Sweepline algorithm

Liam Volckerick
Bernd Weckx

August 18, 2020

Toepassingen van meetkunde in de informatica (B-KUL-G0Q37C)
Bestemming: N. Scheerlinck & D. Roose

1 Introduction

In this short paper we will discuss and analyse three different algorithms for calculating the intersections between any amount of circles. The language used to implement these algorithms was C++. The course for which this short paper was made for: Toepassingen van de meetkunde in de informatica.

B. Weckx & L. Volckerick

Contents

2 Algorithms

2.1 N squared algorithm

We were asked to implement an algorithm that calculates intersection points between circles on three different complexities. The first algorithm has a time complexity of $\mathcal{O}(N^2)$ and will be written in high level pseudo code in subsection 2.1.2.

2.1.1 Data structures and help functions

In order to fully implement this algorithm we will need to create certain data structures which allow us to write efficient code.

1. We created a structure called 'Point'. This data structure keeps track of the x-coordinate and the y-coordinate of a certain 'point'. These points will always be owned by a certain circle. Thus meaning a point can be any pixel within that particular circle's area. We will keep track of that circle's ID for further references. There will occur some slight changes to this data structure in the implementation in algorithm 10. See 2.3.1 for the changes.

2. When the input file is read by our program, it will be handled by our program such that each input circle will become an object of the form 'circle'. These data structures keep track of the center and its radius. We will also name each circle with a unique ID.

2.1.2 High-level pseudo code

Algorithm 1 is a function required to measure the distance between two points. This function will be used in all three algorithms.

Algorithm 1 distanceCalculator

```
1: function DISTANCECALCULATE(pointp1, pointp2)
2:     x ← p1.x − p2.x
3:     y ← p1.y − p2.y
4:     distance ← x^2 + y^2
5:     return distance
6: end function
```

Algorithm 2 will calculate intersections between circles. This function will be adapted accordingly such that it can be used for all three algorithms. When first reading this algorithm you'll notice the function $'intersectionpoints'$. This function, which we will not explicitly explain further in to detail, will calculate with the help of parametric equations all intersection points between two circles. This can be either two intersections or one. See [1] for further details as to why this works.

Algorithm 2 calcIntersection

```
1: function CALCINTERSECTION(c1, c2, I)        ▷ I is a set containing all intersection points. c1 and c2 are two circles which will be compared.
2:     P ← intersectionpoints between c1 and c2
3:     for all point ∈ P do
4:         if point ∉ I then                   ▷ This condition is not required in our implementation. We use a set with unique elements thus no element can already be in that given goal set.
5:             I.Add(point)
6:         end if
7:     end for
8: end function
```

Algorithm 3 illustrates the main principle of the first algorithm with a complexity of $\mathcal{O}(N^2)$.

Algorithm 3 Non sweep line algorithm: $\mathcal{O}(N^2)$

```
1: function ALGO1(C, I)        ▷ C is list of all circles, I is a set of intersection points between circles
2:     counter ← 0
3:     while counter ≠ length(C) do        ▷ We iterate over all the circles
4:         c ← C.at(counter)
5:         for all circle ∈ C do
6:             if circle ≠ c then
7:                 CALCULATEINTERSECTION(c, circle, I)
8:             end if
9:         end for
10:    end while
11:    return I                            ▷ Return all found intersection points
12: end function
```

2.1.3 Complexity analysis of algorithm 3

Assuming all the algorithms are correct and the data structures work as expected, we can determine the complexity with ease for Algorithm 10.

- calcIntersection (2.1.2): $\mathcal{O}(1)$ This can be explained due to simple equations and calculations done by the function 'intersectionpoints'. There is no statement that has any relation with N.
- distanceCalculator (2.1.2): $\mathcal{O}(1)$
- Non sweep line algorithm (2.1.2): $\mathcal{O}(N^2)$ As seen in the pseudo code of 2.1.2 we notice that each circle from C will be compared to all other circles in C. This iteration will have a tilde complexity of $\sim N*(N-1)$ resulting in a $\mathcal{O}(N^2)$.

2.2 N squared worst case algorithm

The second circle intersection algorithm will make use of a sweep line for calculating the intersection points. This algorithm will make use of algorithm 4(see 2.1.2 for pseudo code).

Algorithm 4 Sweep line Algorithm with $\mathcal{O}(N^2)$ worst case

```
1: function ALGO2(EP, I)▷ EP is a priority queue of event points, either left or right. I is a set of intersection points between circles
2:     AC ← ∅                                   ▷ List of currently active circles
3:     for all eventpoint ∈ EP do
4:         if eventpoint = left point of a circle then ▷ event point is left. This circle is now active.
5:             c ← eventpoint.ownerCircle
6:             for all circle ∈ AC do
7:                 CALCULATEINTERSECTION(c, circle, I)
8:             end for
9:             AC.Add(c)
10:        else    ▷ event point would be right. This circle is no longer active.
11:            AC.Remove(eventpoint.ownerCircle)
12:        end if
13:    end for
14:    return I                                 ▷ Return all found intersection points
15: end function
```

2.2.1 Complexity analysis of algorithm 4

We already stated in the title of the algorithm as in the section's title that this sweep line algorithm has a worst case time complexity of $\mathcal{O}(N^2)$. Worst case will occur whenever all circles are active at the same time, thus resulting in a comparison of N circles to find the intersecting points. We can assume that in a $Z * Z - square$ not all circles will be stacked upon each other. Resulting in a complexity lower than $\mathcal{O}(N^2)$ as the average case. This result will also be shown in section 3.

2.2.2 Edge cases of algorithm 4

In the first intersection algorithm, there were no edge cases/exceptions that could not be processed. In this algorithm there is one edge case that cannot be handled by our program. Whenever a circle ends on a x-coordinate, where another circle starts, the program may not recognize this intersection due to it first removing the ending circle from the Priority queue as seen in 4. This edge case can "easily" be solved by doing the following; When we process/build the event point queue EP for the first time, it should detect multiple event points on the same x-coordinate. We'd want to sort these event points in such a way

that all event points that activate new circles, will appear before *any* event point occurs that may deactivate an existing one. Keep in mind that our sweep line moves from Left to Right. This way, we would first process and find any new intersection points between already active circles and newly introduced ones. But this simple fix, which we have not implemented due to time saving and it brings a lot of inconveniences, will resolve this edge case.

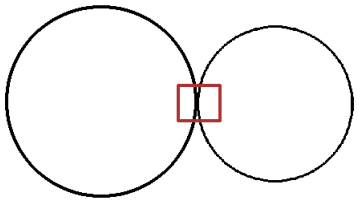

Figure 1: Edge case where there is an intersection point right at the right edge point of a circle.

2.3 (N+S) log N Sweep line algorithm

2.3.1 Data structures and help functions

In order to perform our implementation of algorithm 10 we need several new data structures and help functions to process and properly maintain our sweep line status.

1. A way to keep track of the x-coordinate of the sweep line. We use a static variable in this implementation since we know only 1 sweep line would be active at a time. A non-static variant is easy to implement, just pass the x-coordinate of the currently active event point in algorithm 10 (this is in essence, the sweep line location).

2. A small extension of our EP (event point) PriorityQueue from algorithm 4. In this particular implementation it's important this data structure also has the properties of a set, such that no 2 identical event points can populate the queue.

3. calculateIntersection has been modified slightly and renamed to calcIntersection for algorithm 10. The difference between the two is that calcIntersection calculates intersection points between half circles and adds those points to the EP (event point) queue instead of the intersection points set.

- CalcIntersection is only interested in event points (in this case intersection points) that occur to the *right* of the sweep line, since the algorithm assumes all event points to the left of the sweep line have already been processed.
- It's for this reason we needed a variable that keeps track of the sweep line's x-coordinate.
- Although intersection points are calculated in calcIntersection they will be added to the intersection points set I at a later point in time (when we process the intersection point event itself).

Note : We assume that (just like calculateIntersection) calcIntersection ends without modifying any data structure if either of the 2 given half circles are invalid. For example if the top element of the status line gets compared to it's upper neighbour (which shouldn't exist) the function calcIntersection will terminate immediately without having done any work.

4. We will need a data structure that keeps track of the sweep line status (of all activated half circles) and maintains the correct order of its elements throughout the entire runtime of the program. The optimal choice for this sweep status would be a self balancing tree with at least the following possible operations:

   (a) Adding (and removing) half circles to the tree, essentially activating (and deactivating) half circles similar to the process in algorithm 4.

   (b) Obtaining both upper and lower (resp. *successor* and *predecessor*) elements from the status tree. As defined by the functions $getAbove$ and $getBelow$.

   (c) Checking whether a certain element is above (or below) another element, meaning, is a given $halfCircle$ the *successor* (or *predecessor*) of another given $halfCircle$. As defined by the functions $isAbove$ and $isBelow$, so for example $isAbove(T, hc1, hc2)$ checks status tree T whether $hc1$ is placed above $hc2$, meaning that this functions returns $TRUE$ if $hc1$ is $hc2$'s successor.

   (d) We will need a clear ordering structure for our status tree. We will order our half circles based on the x-coordinate of the sweep line and given it's *center* position and *radius* with the following formula (based on the equation of a circle):

   $$y = \pm\sqrt{radius^2 - (sweeplineX - center.x)^2} + center.y$$

   We get 2 y results, this is normal, hence a straight line downwards through a circle would intersect 2 points, these are the y-coordinates of those 2 points. Since we work with half circles, 1 of those points is guaranteed to be on the upper circle and one of them on the lower one.

(e) Lastly we will need a $swap(T, hc1, hc2)$ function that swaps 2 half circles, $hc1$ and $hc2$, inside the status tree T. This is required such that if $hc1$'s *predecessor* is $hc2$ before the swap, $hc1$'s *successor* becomes $hc2$ after the swap.

$Note$: this is actually the trickiest operation of the status tree T since you aren't supposed to switch items in a self balancing tree, our solution works as follows: remove both half circles from the tree, take a "very small" step to the right (move the sweep line just a tiny bit to the right so that it's x-coordinate is a little bit larger) and add the half circles back into the tree, due to the formula of y above the new status tree should now be in the correct order.

$Note$: Our implementation uses a vector to represent the sweep line status, combined with a binary search to find the correct position of newly inserted elements. A slightly slower approach than the status tree described above, but easier to implement the above mentioned operations and that, while debugging, the data structure visually represents the status order of the elements.

5. A last small modification of previously existing data structures would be of our *Point* data, for this algorithm every event point should know "from which 2 intersecting half circles" this event point originates:

   - Left point, just like in algorithm 4, activates new elements in our status line. In this particular case 2 half circles for our status tree T. It should hold a reference to the 2 new half circles that are to be introduced, both a single upper and lower circle.
   - Right point, analogous to the explanation above, the right point should have a reference to the 2 half circles that are to be removed, both a single upper and lower circle.
   - Intersection point, an intersection event should keep track of the spawn of this event point (meaning on which of the 2 half circles), so that it can swap those 2 elements in the status tree T. It should also keep track whether the intersection point was tangent or not, since tangent points are an exceptional case where the 2 half circles $shouldn't$ be swapped.

2.3.2 Complexity analysis of algorithm 10

Assuming that we have correctly chosen and implemented all given data structures and help functions we can argue that the following $\mathcal{O}$ complexities hold true (without proof, many of these are due to using a self balancing tree):

- calcIntersection: $\mathcal{O}(1)$
- isAbove: $\mathcal{O}(\log n)$

Algorithm 5 calcIntersection

```
1: function CALCINTERSECTION(hc1, hc2, eventPoints)
2:     P ← intersectionpoints between hc1 and hc2
3:     for all point ∈ P do
4:         if point.x >= sweeplineX then ▷ We assume sweeplineX is the x-coordinate of the sweep line, we handle the coordinate as a static variable in this case.
5:             eventPoints.Add(point)
6:         end if
7:     end for
8: end function
```

Algorithm 6 isAbove

```
1: function ISABOVE(StatusTree, halfCircle1, halfCircle2)
2:     if halfCircle1 is placed above halfCircle2 in the StatusTree then
3:         return TRUE
4:     else
5:         return FALSE
6:     end if
7: end function
```

Algorithm 7 isBelow

```
1: function ISBELOW(StatusTree, halfCircle1, halfCircle2)
2:     if halfCircle1 is placed below halfCircle2 in the StatusTree then
3:         return TRUE
4:     else
5:         return FALSE
6:     end if
7: end function
```

Algorithm 8 getAbove

```
1: function GETABOVE(StatusTree, halfCircle)
2:     return successor of halfCircle from the StatusTree ▷ Get halfCircle's upper neighbour.
3: end function
```

Algorithm 9 getBelow

```
1: function GETBELOW(StatusTree, halfCircle)
2:     return predecessor of halfCircle from the StatusTree ▷ Get halfCircle's lower neighbour.
3: end function
```

Algorithm 10 (N+S)log(N) sweep line algorithm

```
1: function ALGO3(EP, I) ▷ EP is a priority queue of event points, either left or right. I is a set of intersection points between circles
2:     T ← ∅    ▷ Self balancing tree (T) that represents the status line at any point in time.
3:     while EP not empty do
4:         eventpoint ← EP.pop()
5:         switch eventpoint do
6:             case is intersection and not is tangent    ▷ Swap both halves.
7:                 I.add(eventpoint)
8:                 hc1 ← eventpoint.halfCircle1
9:                 hc2 ← eventpoint.halfCircle2
10:                if isAbove(T, hc1, hc2) then
11:                    CALCINTERSECTION(hc2,getAbove(T, hc1),EP)
12:                    CALCINTERSECTION(getBelow(T, hc2),hc1,EP)
13:                else
14:                    CALCINTERSECTION(hc1,getAbove(T, hc2),EP)
15:                    CALCINTERSECTION(getBelow(T, hc1),hc2,EP)
16:                end if
17:                SWAP(T,hc1,hc2)
18:            end case
19:            case is left    ▷ Add both halves.
20:                hc1 ← eventpoint.halfCircle1
21:                hc2 ← eventpoint.halfCircle2
22:                addHalfCircle(T, hc1)
23:                addHalfCircle(T, hc2)
24:                CALCINTERSECTION(hc1,getAbove(T, hc1),EP)
25:                CALCINTERSECTION(hc2,getBelow(T, hc2),EP)
26:            end case
27:            case is right    ▷ Remove both halves.
28:                hc1 ← eventpoint.halfCircle1
29:                hc2 ← eventpoint.halfCircle2
30:                if isAbove(T, hc1, hc2) then
31:                    CALCINTERSECTION(getAbove(T, hc1),getBelow(T, hc2),EP)
32:                else
33:                    CALCINTERSECTION(getAbove(T, hc2),getBelow(T, hc1),EP)
34:                end if
35:                removeHalfCircle(T, hc1)
36:                removeHalfCircle(T, hc2)
37:            end case
38:        end switch
39:    end while
40:    return I
41: end function
```

- isBelow: $\mathcal{O}(\log n)$
- getAbove: $\mathcal{O}(\log n)$
- getBelow: $\mathcal{O}(\log n)$
- insert in status tree T: $\mathcal{O}(\log n)$
- deletion from status tree T: $\mathcal{O}(\log n)$
- swap in T: assumed to be both 2 insertions and 2 deletions $\mathcal{O}(\log n)$
- we assume that popping a new element from the event queue EP happens in $\mathcal{O}(1)$

  $Note$: all these complexities assume n being the amount of elements present in a certain status tree T.

We now follow algorithm 10 from start to end. Algorithm 10 generates data by processing event points from the event queue EP one by one from "left to right". At the start of algorithm 10 EP will be populated by N*2 event points (1 left and 1 right event point per circle), with N being the amount of circles to process. Assuming we find all intersection points, let the amount of intersection points be S, we know that the exact amount of event points processed by algorithm 10 will be equal to $(N * 2 + S)$, since every intersection point adds a new intersection event point to the event queue EP. There are 3 ways an event point can be processed by this algorithm:

1. A left event point causes 2 half circles to be added to the status tree (both with complexity $\mathcal{O}(\log n)$) and up to 2 getAbove and/or getBelow (also with $\mathcal{O}(\log n)$ complexity) and with a maximum of 2 calcIntersection calls ($\mathcal{O}(1)$ complexity). So one can argue that in worst case (maximum amount of getAbove/Below and calcIntersection calls) complexity will be of order $\mathcal{O}(2 * \log n + 2 * \log n + 2 * 1) = \mathcal{O}(\log n)$. There are precisely N left event points in the event queue EP so this particular operation by algorithm 10 is of order $\mathcal{O}(n \log n)$.

2. A right event point causes 2 half circles to be removed from the status tree. And possibly checks the existing *successor* and *predecessor* of those 2 half circles for intersections. So similar to left event points its total complexity can be argued to be that of order of $\mathcal{O}(n \log n)$.

3. An intersection event point causes 2 half circles to be swapped. We argued before that we do this by first removing the circles and then re-inserting them into the status tree T. We then check the *predecessor* and *successor* of those half circles in this new status tree and do up to 2 additional intersection checks. So it's complexity can be calculated and appears to be an order of $\mathcal{O}(\log n)$. But in this case, there are S amount of intersection point events, thus the total complexity from start to finish should be equal to $\mathcal{O}(S \log n)$.

Note : We do note that tangent intersection points may occur, these points are handled in $\mathcal{O}(1)$ time complexity, but we assume that the ratio of tangent points compared to non-tangent points is near 0, thus total complexity should be equal to $\mathcal{O}(S \log n)$.

Add these 3 cases together and we should calculate something in the order of

$$\mathcal{O}(N \log N) + \mathcal{O}(N \log N) + \mathcal{O}(S \log N) = \mathcal{O}((S + 2N) \log N)$$

But since $\mathcal{O}(2 * N \log N) = \mathcal{O}(N \log N)$ we can rewrite the former as:

$$\mathcal{O}((S + N) \log N)$$

With S being the amount of intersection points and N being the amount of circles.

2.3.3 Edge cases of algorithm 10

This algorithm can have trouble with the following 4 edge cases:

1. The same edge case(s) as mentioned in algorithm 4, for the same reasons.
2. As shown in Figure 2, where an intersection occurs right on the exact same x-coordinate as the right event point for one of the circles. This algorithm has undefined behaviour in this algorithm, in the sense that one of the circles may be made inactive and thus deleted from the active status tree T, the base algorithm crashes trying to "swap" a non existent circle. We have "solved" this particular problem by checking whether both parent circles of an intersection point are active, if they are continue the algorithm as usual, in all other cases delete the current event point from the queue and skip the current algorithm iteration. This is possible since it's "safe" to assume that any swap made right before a circle deletion by those particular set of circles has no effect on the status tree.
3. Other problems may arise in the case of Figure 3, in this particular case multiple circles are added at the exact same x-and y-coordinate. Due to the way we order circles (adding half circles one by one in the status tree T) incorrect ordering may happen due to all half circles having the same y-coordinate on insertion into the status tree.
4. Lastly we have another particularly tricky case when an intersection point intersects more than 2 circles at the same time. We won't go too much into detail on why this is a difficult one to deal with, but remember that in order to keep the status tree correct at any point in time we must switch 2 half circles in the tree T when they intersect with each other. In this case, where 3 circles intersect each other, a single swap won't guarantee a correct status tree, and we can't add multiple identical intersection points into the event queue (since it behaves similar to a set in algorithm 10). A possible (unimplemented) implementation is to somehow keep track of *all*

circles that go through the same intersection point and when processing the intersection event, to remove all those half circles from the status tree and (move the sweep line slightly to the right) re-add them to the status tree and check all *predecessors* and *successors* for all the added half circles whether they intersect or not.

Note : Our algorithm 10 is based on the so-called "Bentley–Ottmann algorithm" using a sweep line from left to right using event points, Bentley-Ottmann's algorithm also makes the assumption that no 2 event points can happen on the same x-coordinate[2]. So if we make the same assumptions, edge cases 1,2 and 3 (and technically 4) shouldn't occur.

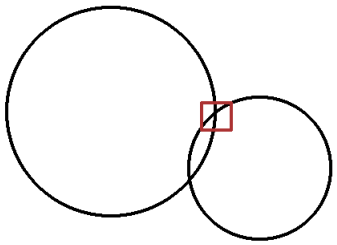

Figure 2: Edge case where there is an intersection point right at the right edge point of a circle.

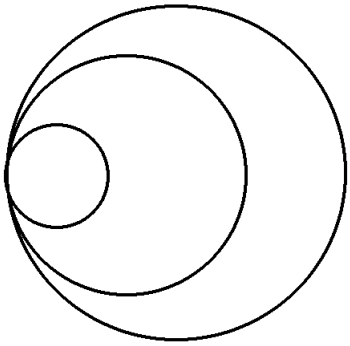

Figure 3: Edge case where several circles are introduced (become active) at the same x-coordinate. (Also refer to the "many intersections" edge case below)

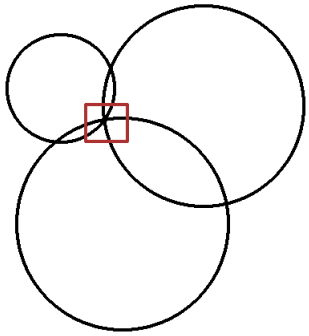

Figure 4: Edge case where several circles (more than exactly 2) intersect in a single particular point.

3 Experiments

This section will explain our results and verify noted time complexities from previous sections. We will start by showing a table with the measured timestamps for each algorithm for a given N. The shown results are measured in milliseconds.

Measured times for each algorithm									
N(circles)	100	500	1000	2000	4000	8000	16000	32000	64000
Algorithm 3	1	371	855	4.368	Too long	Too long	Too long	Too long	Too long
Algorithm 4	1	27	81	242	1050	4003	15873	68409	273636
Algorithm 10	3	68	156	531	1819	5873	18231	53256	174781

At a first glance, we notice that algorithm 4 is faster for a given N<32000. This can be explained due to algorithm 10 having a lot of complex data structures. All details in regard of why these times are higher than algorithm 4 were explained in section 2.3. We will always compare the correctness of each algorithm to algorithm 1. We do this because of the simple fact that algorithm 2.1.2 is complete even though it's not the fastest nor the most efficient algorithm. All algorithms had the correct output for each test. The table underneath shows the amount of intersection points found by all three algorithms.

Amount of intersections (S) found by all algorithms								
N(circles)	100	500	1000	2000	4000	8000	16000	32000
Algorithm 3	24	716	2696	10396	41532	167496	679753	2742017
Algorithm 4	24	716	2696	10396	41532	167496	679753	2742017
Algorithm 10	24	716	2696	10396	41532	167496	679753	2742017

Thus we can conclude that all times and other derived data will be correct in following sections.

3.1 Doubling experiment

To validate the complexities of each algorithm, we can implement the doubling-ratio test to our measured output times. If our algorithms are correct, the factors at which each algorithm increases with should be following:

$$Constant : 1$$

$$Logarithmic : 1$$

$$Linear : 2$$

$$Linearithmic :\sim 2$$

$$Quadratic : 4$$

$$Cubic : 8$$

We measured all factors for given dataset. The table underneath shows the results for each amount of circles. It had no point in starting with N=100 or 500 due to the very big changes in measured times. Thus will we start with N=1000, where the value written in that column is the result of the doubling ratio test with the data from N=500.

Doubling ratio test for each algorithm							
N(circles)	1000	2000	4000	8000	16000	32000	64000
Algorithm 3	2.3	4.31	4.37	EMPTY	EMPTY	EMPTY	EMPTY
Algorithm 4	3	2.99	4.34	3.81	3.97	4.31	4
Algorithm 10	2.29	3.4	3.43	3.22	3.1	2.92	3.28

We notice that the doubling ratio coefficient for algorithm 4 lays around a little less than 4 on the average, which is expected whenever N becomes very large. This can be explained due to the fact that, when N gets large in this experiment, we're nearing the worst case situation for algorithm 4 (with $\mathcal{O}(N^2)$ complexity). If we take a closer look at the table consisting of the amount of intersections found, we can see that the amount of intersections also behaves quadratic. This means, theoretically, that for every 2 extra circles added, an extra amount of 4 intersections will be found, causing the algorithm's complexity to effectively behave similar to that of order $\mathcal{O}(N^2)$. This is due to the fact that, for every intersection point that occurs its 2 intersecting circles must be active (in the active circle list AC), thus having a quadratic increasing amount of intersection points also causes a quadratic increase in the active circle list AC's length, causing algorithm 4 to behave quadratic. We could also consider following situations:

- When the amount of intersections grows linear.
- When the amount of intersections remains constant.
- Any other relation that can be found between the amount of circles and its intersections.

But we will not consider these situations in this paper. Algorithm 10 has a doubling rate coefficient of around 3. This result is expected due to the algorithm not being fully linearithmic, it's complexity is $\mathcal{O}(N + S)LogN$. Because of the simple fact that the amount of intersections increases quadratic in regards to the amount of circles. Thus will it behave itself inbetween linearithmic and quadratic complexities. See 2.3.2 for further information.

N(circles)/S(intersections)						
N(circles)	1000	2000	4000	8000	16000	32000
Algorithm 4	0.3709	0.1923	0.0963	0.0477	0.0235	0.0116

References

[1] P. Bourke. Spheres, equations and terminology, 1994. http://paulbourke.net/geometry/circlesphere/, last accessed on 01.04.1994.

[2] wikipedia. Bentley–ottmann algorithm, 2016. https://en.wikipedia.org/wiki/Bentley\%E2\%80\%93Ottmann_algorithm, last accessed on 22.05.2016.