
| 4 | true | $\emptyset$ |

Table 15: Output of Intersections, Algorithm 2.

| (sorted) $\Lambda$ |
|---|
| $(1, 2, x_A, y_A)$ |
| $(2, 1, x_A, y_A)$ |
| $(1, 3, x_A, y_A)$ |
| $(3, 1, x_A, y_A)$ |
| $(1, 4, x_A, y_A)$ |
| $(4, 1, x_A, y_A)$ |
| $(2, 3, x_A, y_A)$ |
| $(3, 2, x_A, y_A)$ |
| $(2, 4, x_A, y_A)$ |
| $(4, 2, x_A, y_A)$ |
| $(3, 4, x_A, y_A)$ |
| $(4, 3, x_A, y_A)$ |

Table 16: List of the intersection points.

| $v$ | $T$ |
|---|---|
| $A$ | $\{1,2,3,4\}$ |

Table 17: Output of FindVertices, Algorithm 3.

| $i$ | (Sorted) $W$ |
|---|---|
| 1 | $\{A\}$ |
| 2 | $\{A\}$ |
| 3 | $\{A\}$ |
| 4 | $\{A\}$ |

Table 18: Output of EnumerateVertices, Algorithm 4, and SortVertices, Algorithm 5.

| $v$ | $H$ |
|---|---|
| $A$ | $\{(1,1,A),(1,0,A),(2,1,A),(2,0,A),(3,1,A),(3,0,A),(4,1,A),(4,0,A)\}$ |

Table 19: Output of BuildStar, Algorithm 6.

| $v$ | Sorted $H$ |
|---|---|
| $A$ | $\{(2,0,A),(1,0,A),(4,1,A),(3,1,A),(3,0,A),(4,0,A),(1,1,A),(2,1,A)\}$ |

Table 20: Sorted vertex stars.

Table 21 below reports the values of the main variables and data-structures during the execution of Step 4. The meaning is the same as for Examples 1 and 2.

| Open | $k$ | $i$ | $\gamma$ | $h$ | fw/bt | $\overline{\mu}$ | $\mu(A)$ | $Q$ | Evaluated |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | $0$ | $0$ | | |
| | | $1$ | $1$ | $A$ | | $1$ | $1$ | | |
| $A$ | $A$ | $4$ | $1$ | $A$ | bt1 | | | $\{4\}$ | $\{4\}$ |
| $A$ | $A$ | $3$ | $1$ | $A$ | bt1 | | | $\{3,4\}$ | $\{3,4\}$ |
| $A$ | $A$ | $3$ | $0$ | $A$ | bt2 | | | $\{4\}$ | |
| $A$ | $A$ | $4$ | $0$ | $A$ | bt2 | | | $\{\}$ | |
| $A$ | $A$ | $1$ | $1$ | $A$ | bt1 | | | $\{1\}$ | $\{1\}$ |
| $A$ | $A$ | $2$ | $1$ | $A$ | bt1 | | | $\{1,2\}$ | $\{1,2\}$ |
| $A$ | $A$ | $2$ | $0$ | $A$ | bt2 | | | $\{1\}$ | |

Table 21: The iterations of ScanStar, Algorithm 9. The effect of Components, Algorithm 8, is indicated above the horizontal line.