Covering Points by a Disk

Group: Mohammad Ali, Curtis Gach, Arsal Mirza, Justin Ordonez

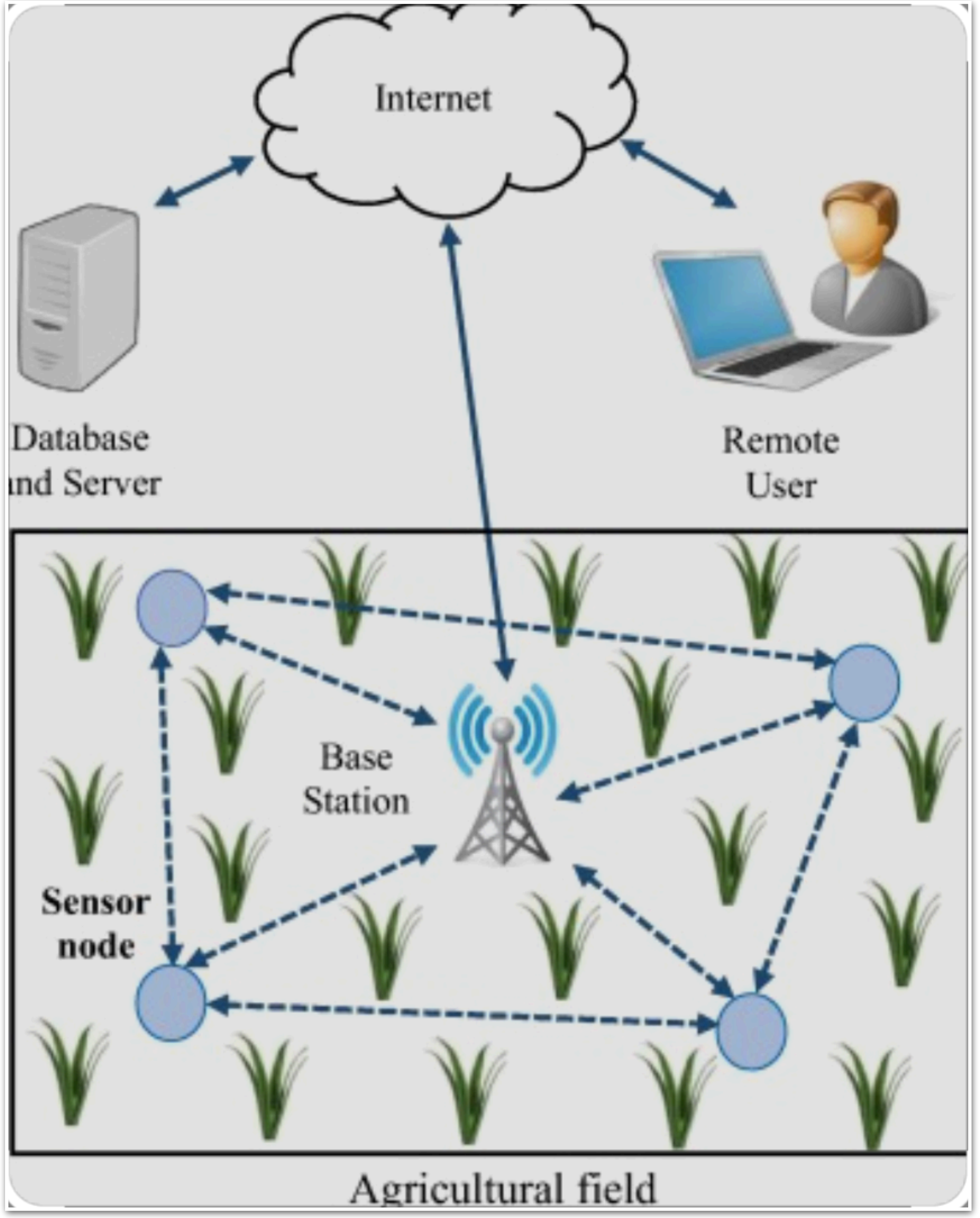
Advisor: Jingru Zhang

Outline

- Introduction
    - Executive Summary
    - Background
    - Objectives
- Technique Approach
    - The Two-Dimensional Version
    - The Line-Constrained Version
    - The One-Dimensional Version
- Deliverables
- Timeline
- Professional Awareness
- Conclusion

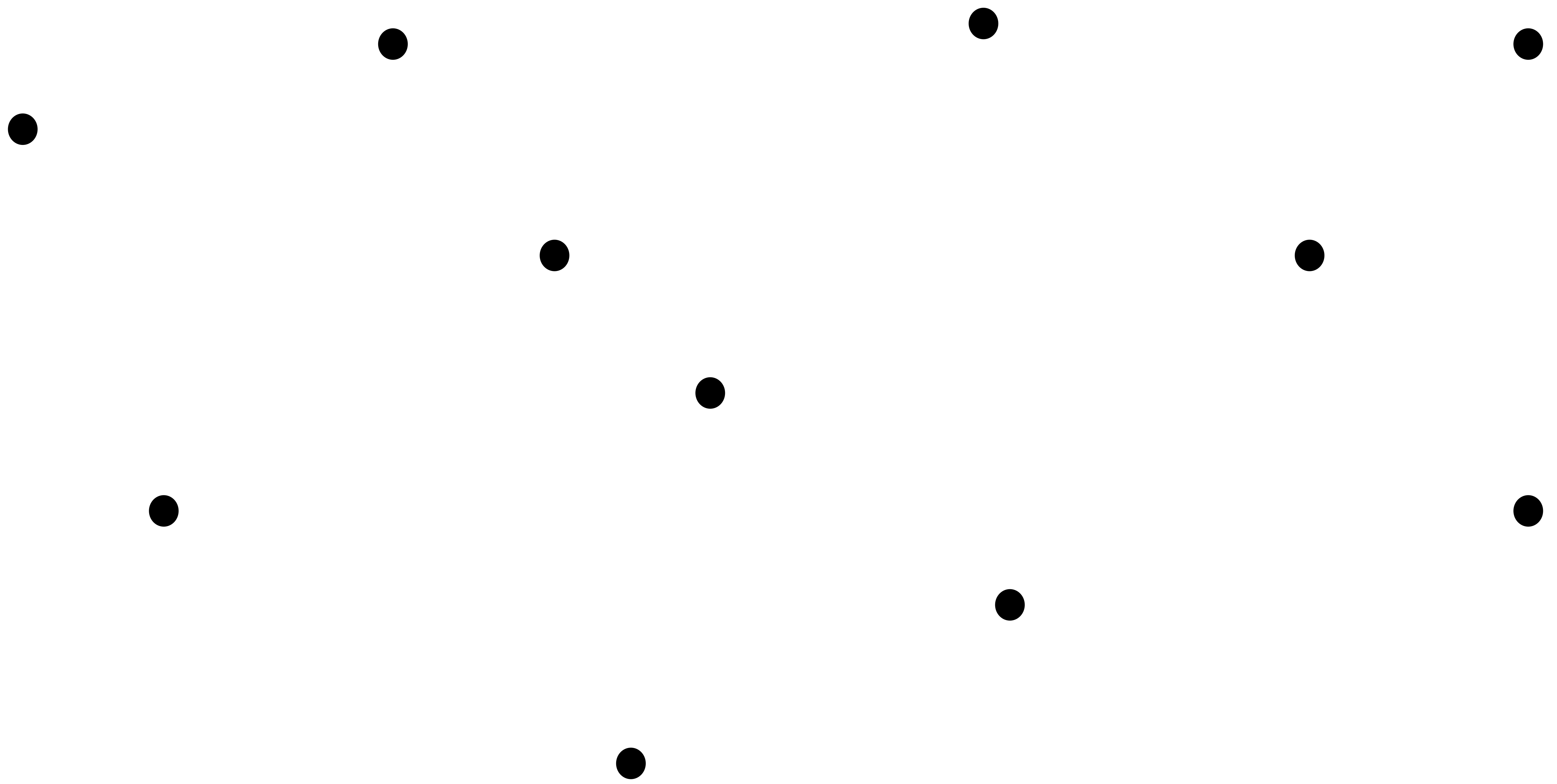
Executive Summary

- Goal of the project is to design three efficient algorithms to calculate the best position for placing a facility(e.g., sensor) to serve/ communicate with as many objects as possible.
- The algorithm will be implemented through the use of a GUI written in the programming language of our choice.



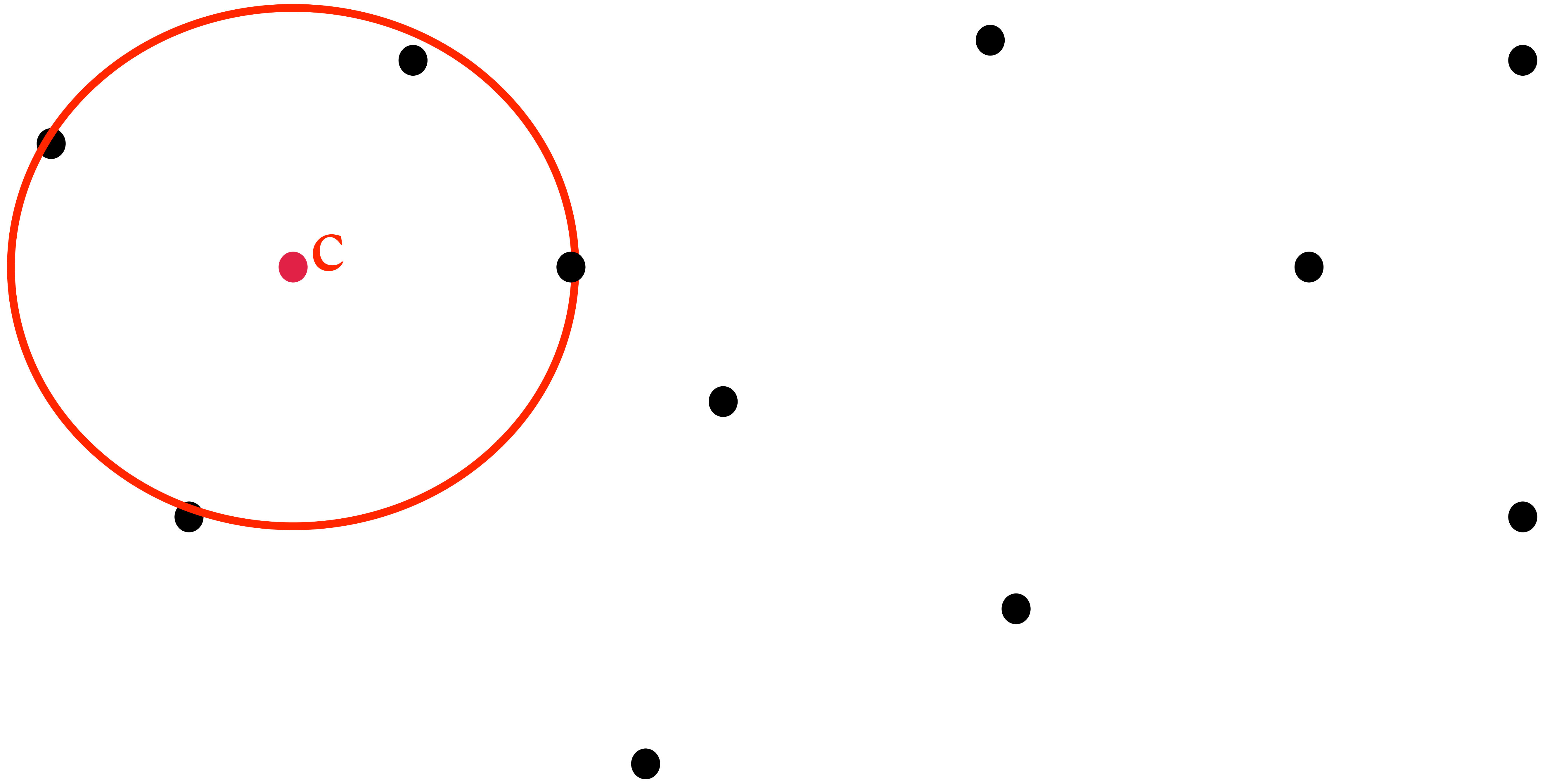
Problem Statement and Background

- Problem Statement:
  - Input: Given n points (objects) in the plane and radius r (i.e., sensor covering range)
  - Output: The center c (to place the sensor) of the cycle of radius r enclosing most points
- Applications:
  - Wireless Sensor Network
  - Facility Location
  - Urban Planning
  - Clustering



Problem Statement and Background

- Problem Statement:
  - Input: Given n points (objects) in the plane and radius r (i.e., sensor covering range)
  - Output: The center c (to place the sensor) of the cycle of radius r enclosing most points
- Applications:
  - Wireless Sensor Network
  - Facility Location
  - Urban Planning
  - Clustering



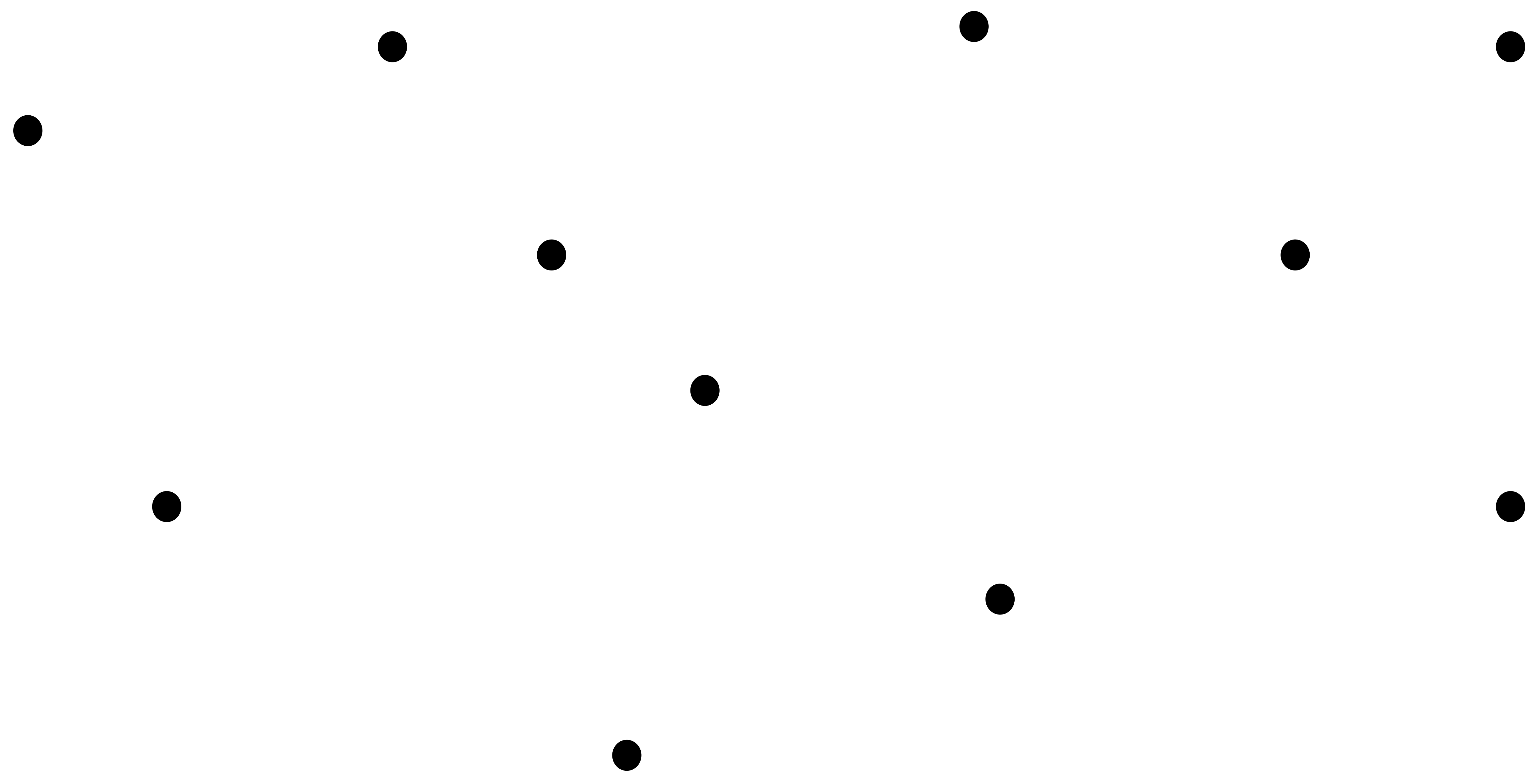
Objectives

- Design algorithms to find the coordinates of the center of the r-circle (i.e., the cycle of radius r) enclosing most input points(objects) under three different scenarios:
  - Two-dimension Version: Input points and the center could be anywhere in the plane;
  - Line-constrained Version: Input points are in the plane but the center is required to be on a given line L;
  - One-dimension Version: Input points and the center are on a given line L.

Technique Approach

Two-dimension Version

Computing the r-Cycle Enclosing Most Points

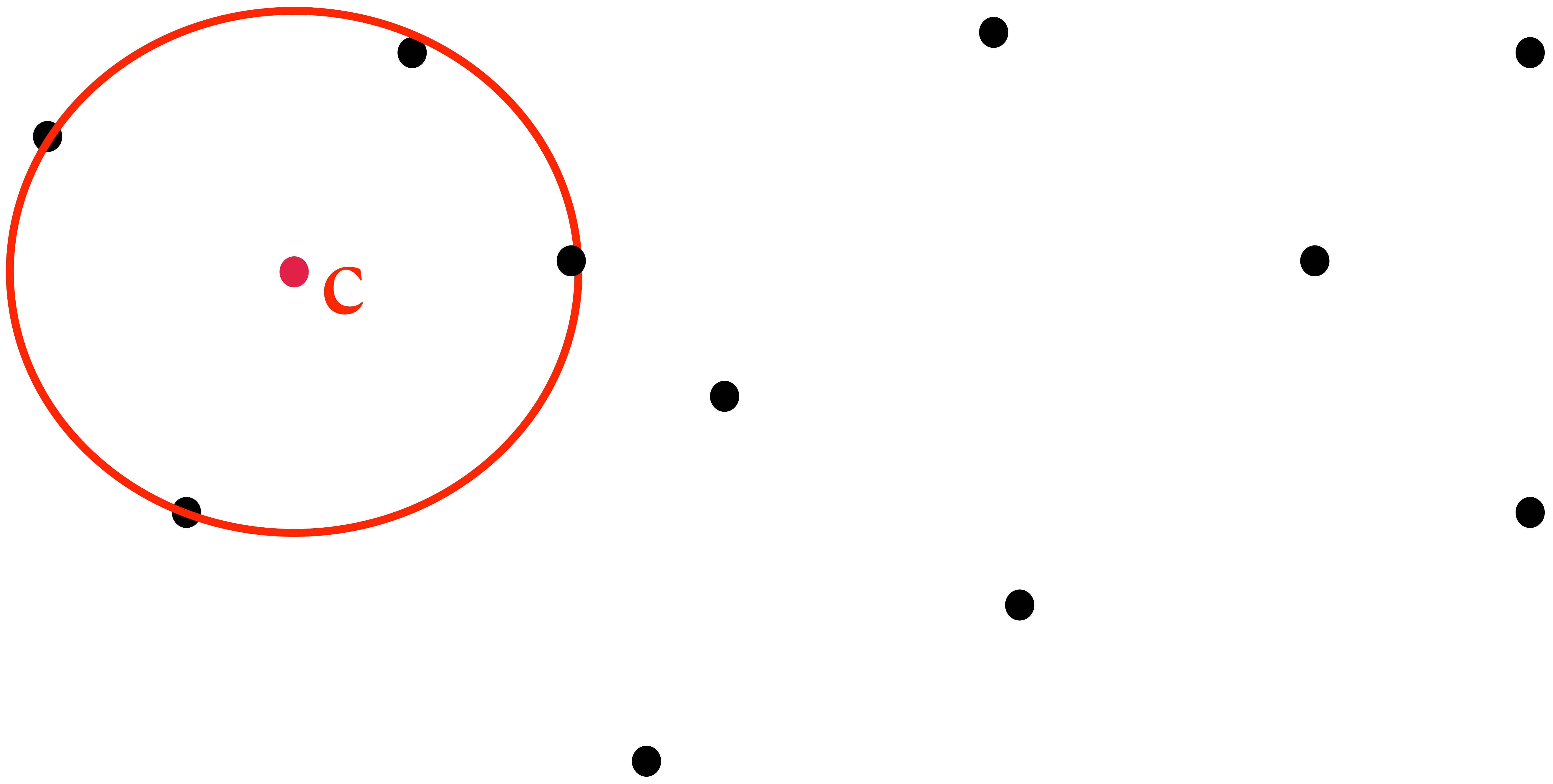


Input: n points

Radius $r > 0$

Output: The center c of the r-cycle enclosing most points.

Computing the r-Cycle Enclosing Most Points

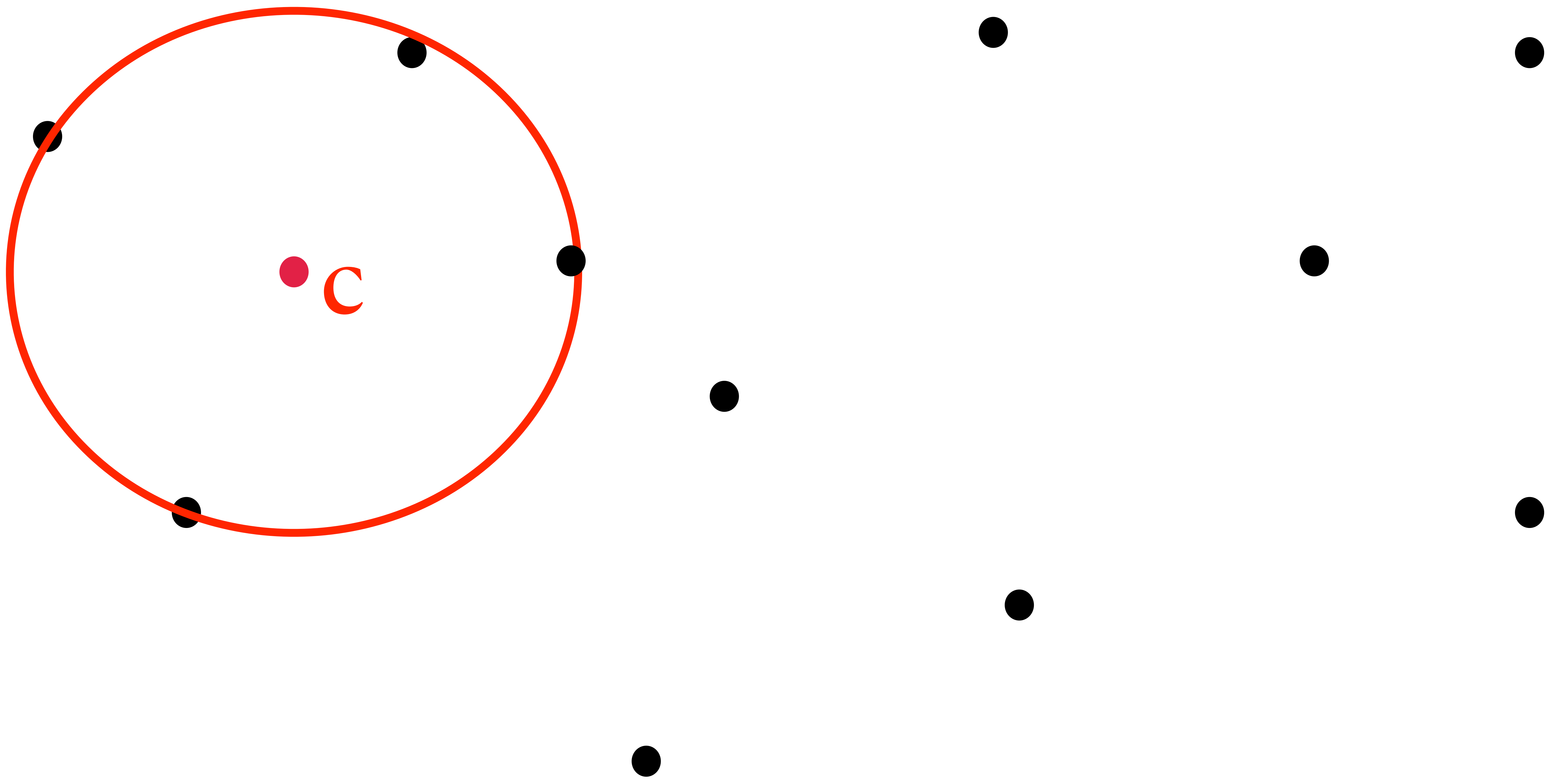


Input: n points

Radius r > 0

Output: The center c of the r-cycle enclosing most points.

Computing the r-Cycle Enclosing Most Points



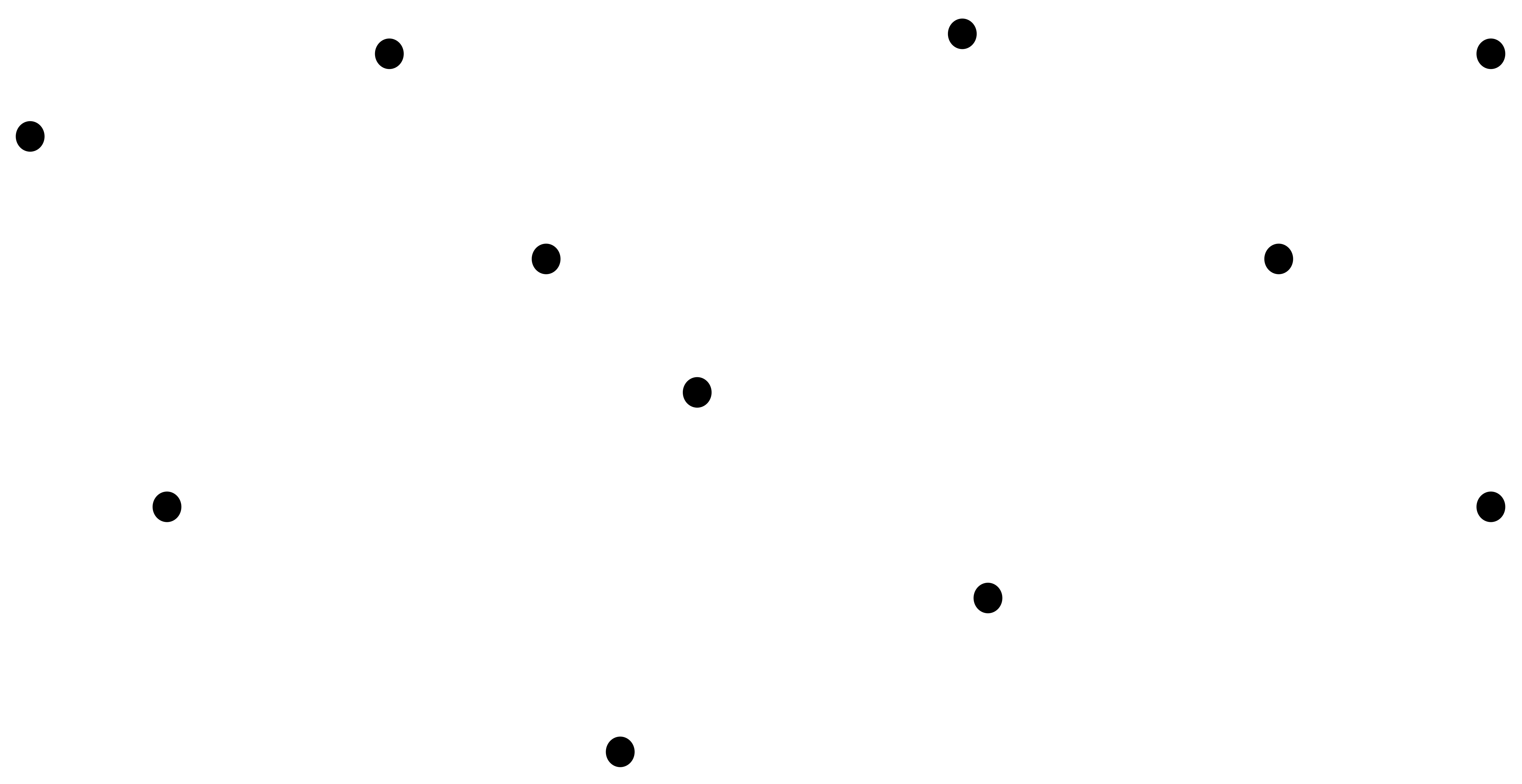
Input: n points

Radius r > 0

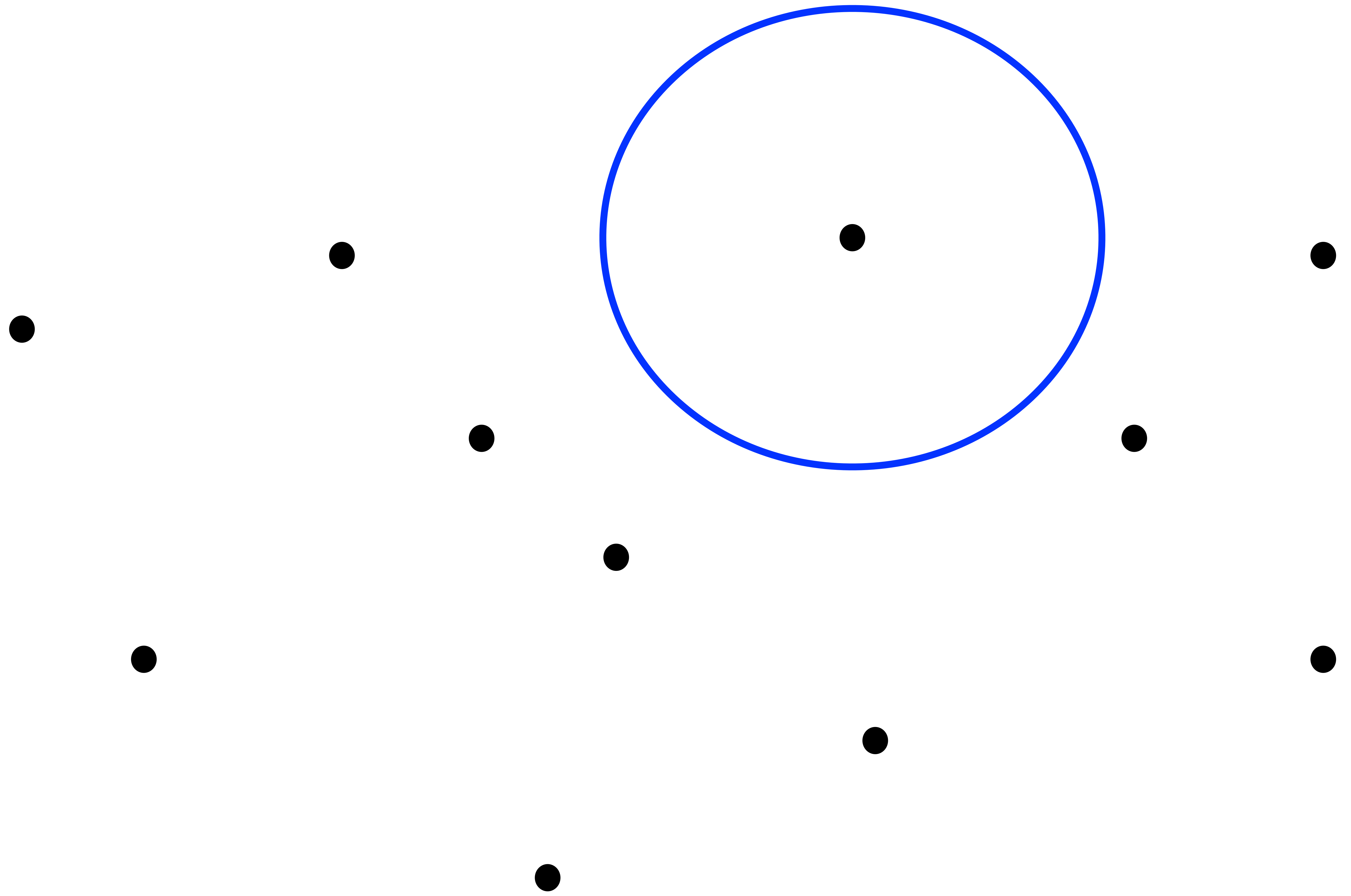
Output: The center c of the r-cycle enclosing most points.

Observation: There is at least one input point on the optimal r-cycle.

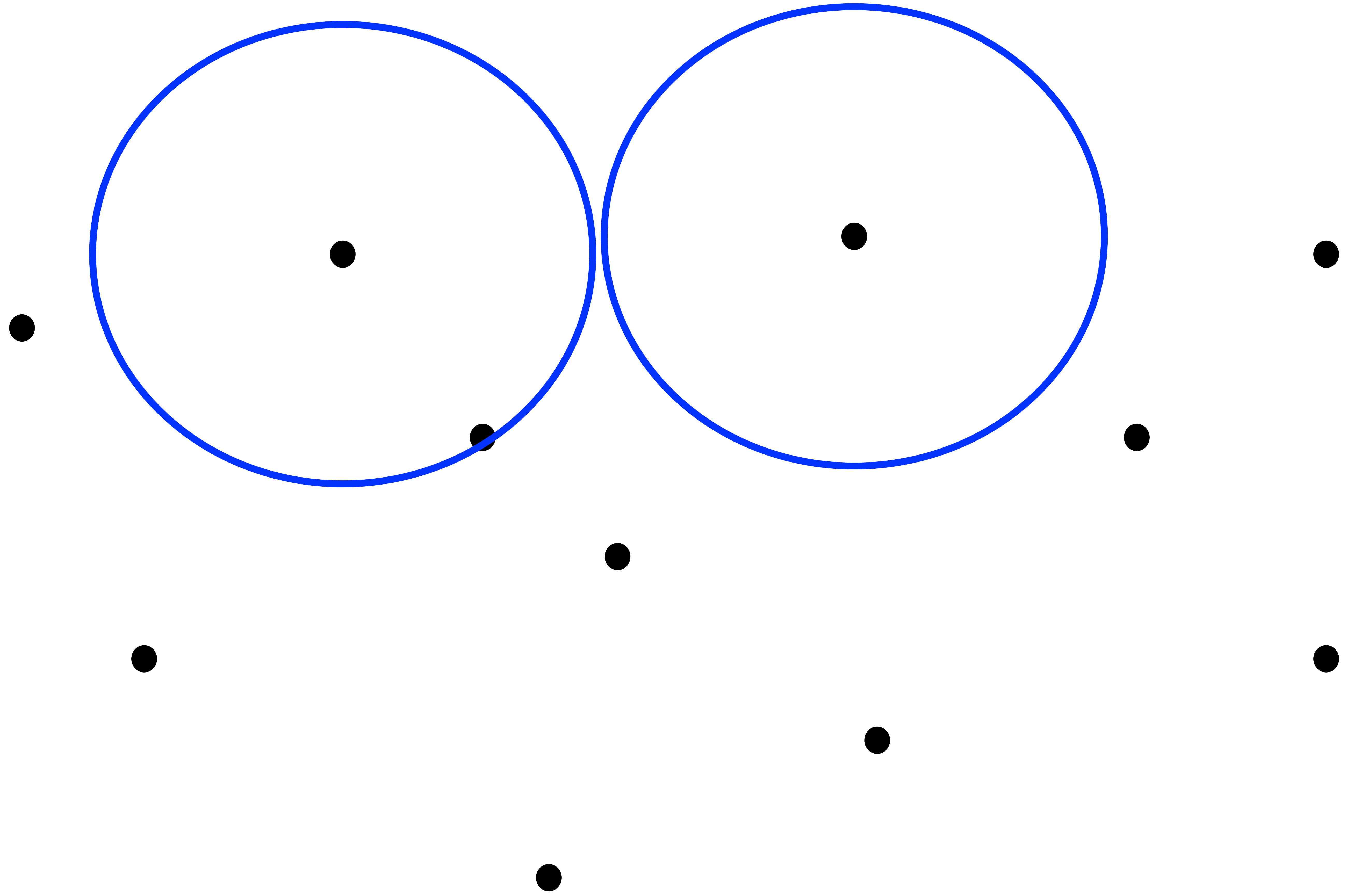
Computing the r-Cycle Enclosing Most Points(Cont.)



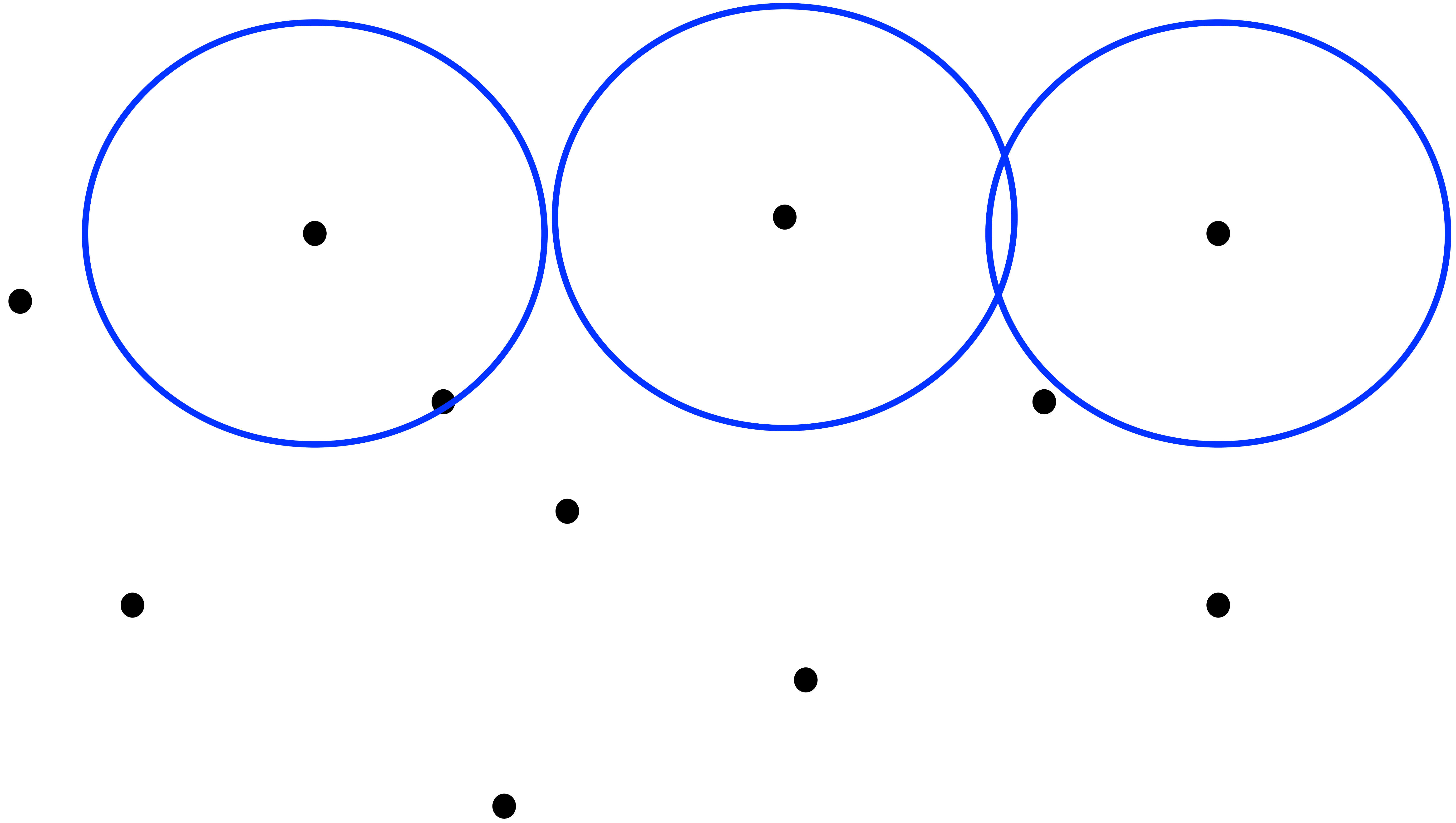
Computing the r-Cycle Enclosing Most Points(Cont.)



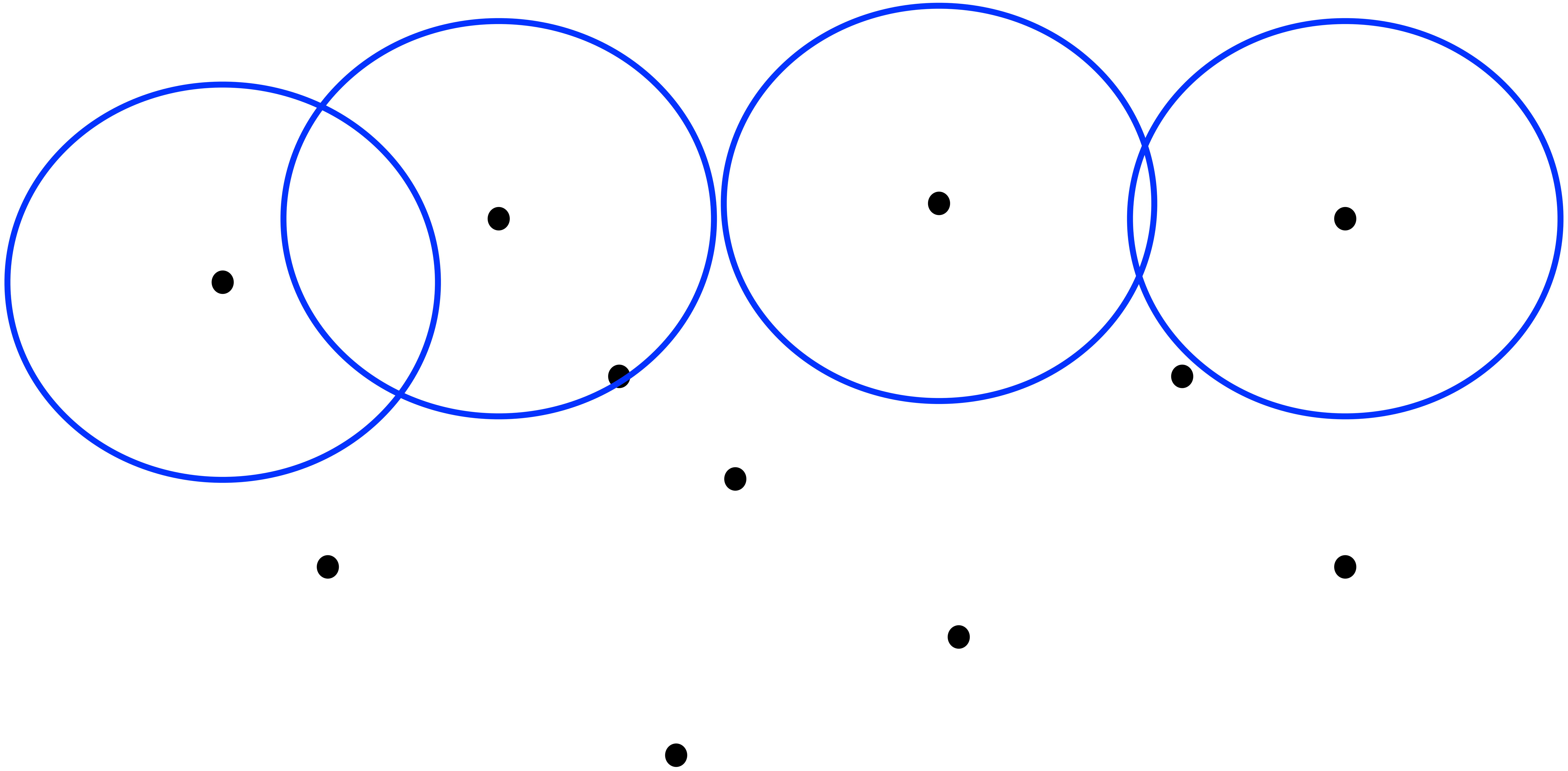
Computing the r-Cycle Enclosing Most Points(Cont.)



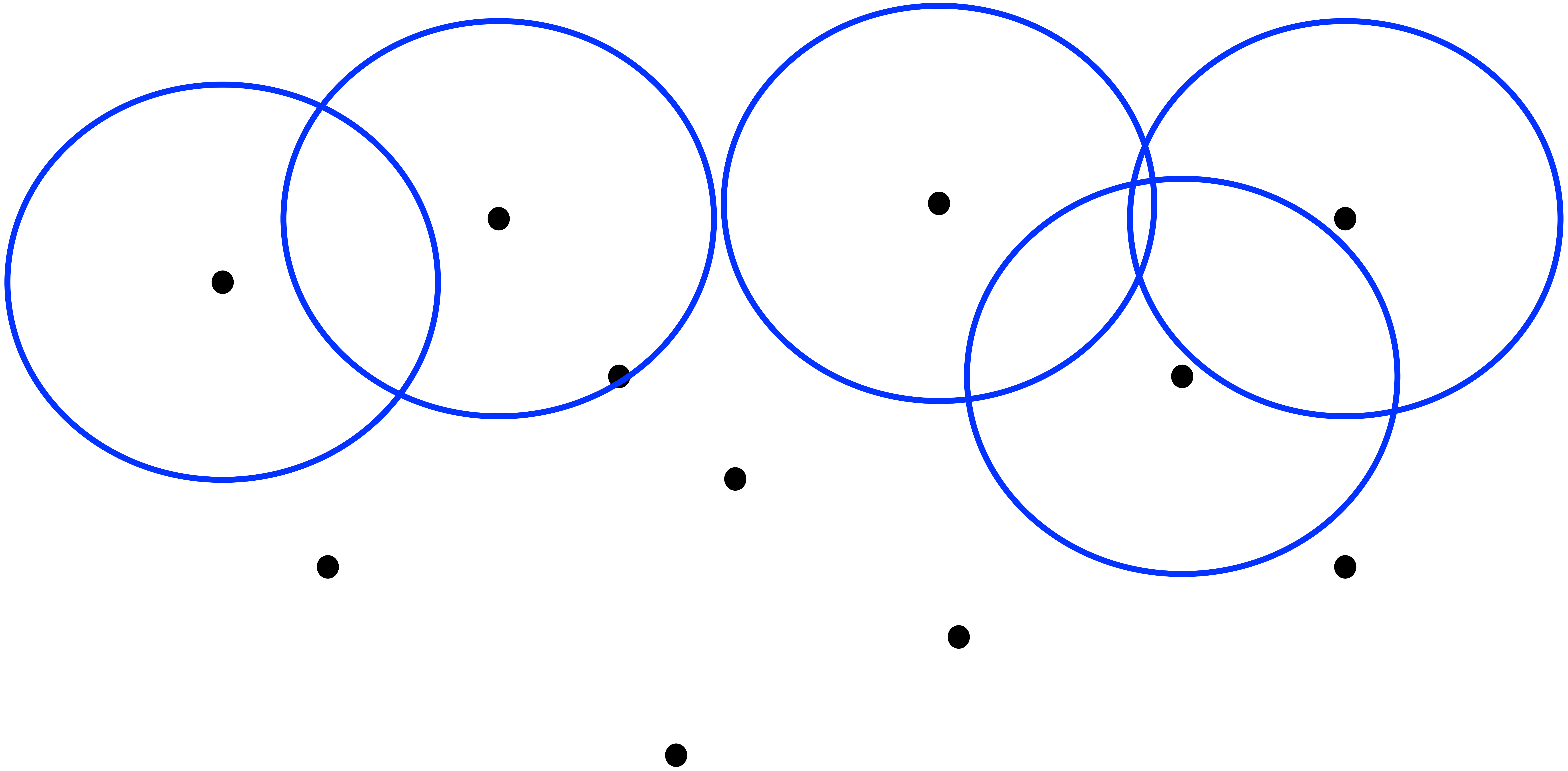
Computing the r-Cycle Enclosing Most Points(Cont.)



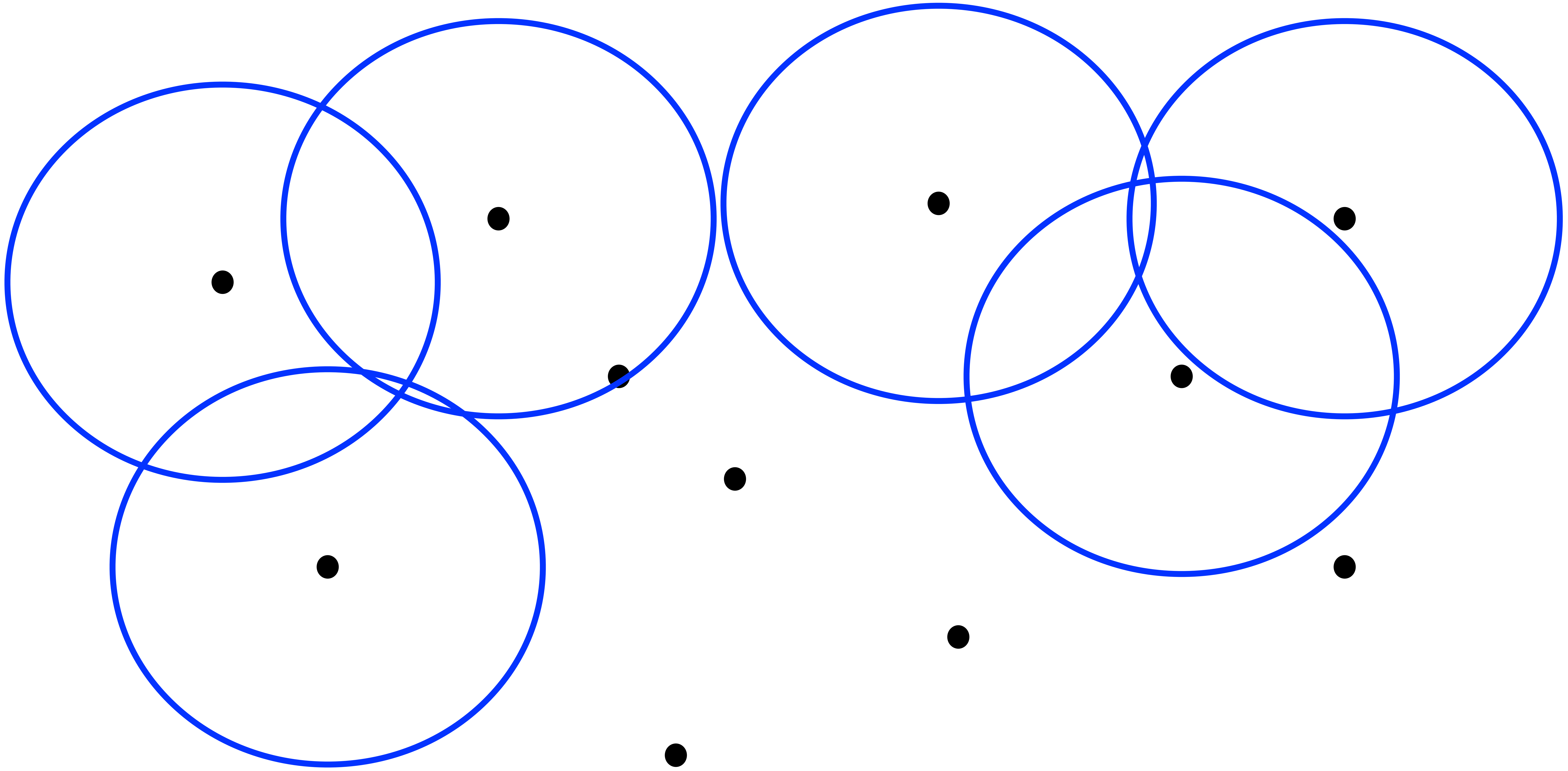
Computing the r-Cycle Enclosing Most Points(Cont.)



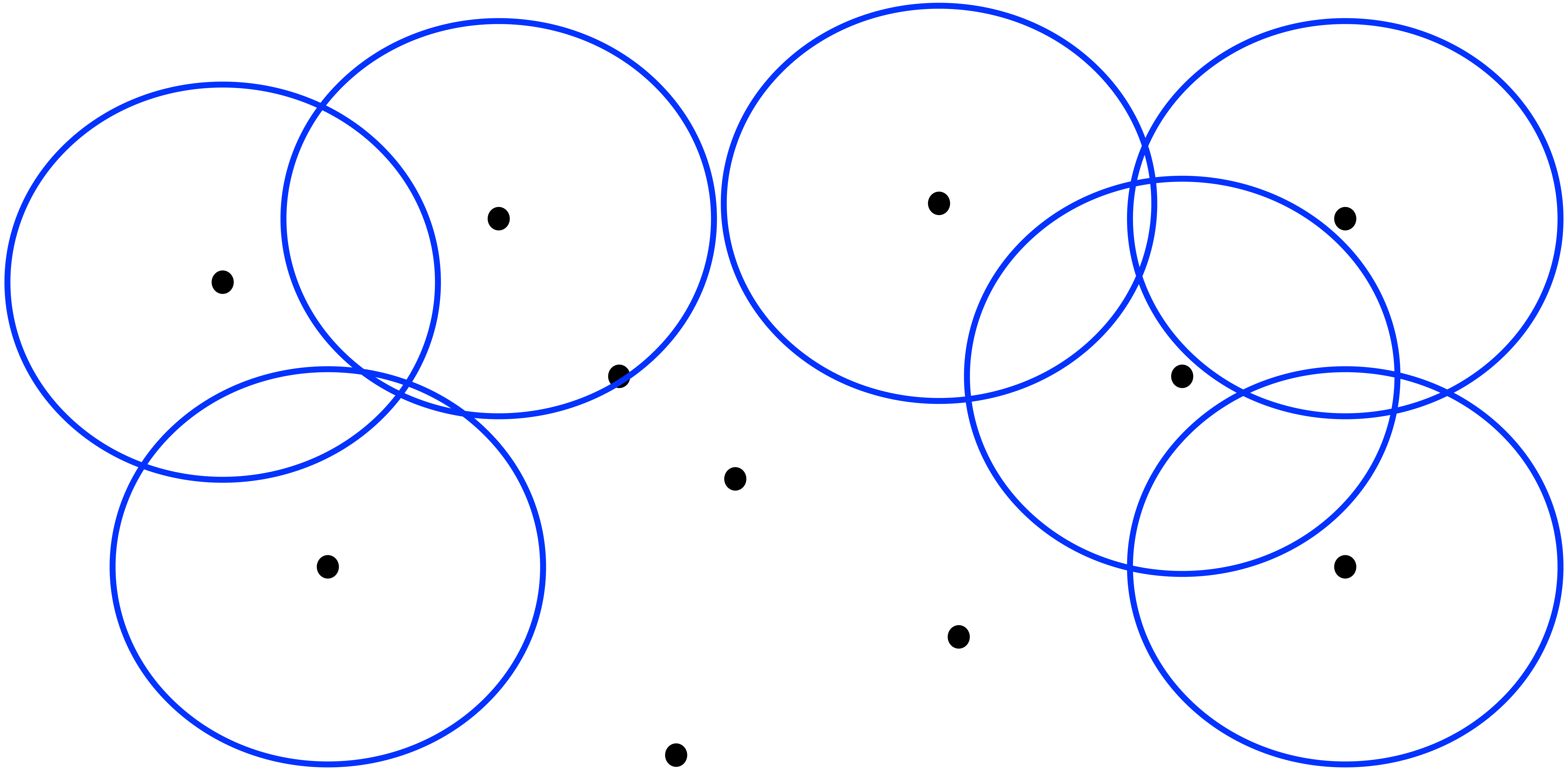
Computing the r-Cycle Enclosing Most Points(Cont.)



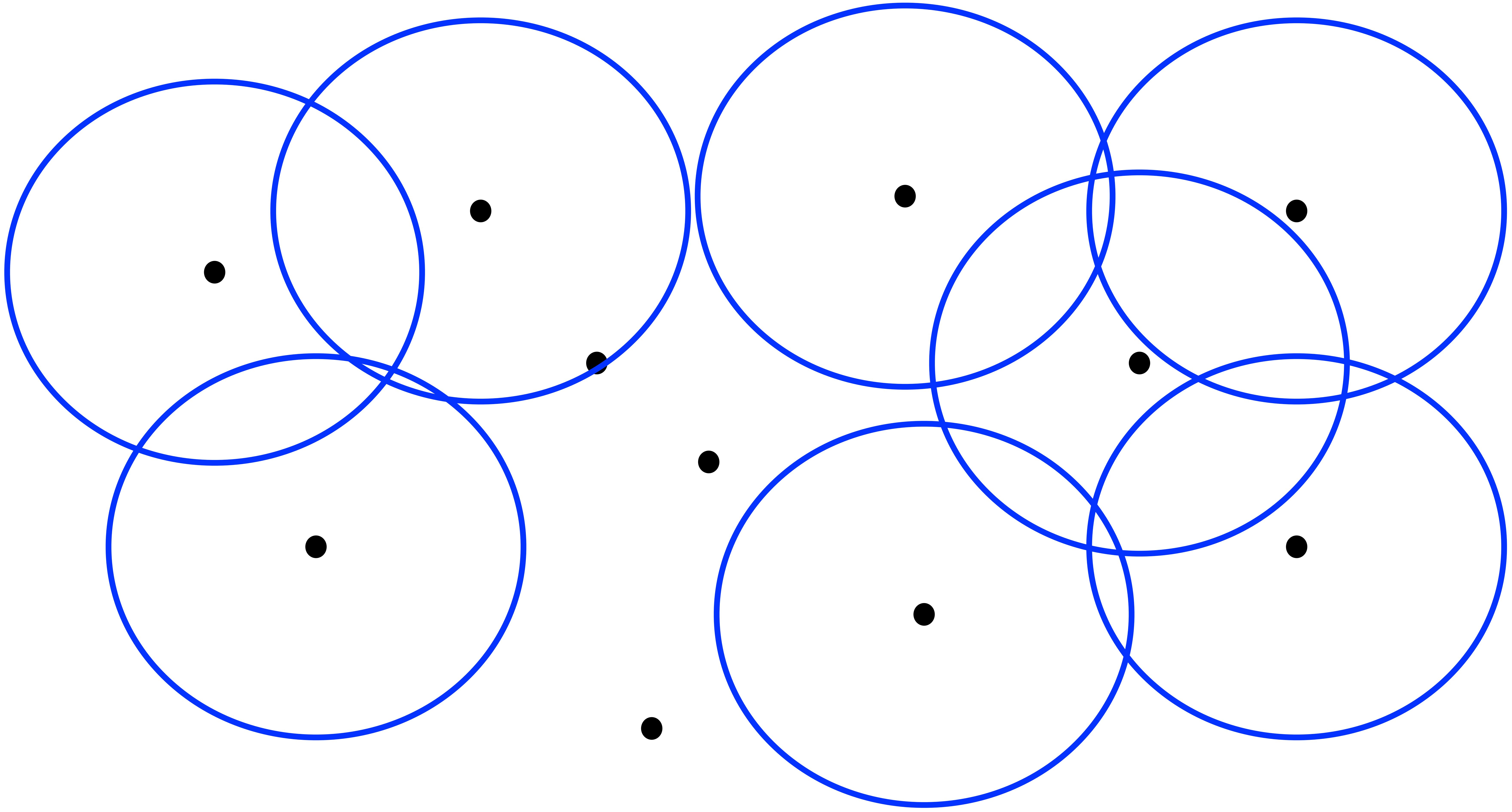
Computing the r-Cycle Enclosing Most Points(Cont.)



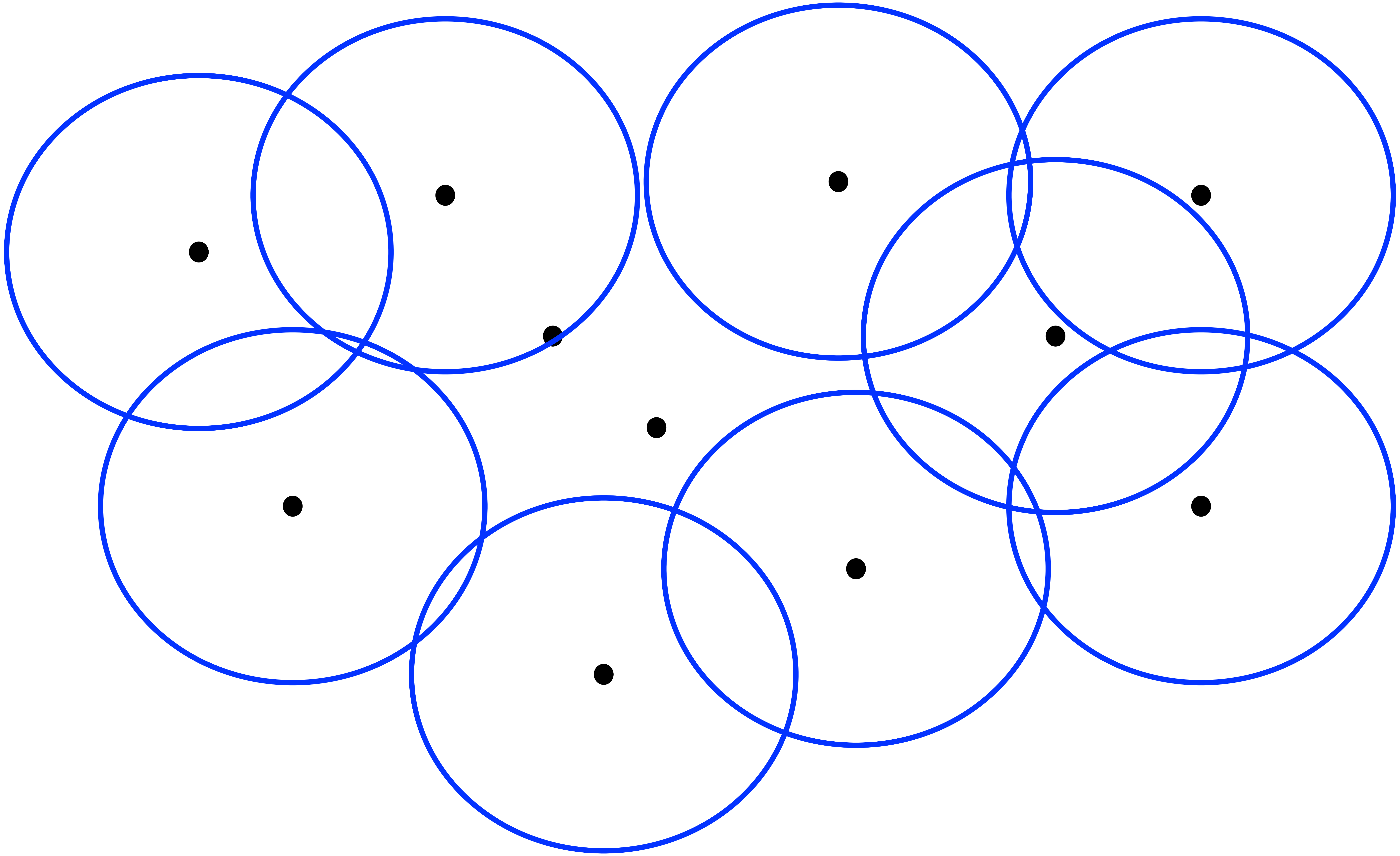
Computing the r-Cycle Enclosing Most Points(Cont.)



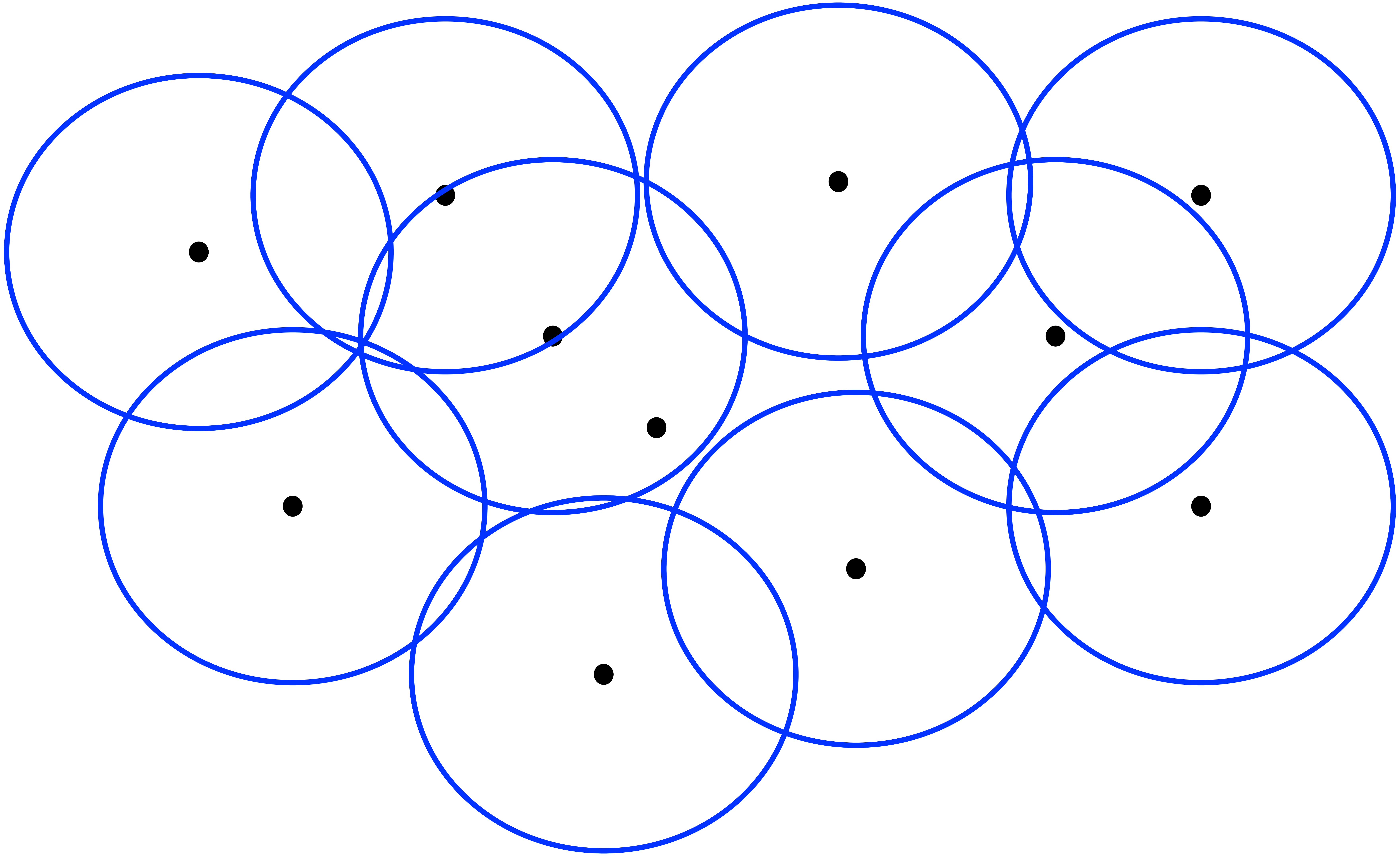
Computing the r-Cycle Enclosing Most Points(Cont.)



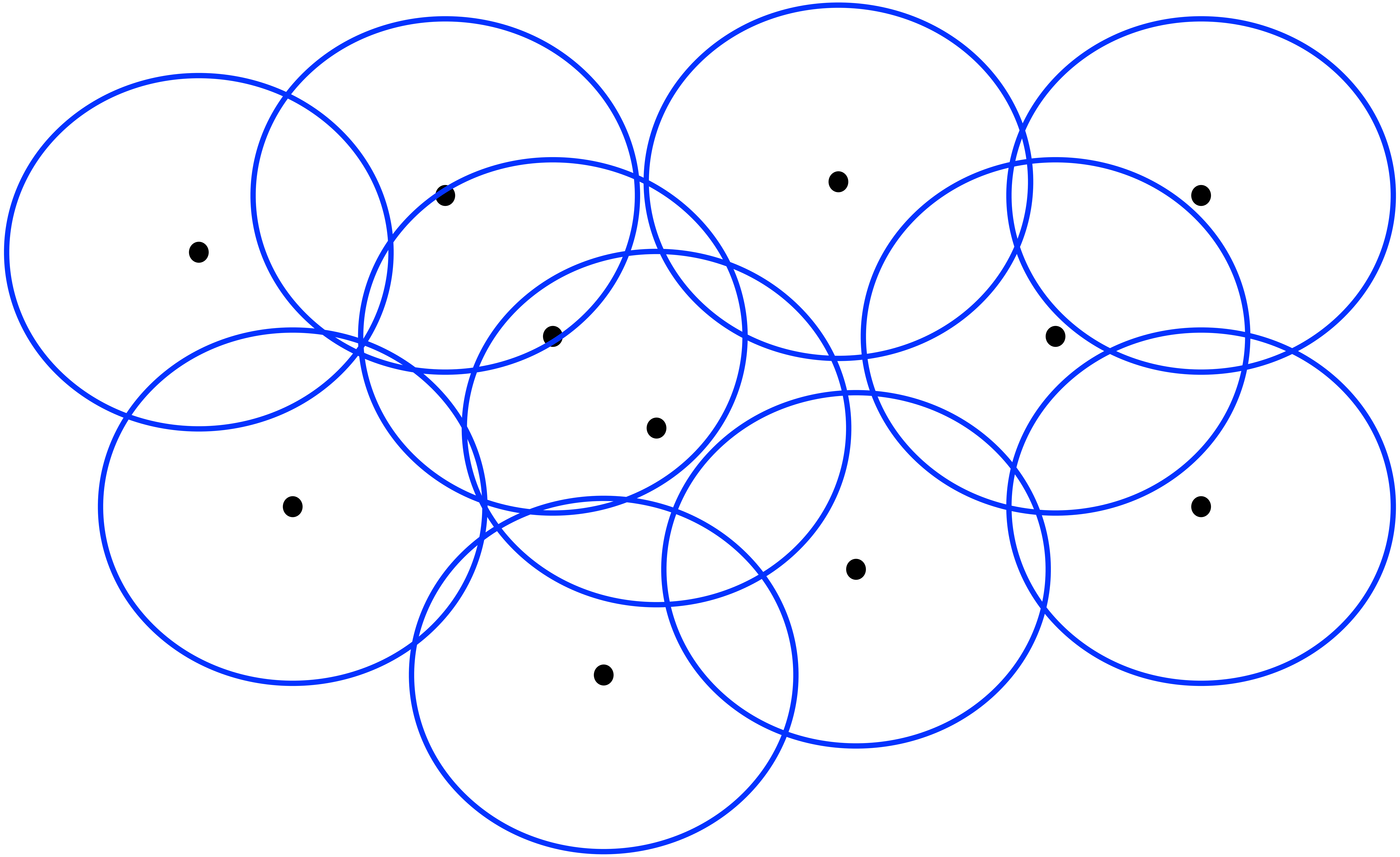
Computing the r-Cycle Enclosing Most Points(Cont.)



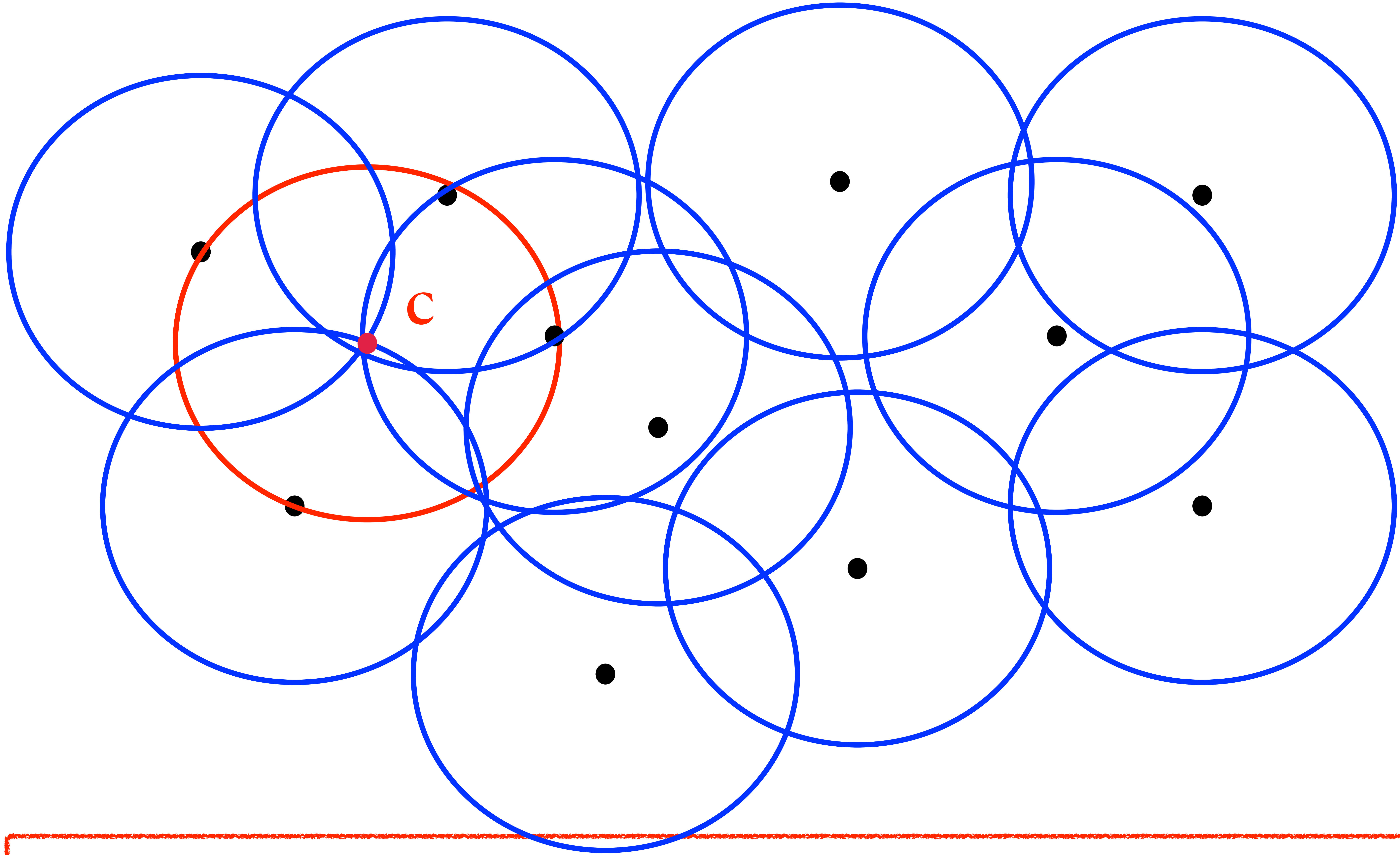
Computing the r-Cycle Enclosing Most Points(Cont.)



Computing the r-Cycle Enclosing Most Points(Cont.)

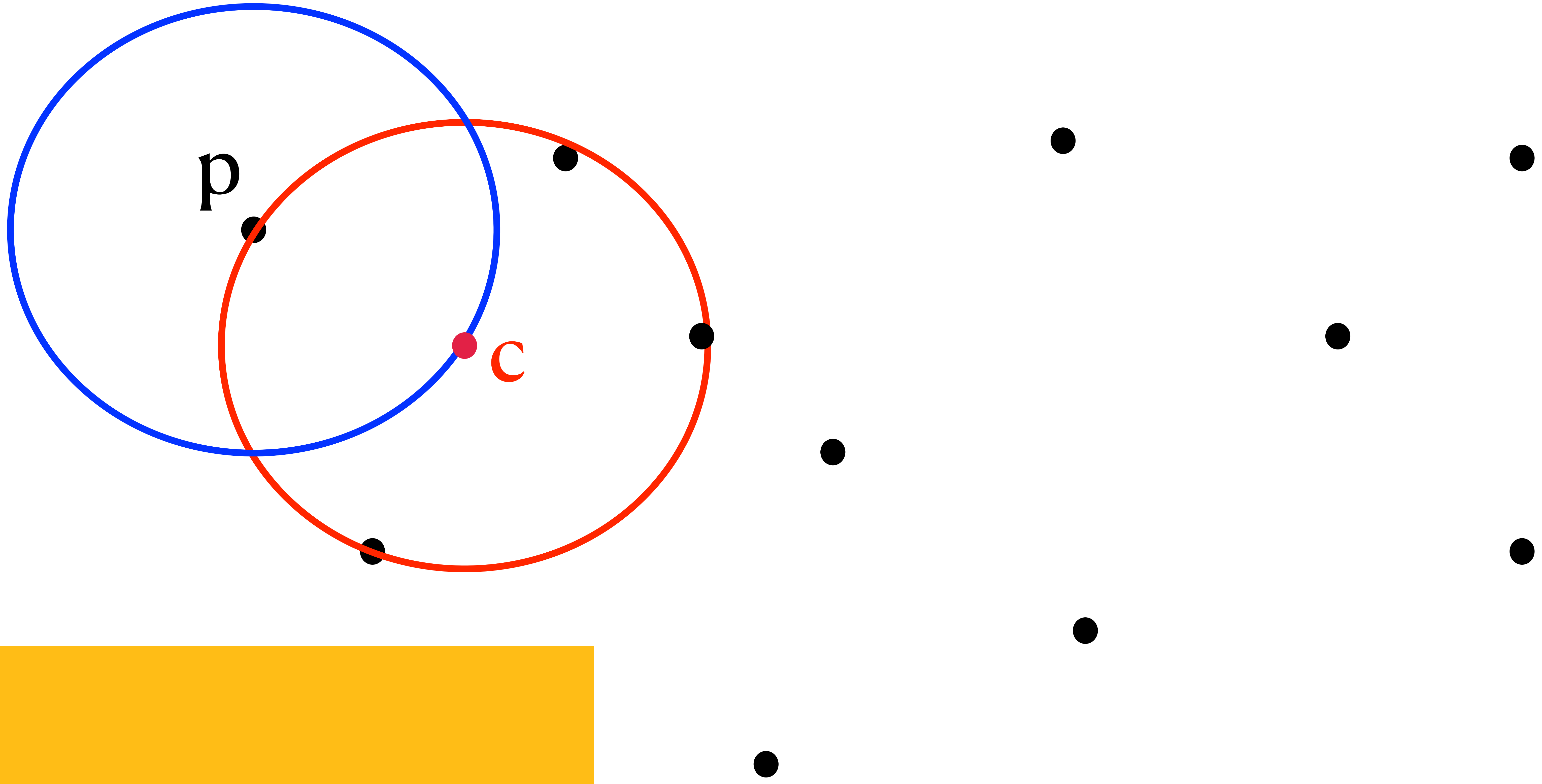


Computing the r-Cycle Enclosing Most Points(Cont.)



The **center c** is on the **boundary** of the **r-cycle** of a point.

Computing the r-Cycle Enclosing Most Points(Cont.)



Algorithm:

For every input point p:

Compute the r-cycle enclosing most points centered at a point on p's r-cycle.

——— The Constrained Version

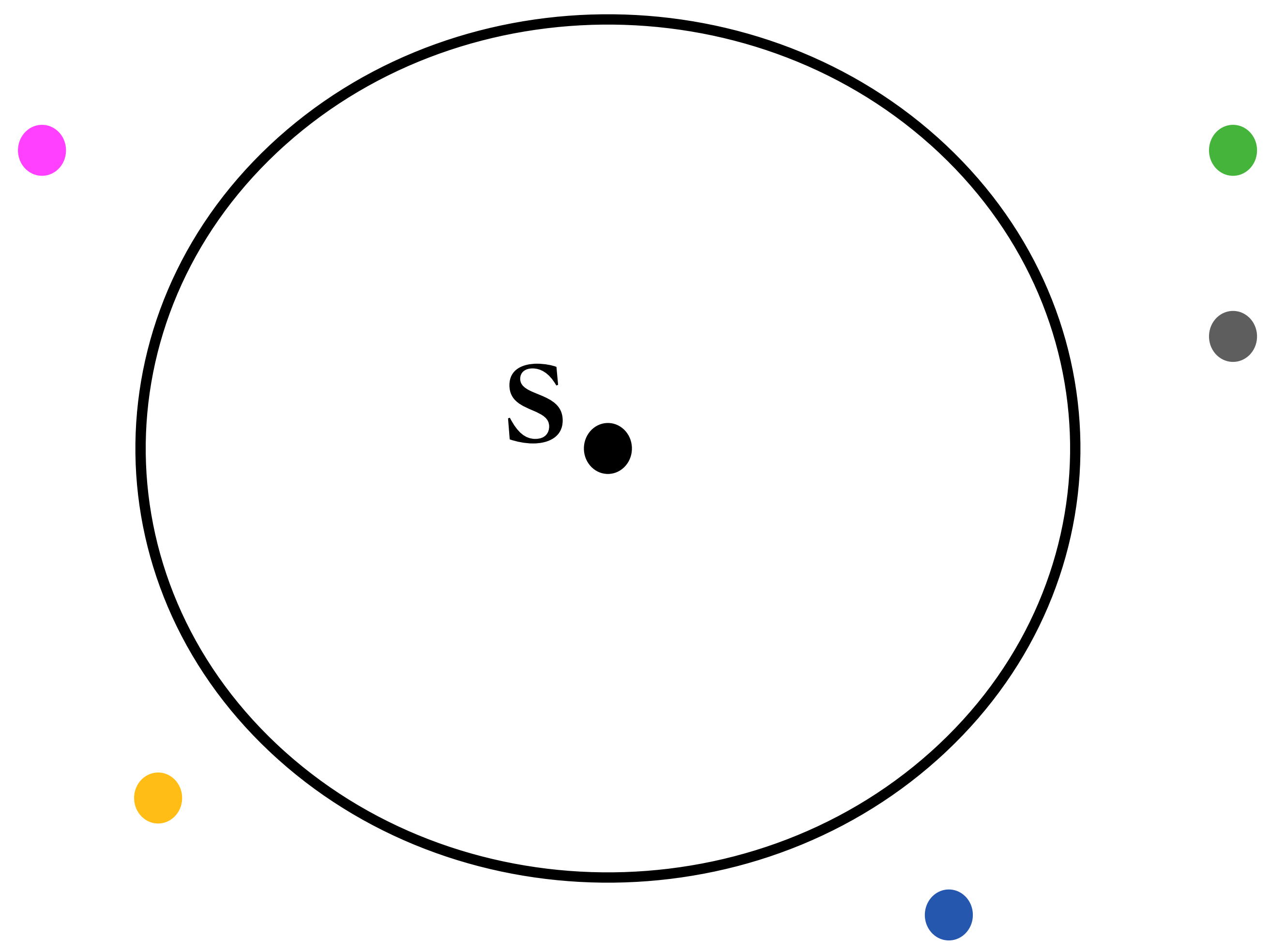
The Constrained Version

Input: constraint point s

n points

Radius r > 0

Output: The center c of the r-cycle enclosing most points s.t c is on the r-cycle of s.



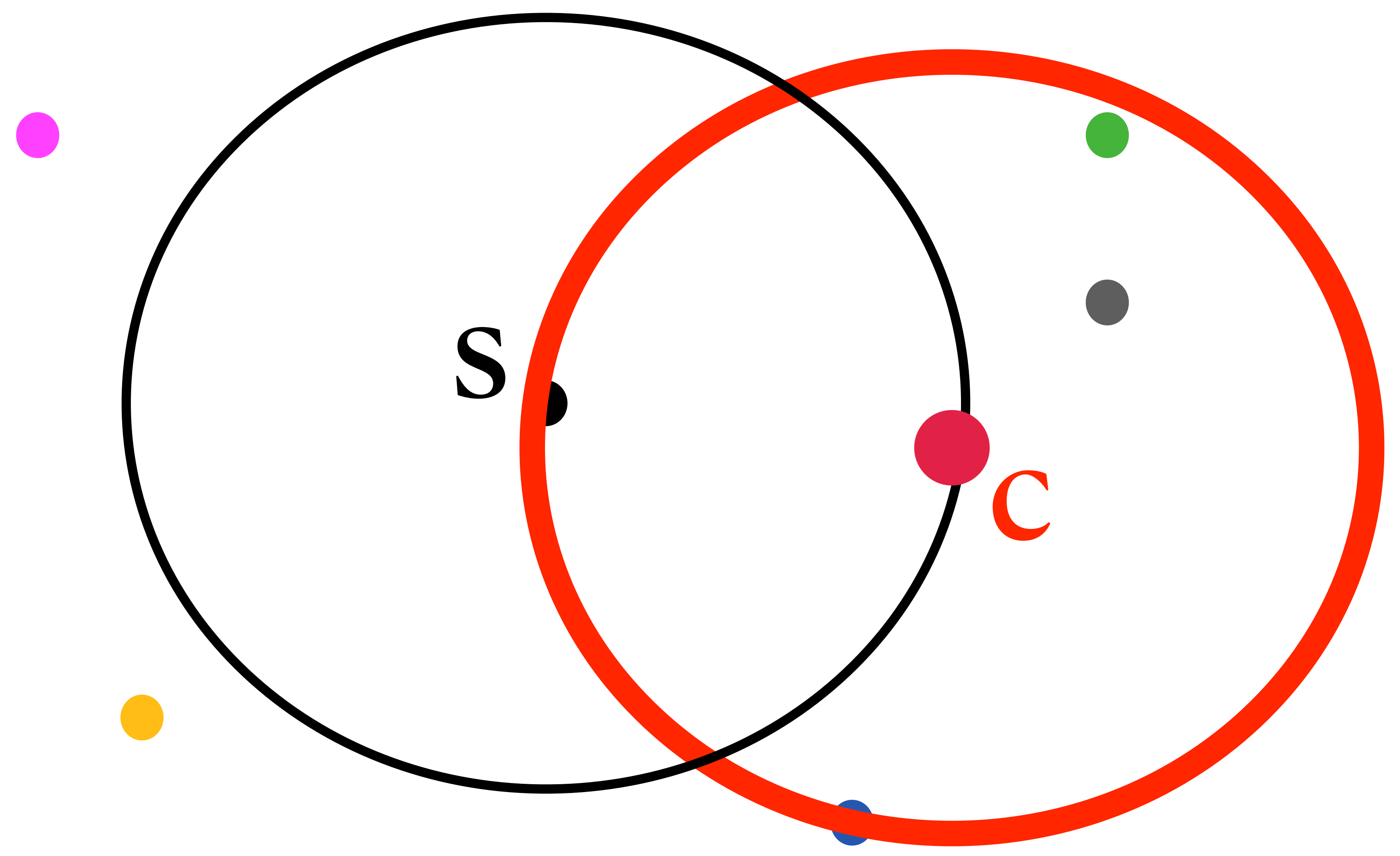
The Constrained Version

Input: constraint point s

n points

Radius $r > 0$

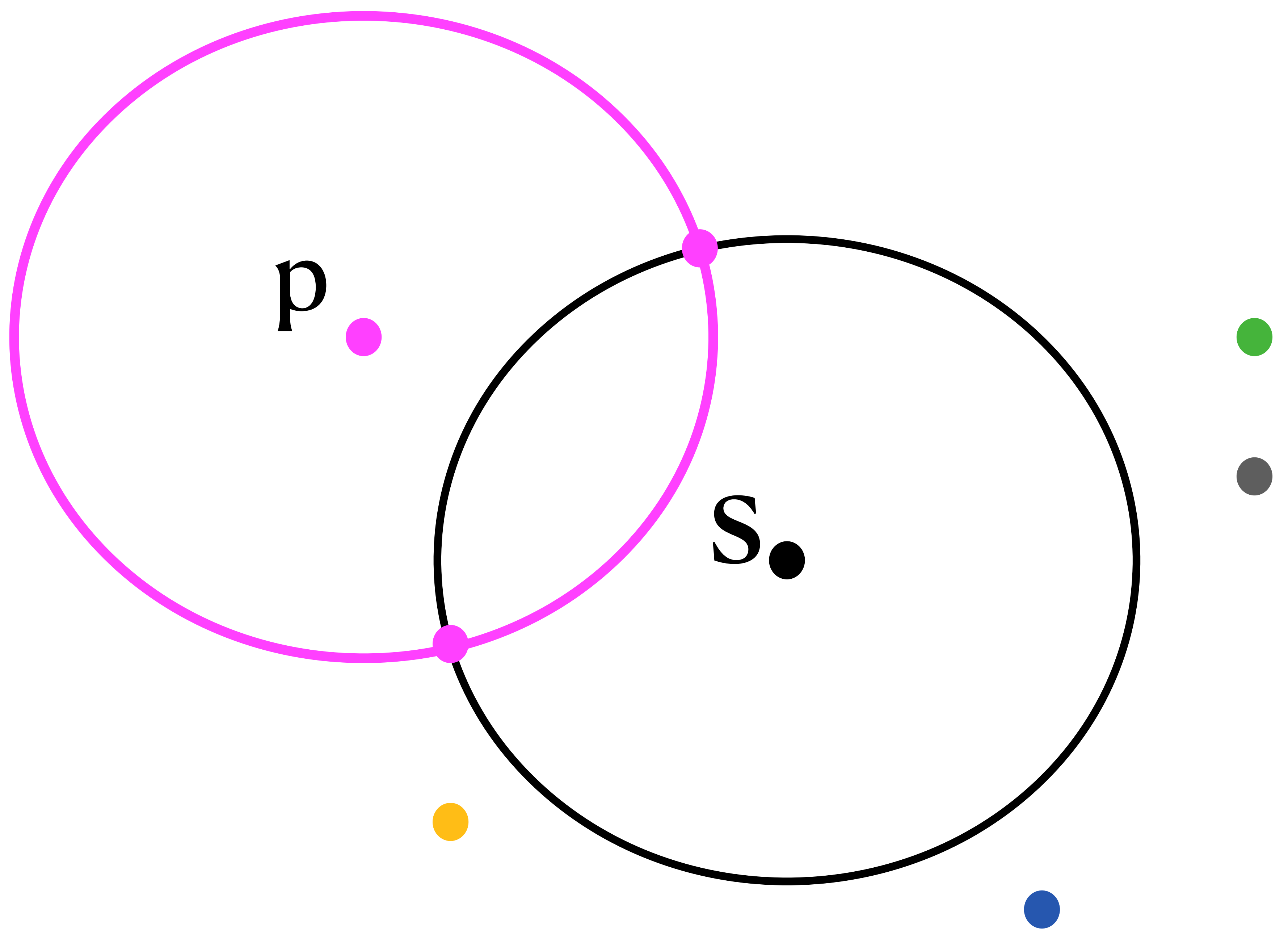
Output: The center c of the r-cycle enclosing most points s.t c is on the r-cycle of s.



The Constrained Version (Cont.)

Observation:

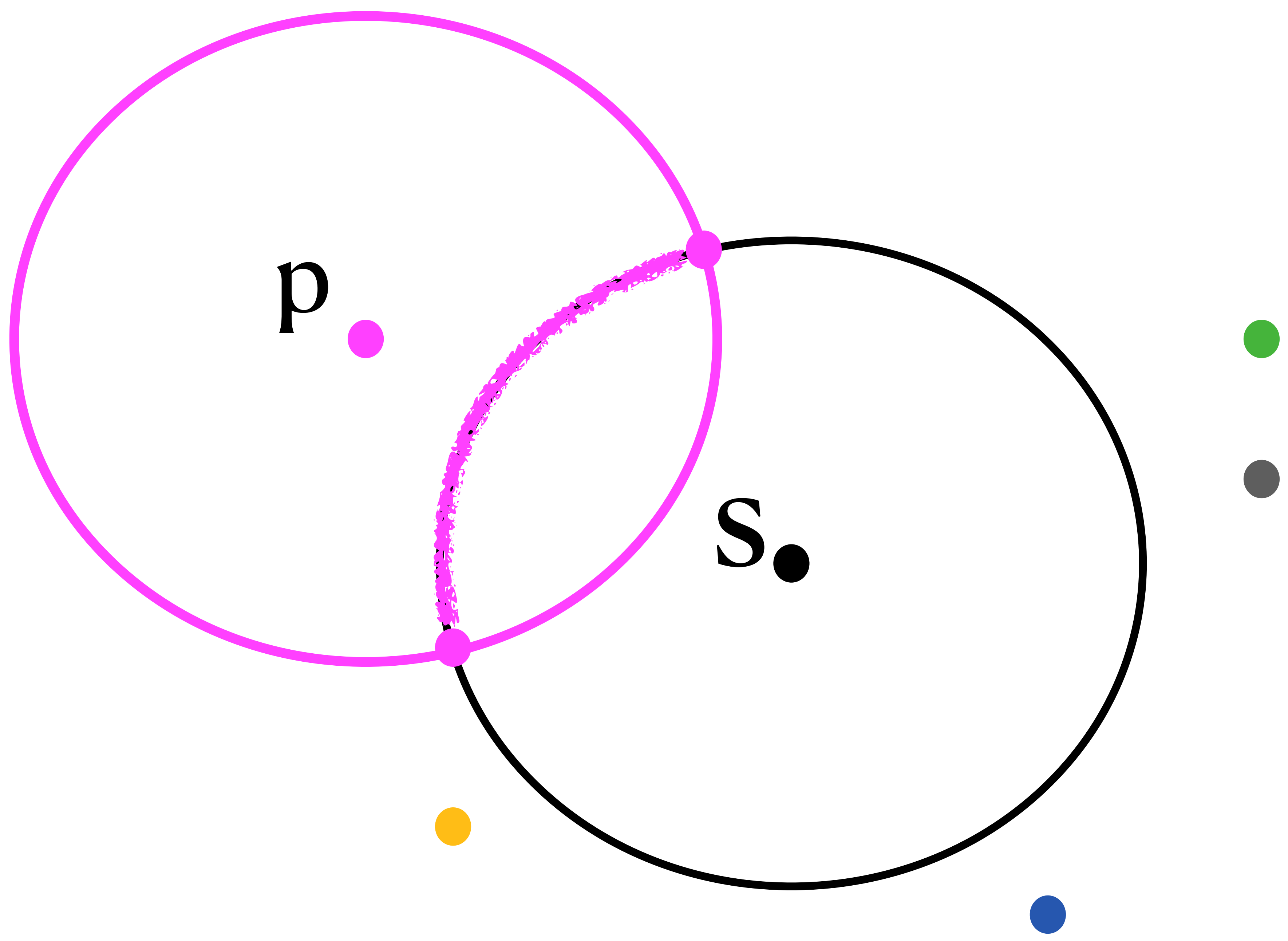
For every input point p, its r-cycle intersects s' r-cycle at an arc, and the r-cycle centered at any point of the arc encloses p and s.



The Constrained Version (Cont.)

Observation:

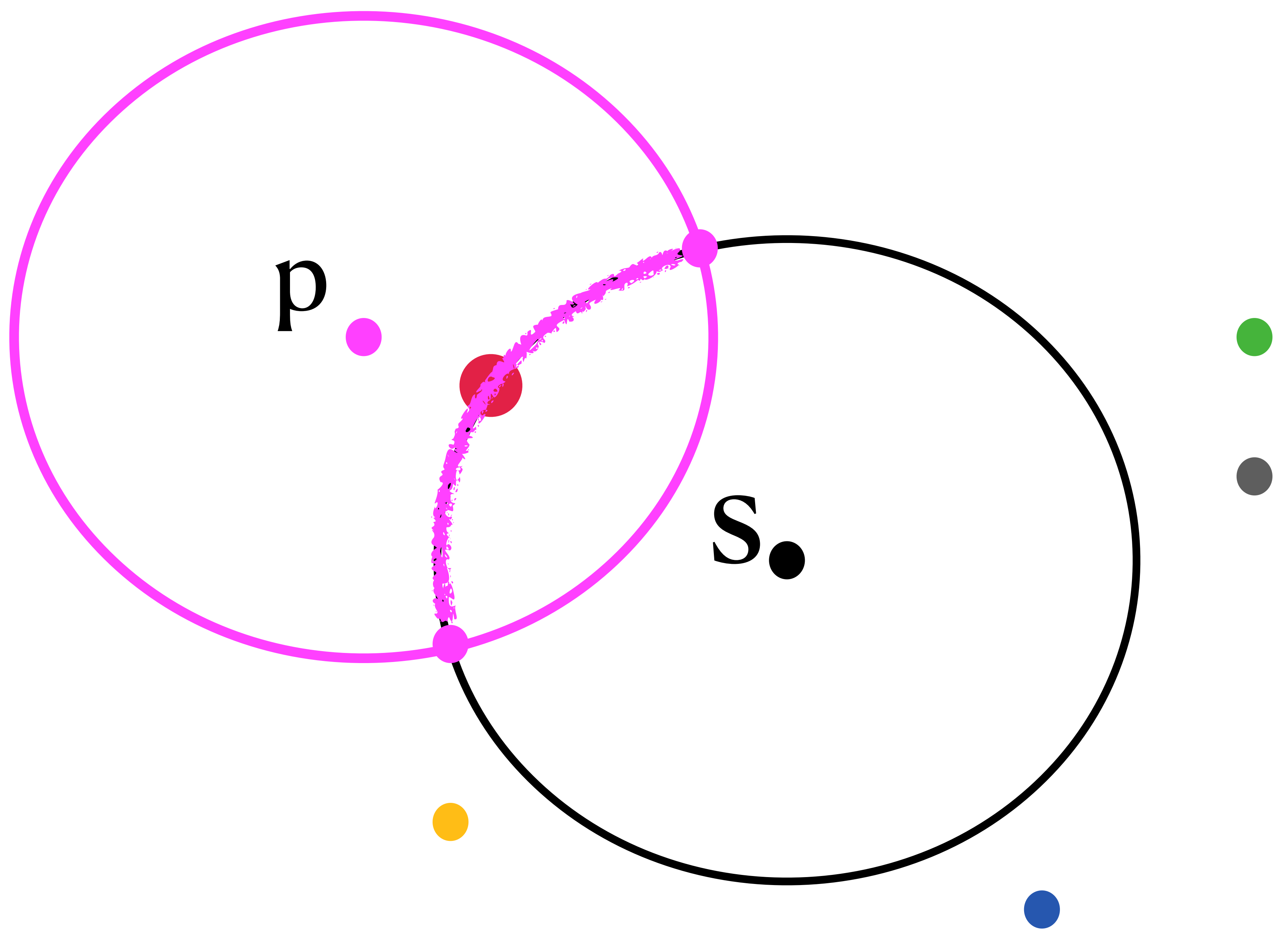
For every input point p, its r-cycle intersects s' r-cycle at an arc, and the r-cycle centered at any point of the arc encloses p and s.



The Constrained Version (Cont.)

Observation:

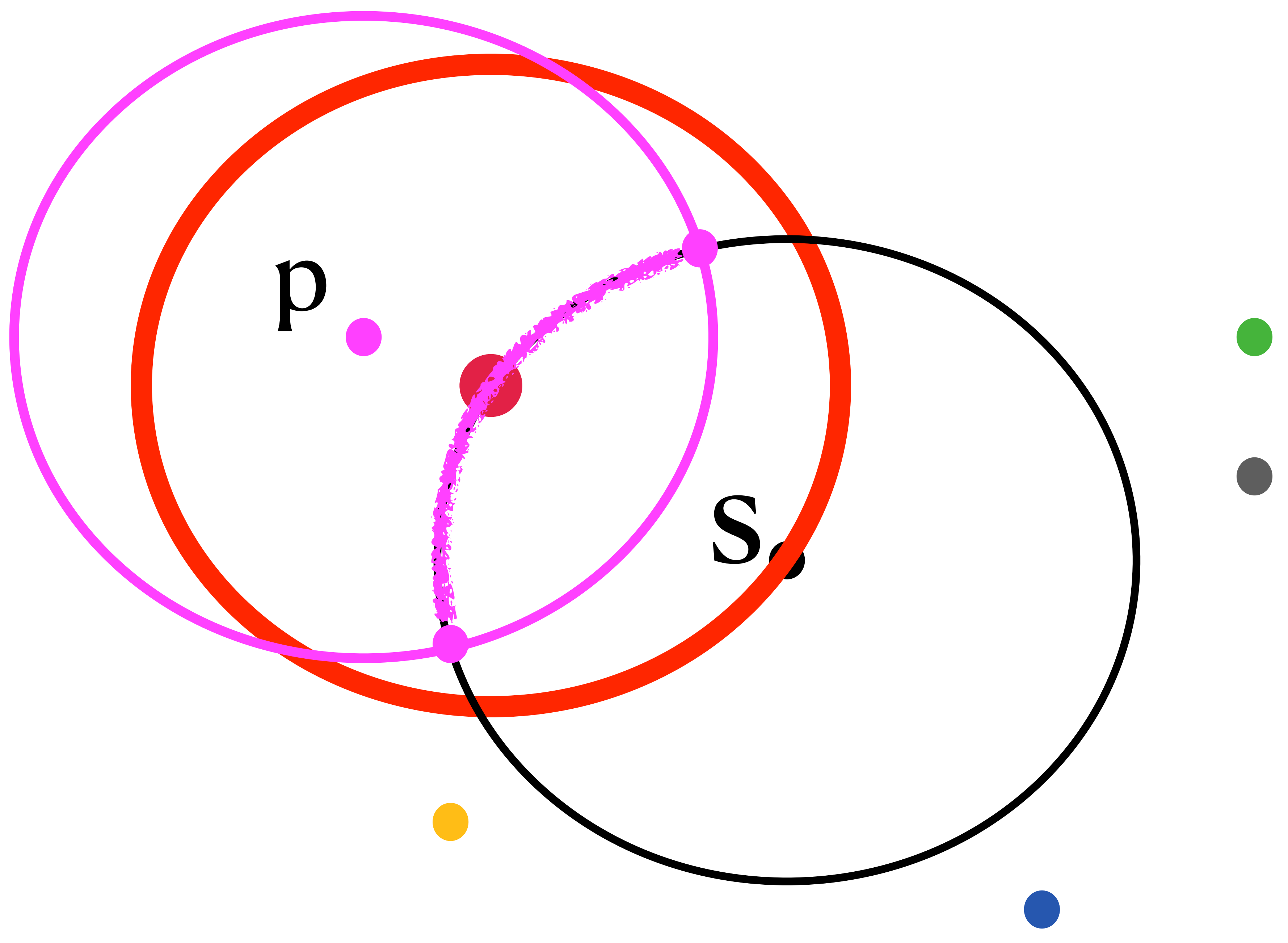
For every input point p, its r-cycle intersects s' r-cycle at an arc, and the r-cycle centered at any point of the arc encloses p and s.



The Constrained Version (Cont.)

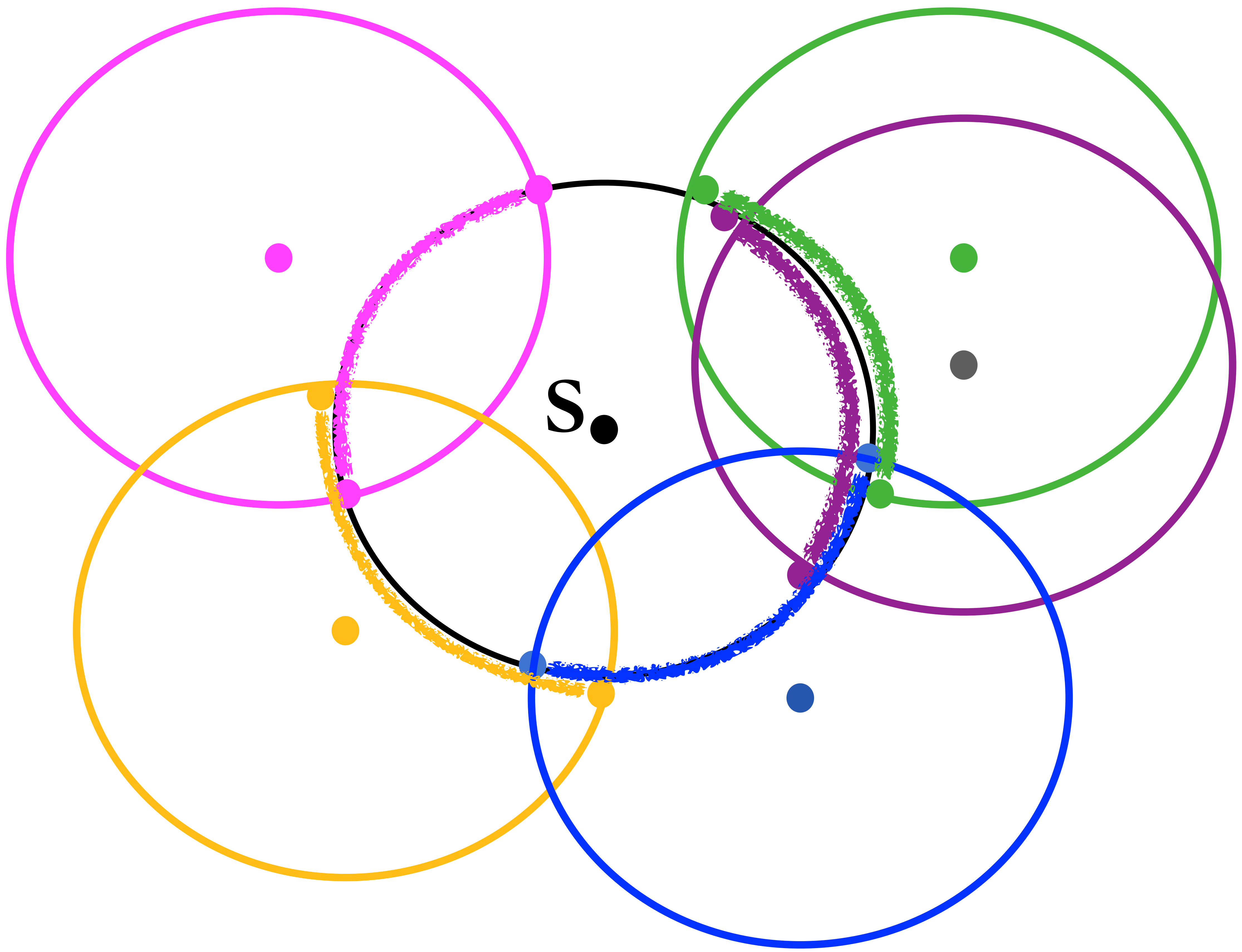
Observation:

For every input point p, its r-cycle intersects s' r-cycle at an arc, and the r-cycle centered at any point of the arc encloses p and s.



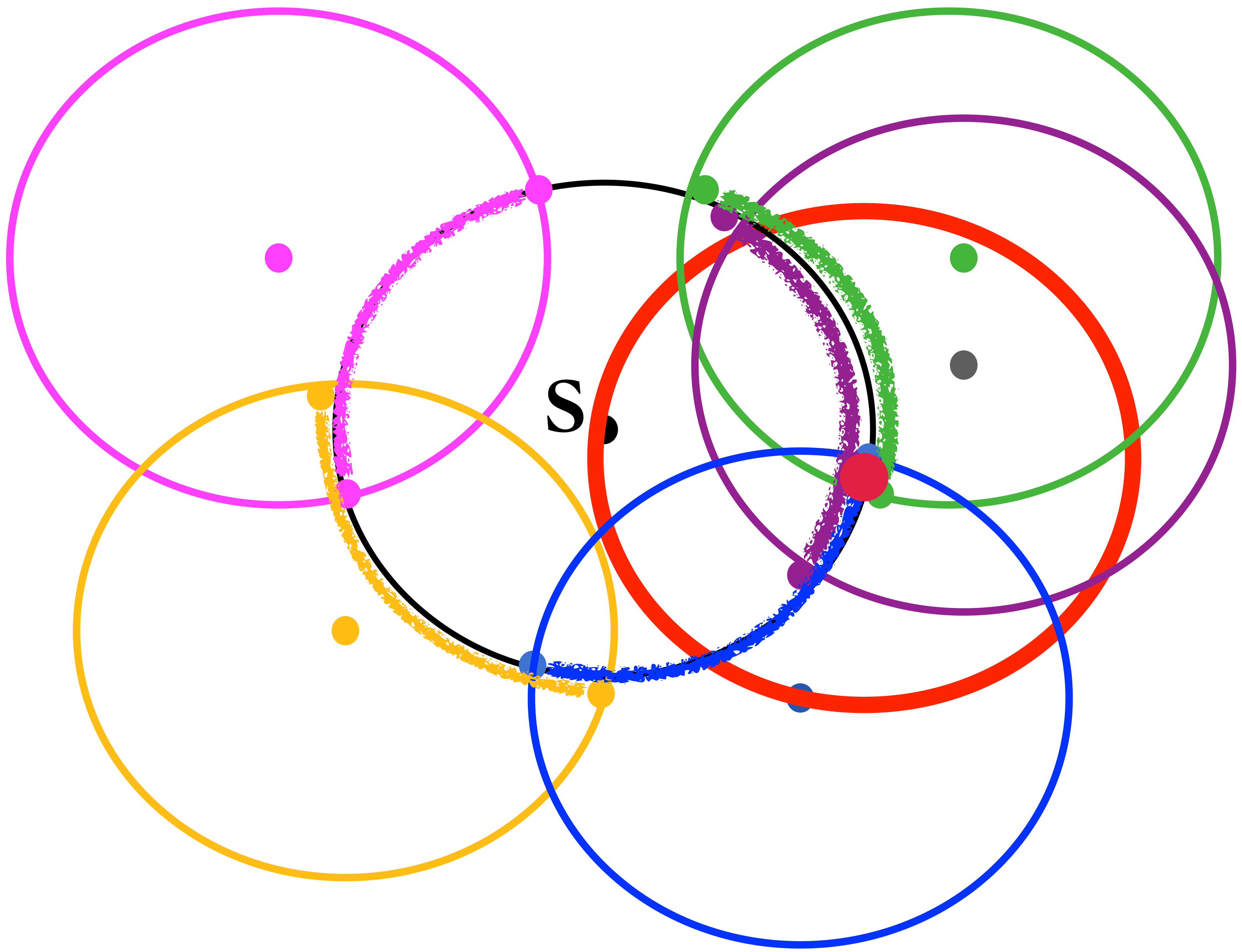
The Constrained Version (Cont.)

Lemma: The center c is any point of the arc that is the intersection of most arcs.



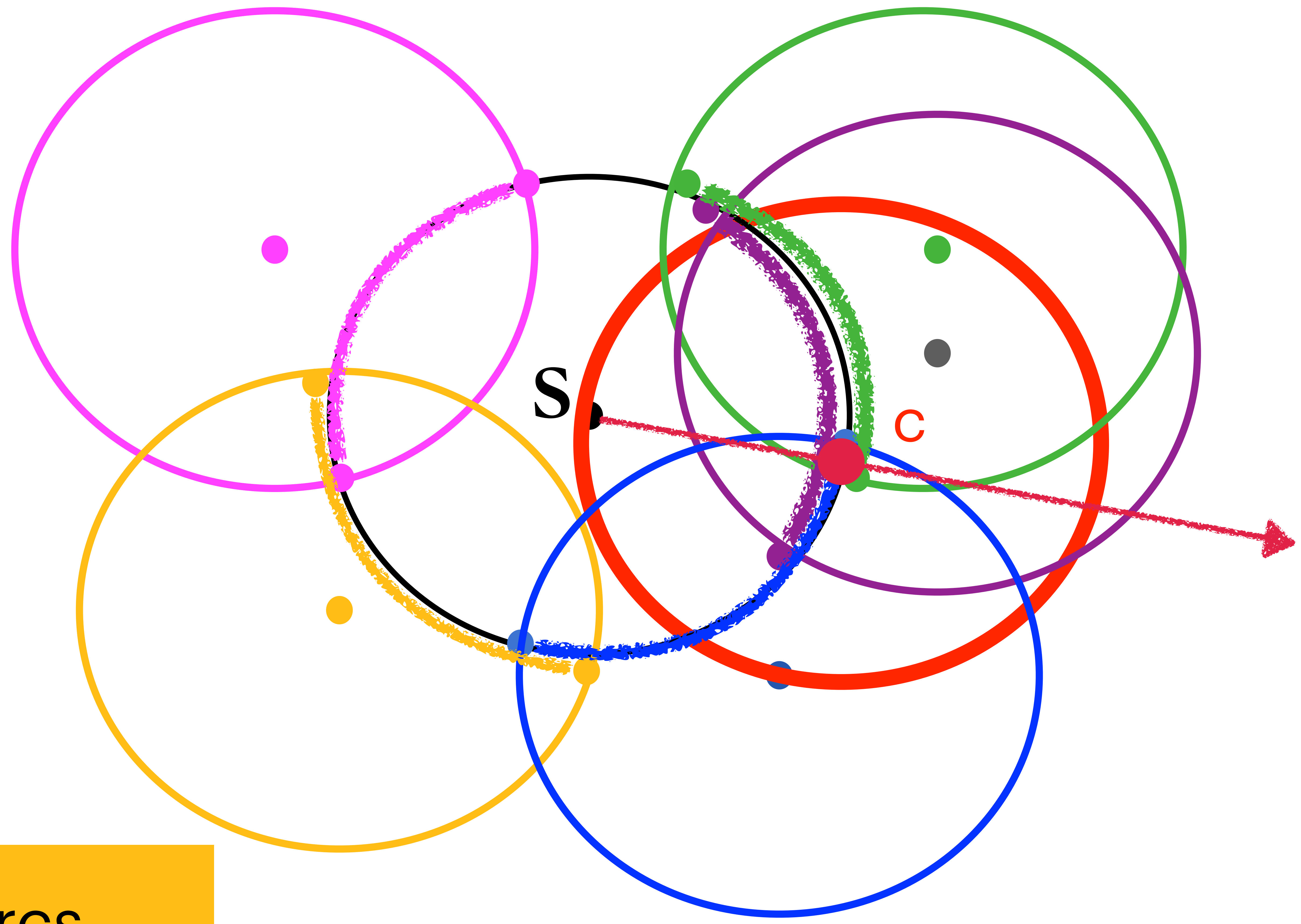
The Constrained Version (Cont.)

Lemma: The center c is any point of the arc that is the intersection of most arcs.



The Constrained Version (Cont.)

Lemma: The center c is any point of the arc that is the intersection of most arcs.

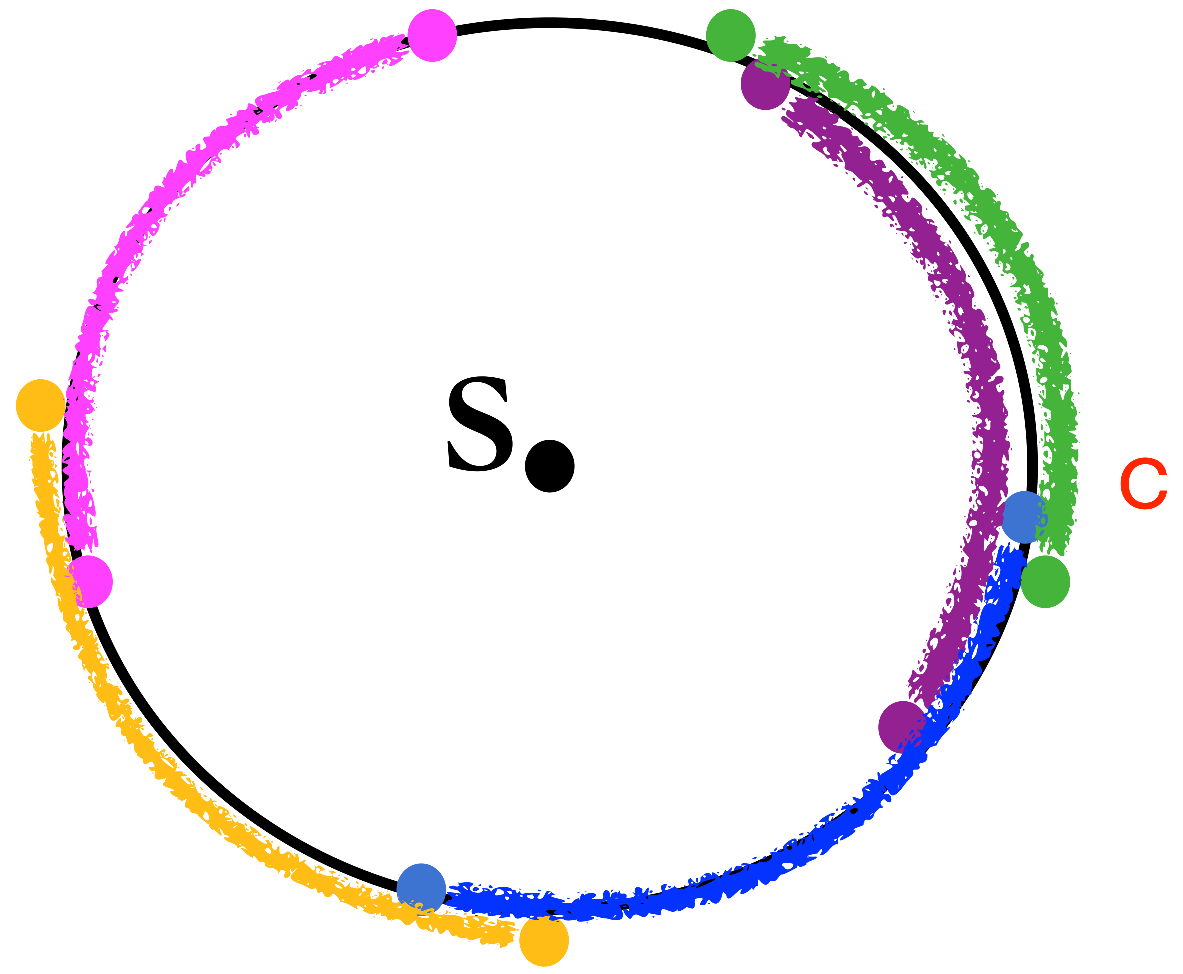


Center c is the point piercing most arcs.

Arc Piercing Problem

Input: n arcs on a cycle

Output: the point piercing most arcs.

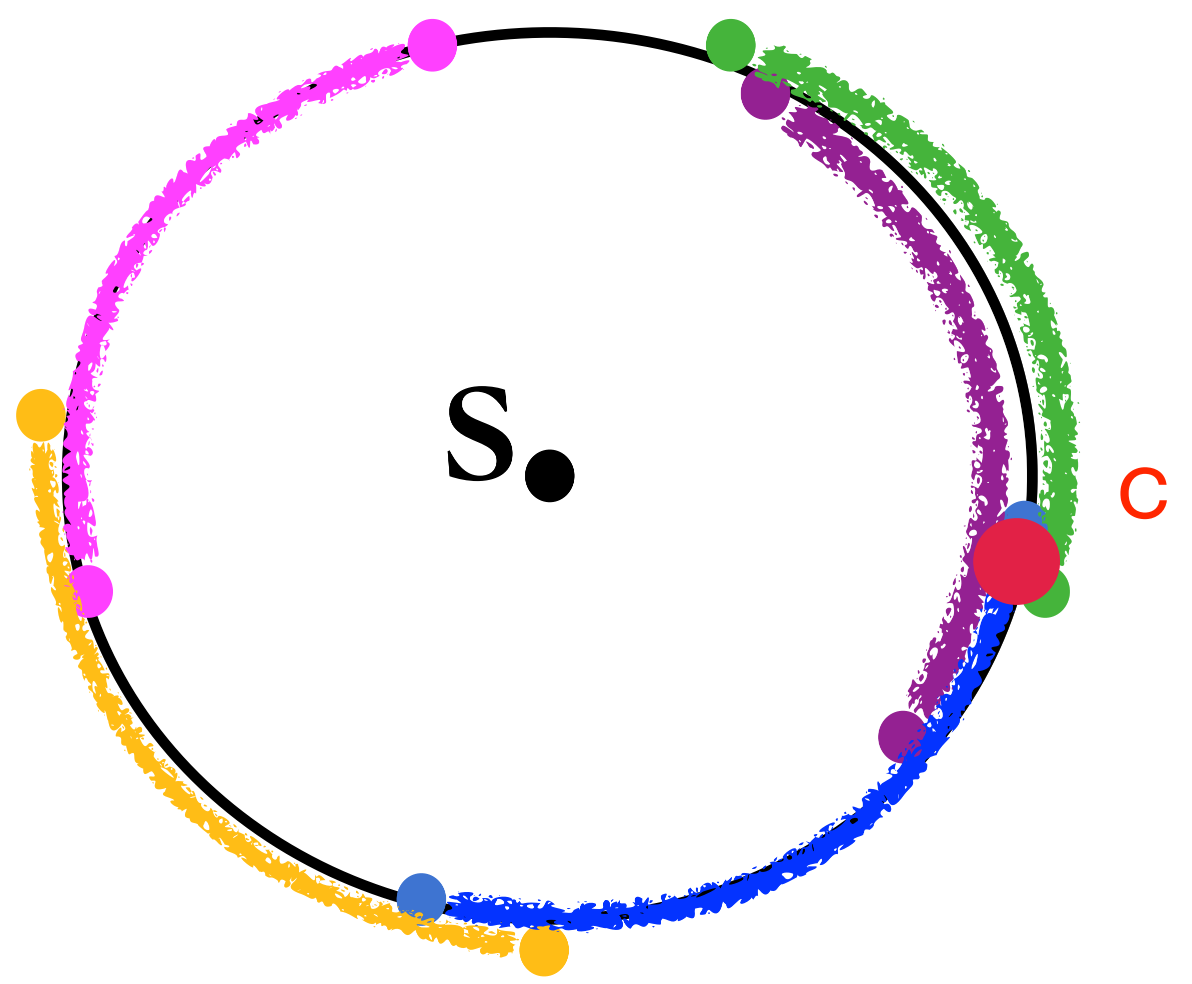


Center c is the point piercing most arcs.

Arc Piercing Problem

Input: n arcs on a cycle

Output: the point piercing most arcs.

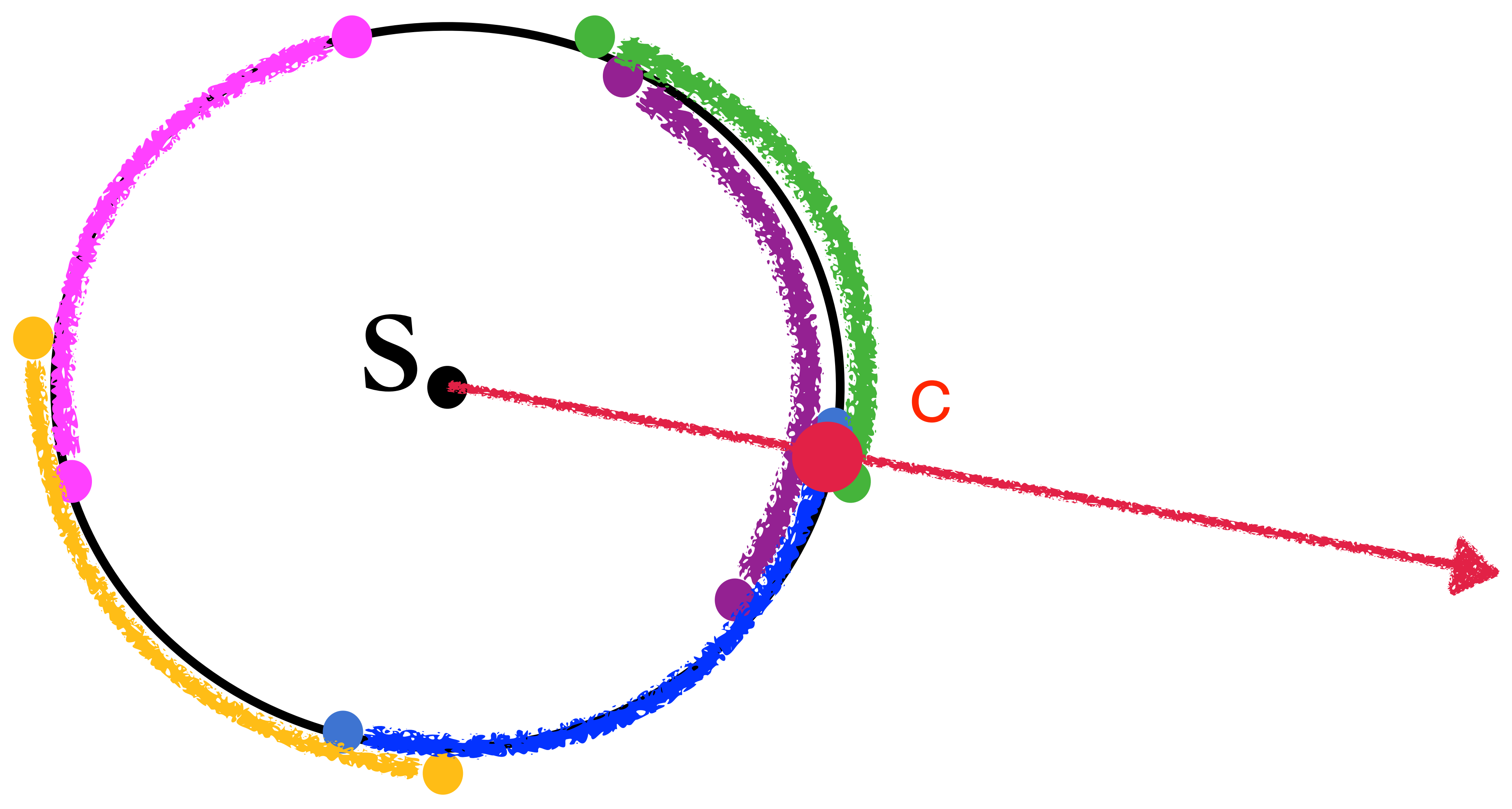


Center c is the point piercing most arcs.

Arc Piercing Problem

Input: n arcs on a cycle

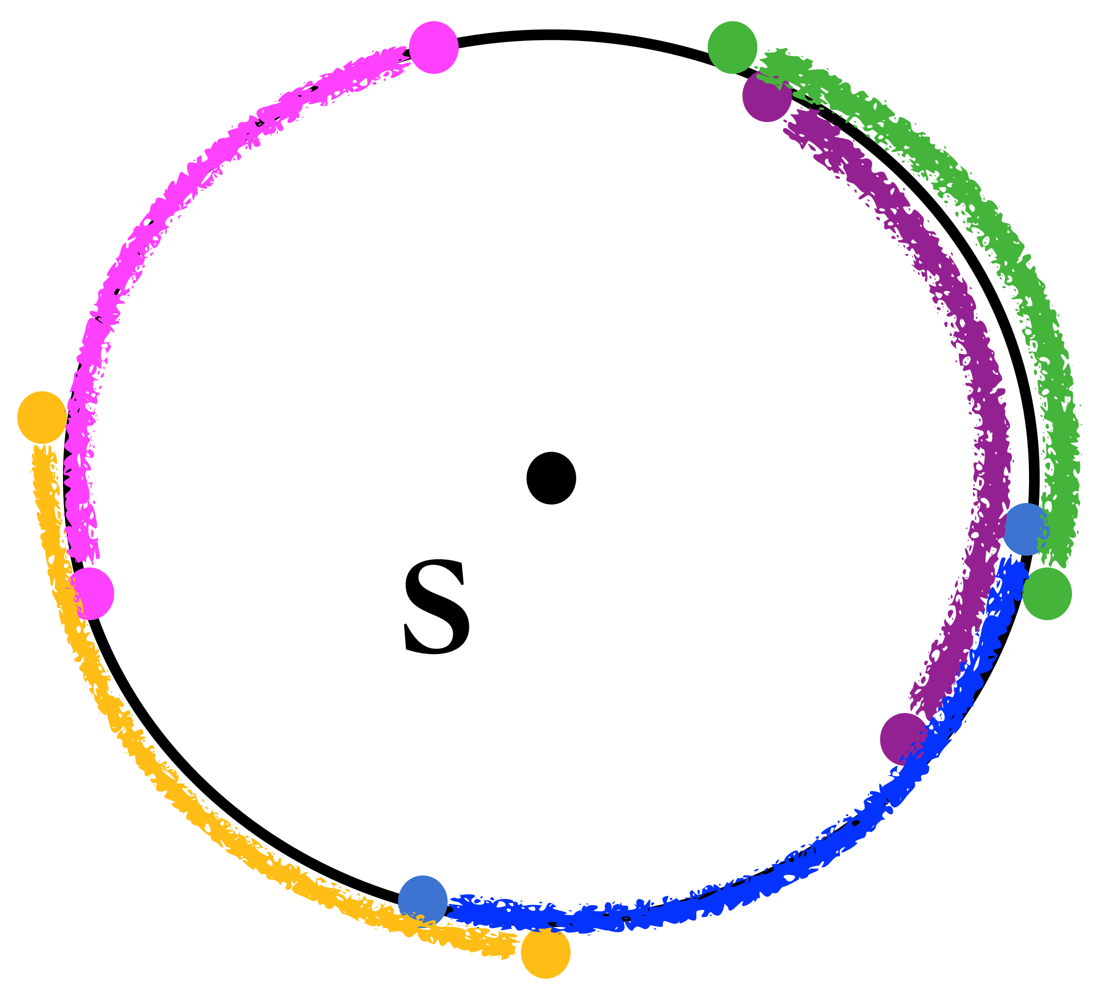
Output: the point piercing most arcs.



Center c is the point piercing most arcs.

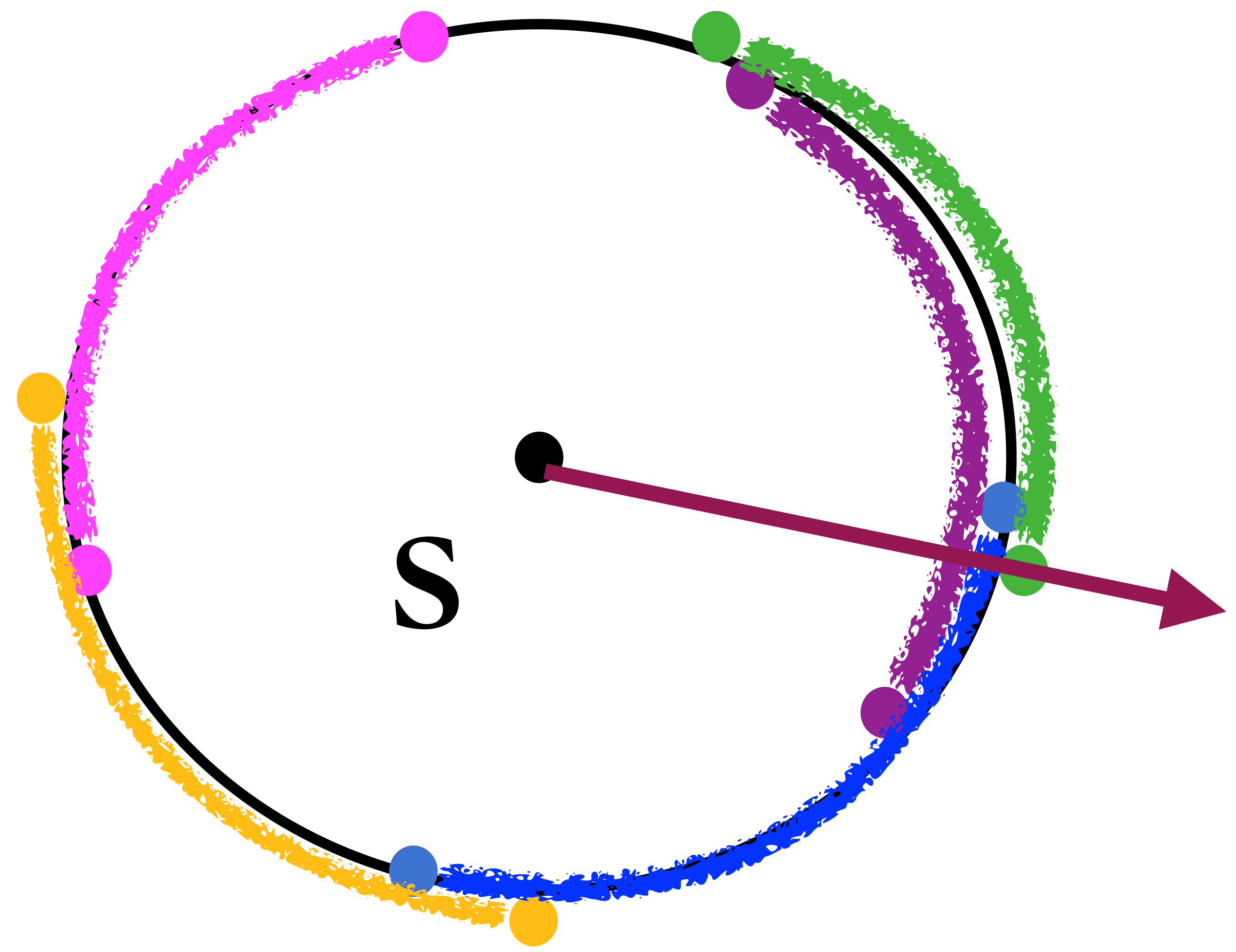
Straightforward Way for Arc Piercing Problem

Compute how many arcs are pierced by every endpoint.



Straightforward Way for Arc Piercing Problem

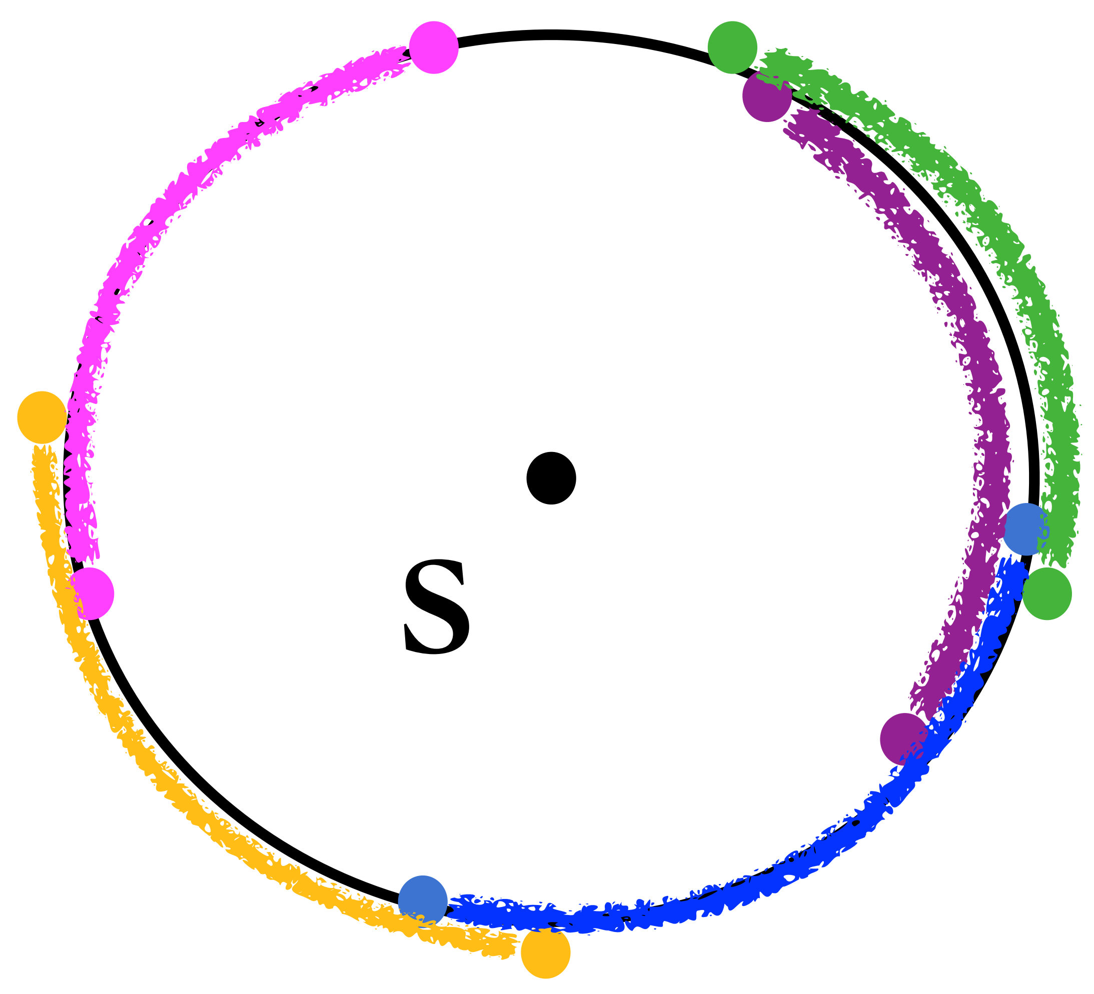
Compute how many arcs are pierced by every endpoint.



Piercing 3 arcs: MaxCount = 3

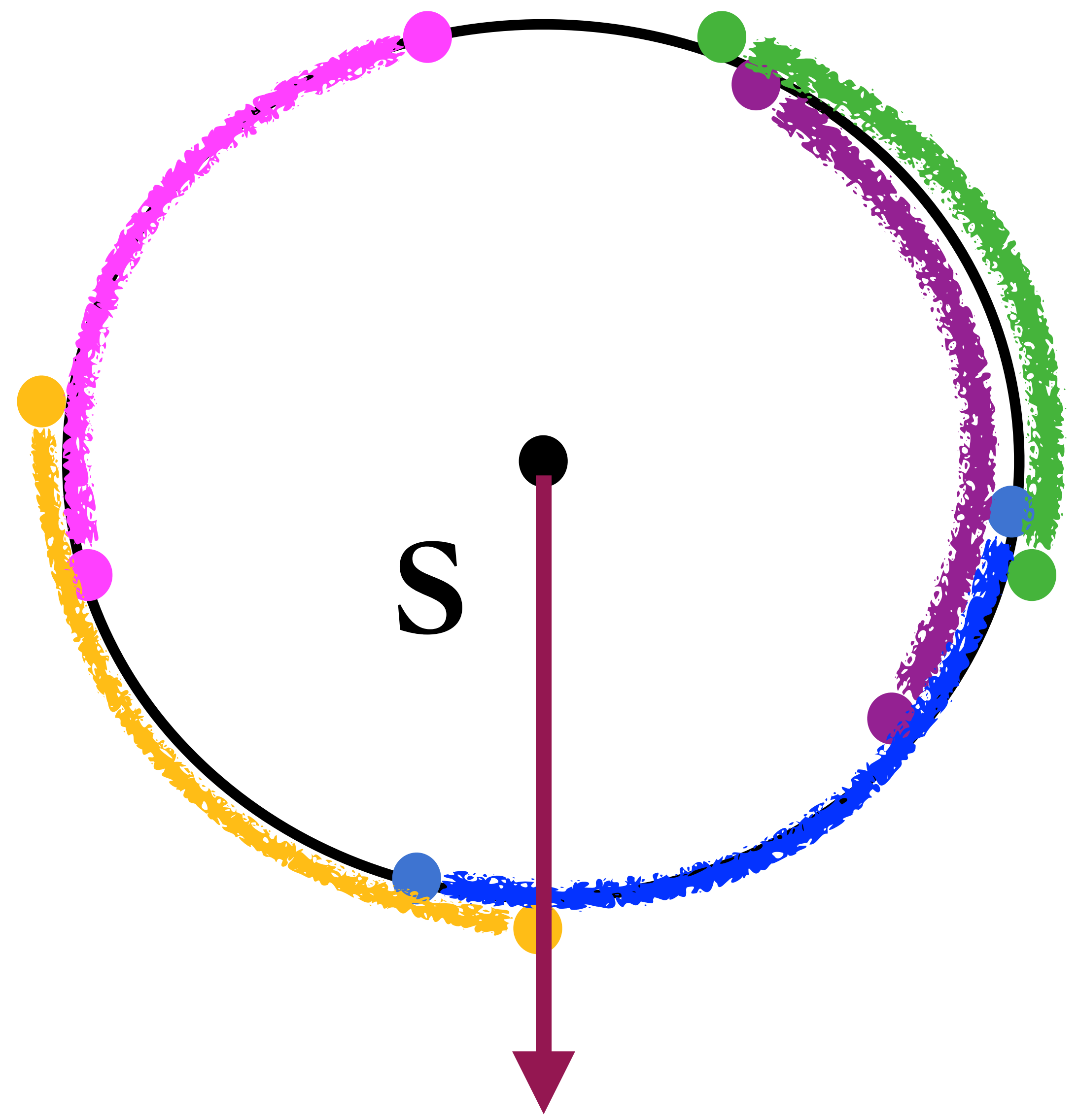
Straightforward Way for Arc Piercing Problem

Compute how many arcs are pierced by every endpoint.



Straightforward Way for Arc Piercing Problem

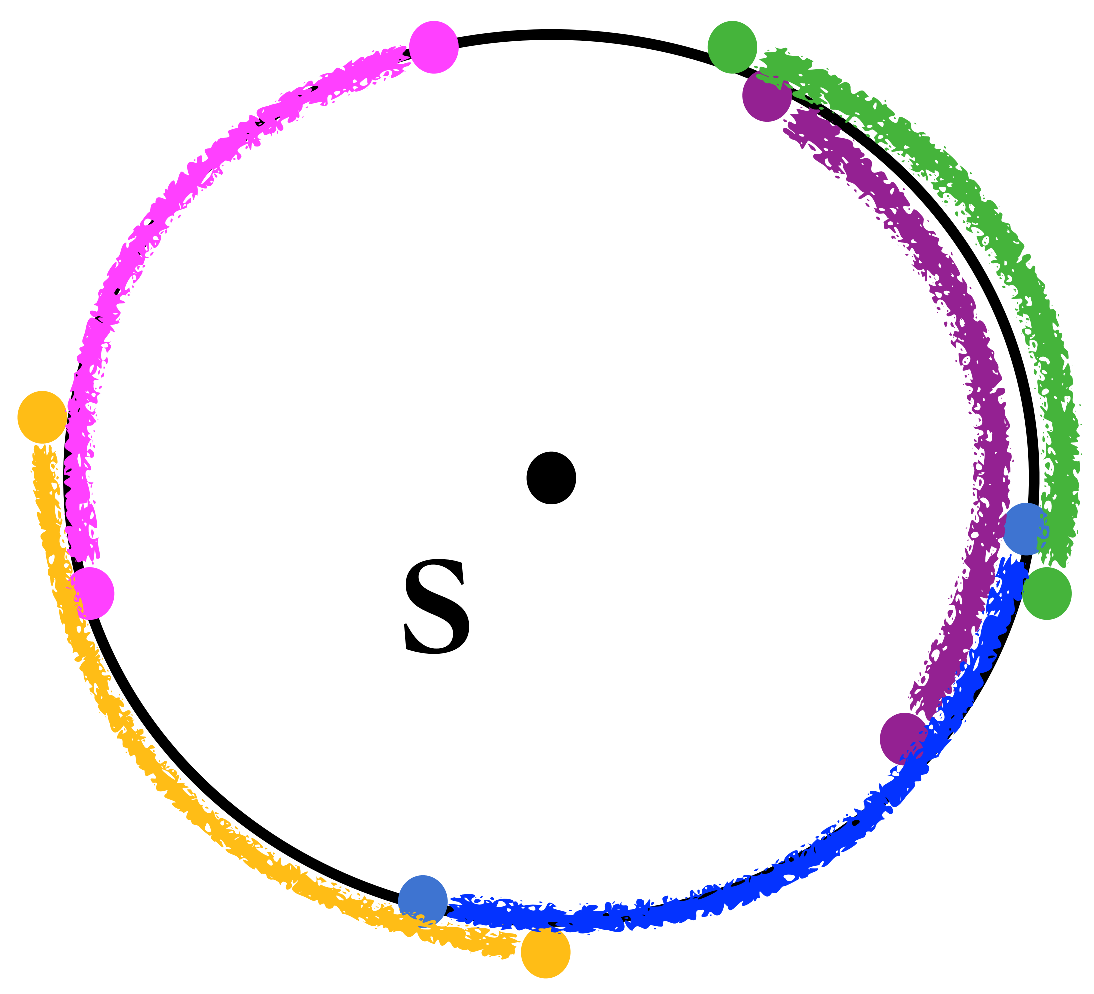
Compute how many arcs are pierced by every endpoint.



Piercing 2 arcs: MaxCount = 3

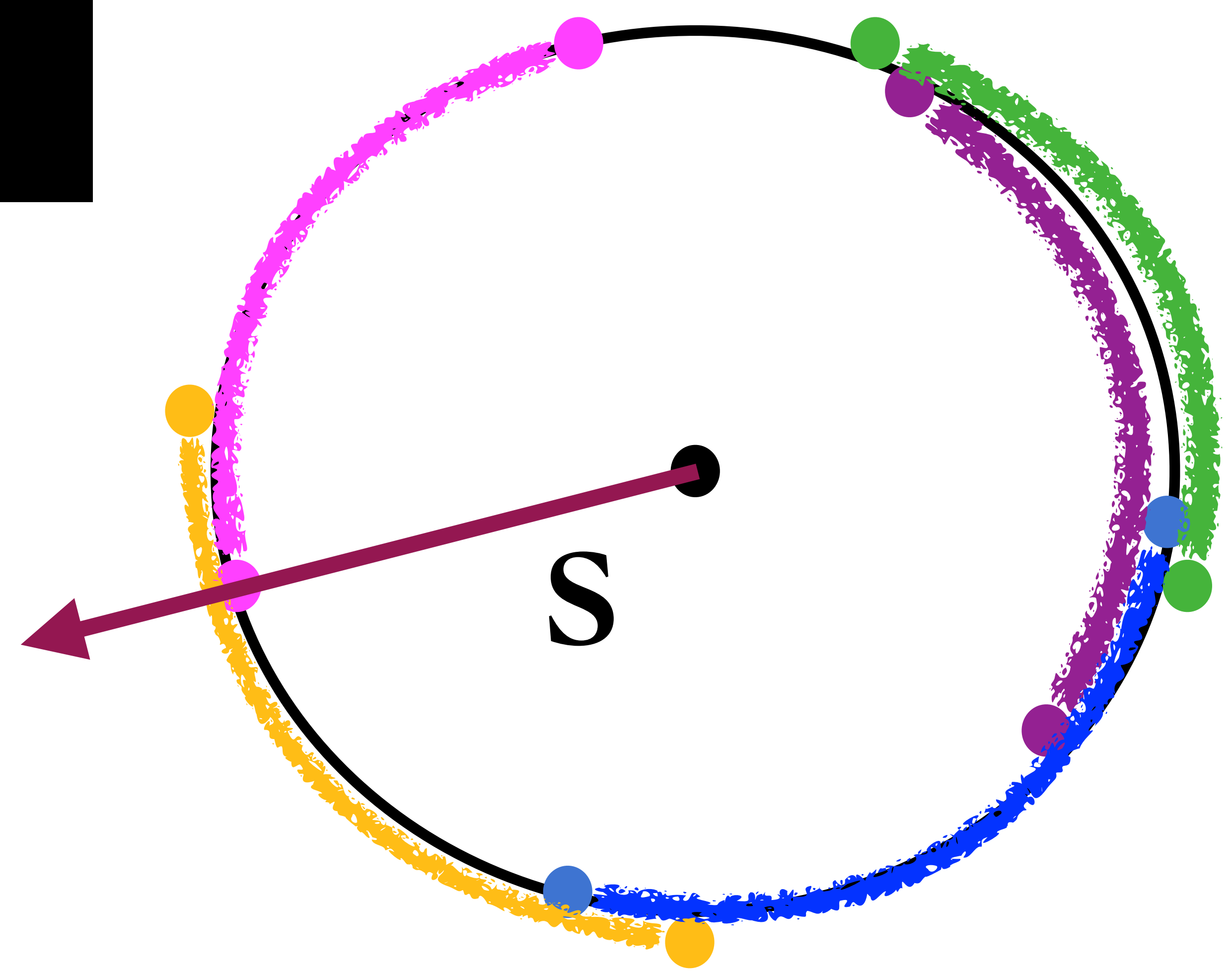
Straightforward Way for Arc Piercing Problem

Compute how many arcs are pierced by every endpoint.



Straightforward Way for Arc Piercing Problem

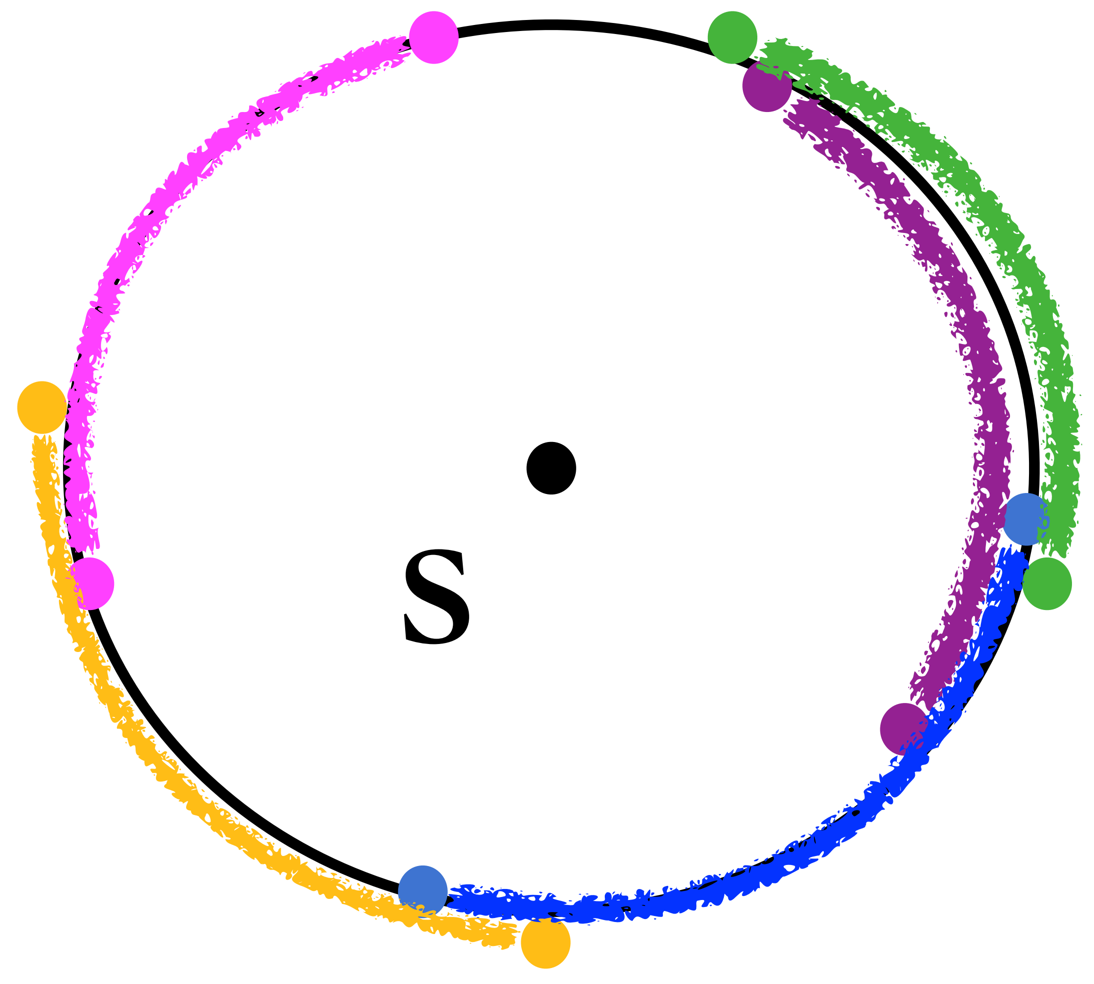
Compute how many arcs are pierced by every endpoint.



Piercing 2 arcs: MaxCount = 3

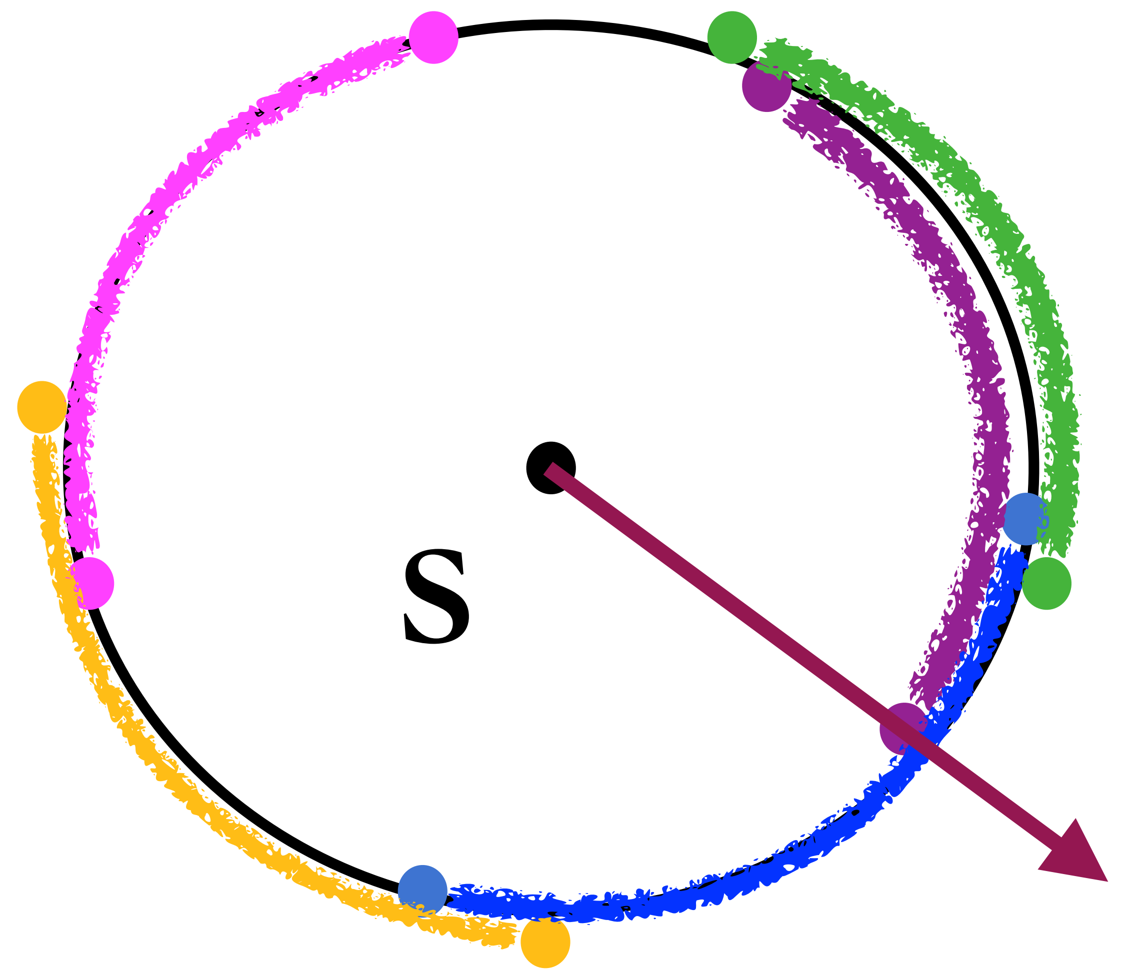
Straightforward Way for Arc Piercing Problem

Compute how many arcs are pierced by every endpoint.



Straightforward Way for Arc Piercing Problem

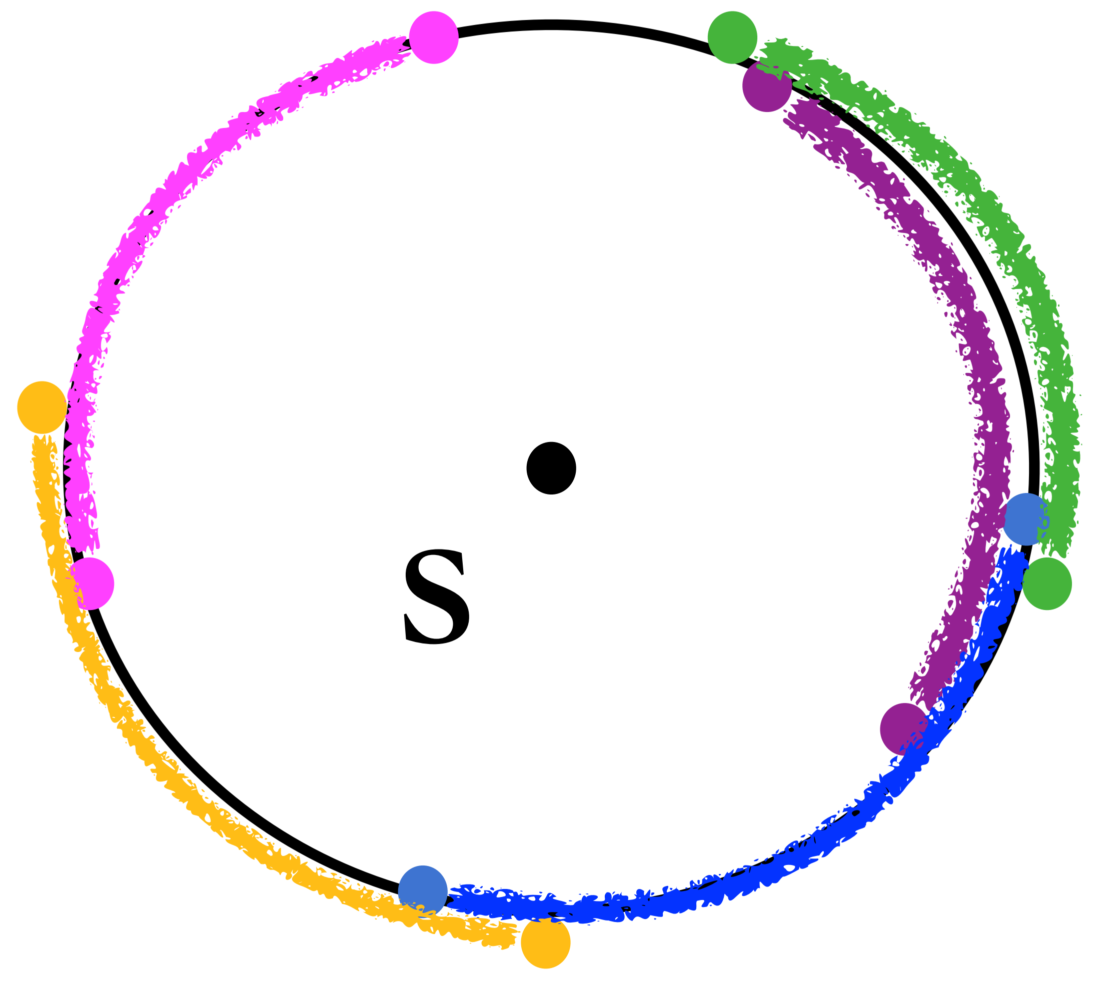
Compute how many arcs are pierced by every endpoint.



Piercing 2 arcs: MaxCount = 3

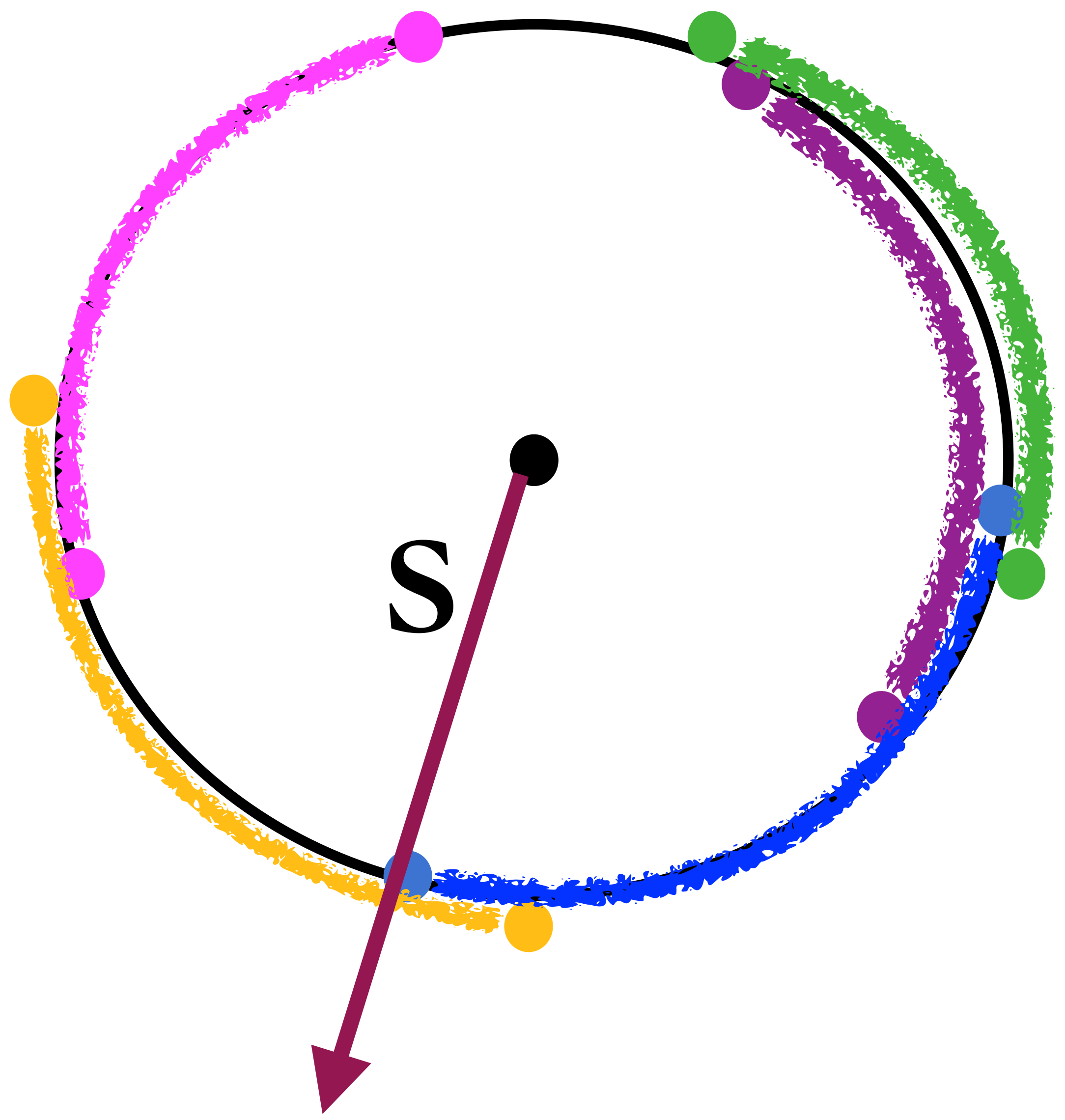
Straightforward Way for Arc Piercing Problem

Compute how many arcs are pierced by every endpoint.



Straightforward Way for Arc Piercing Problem

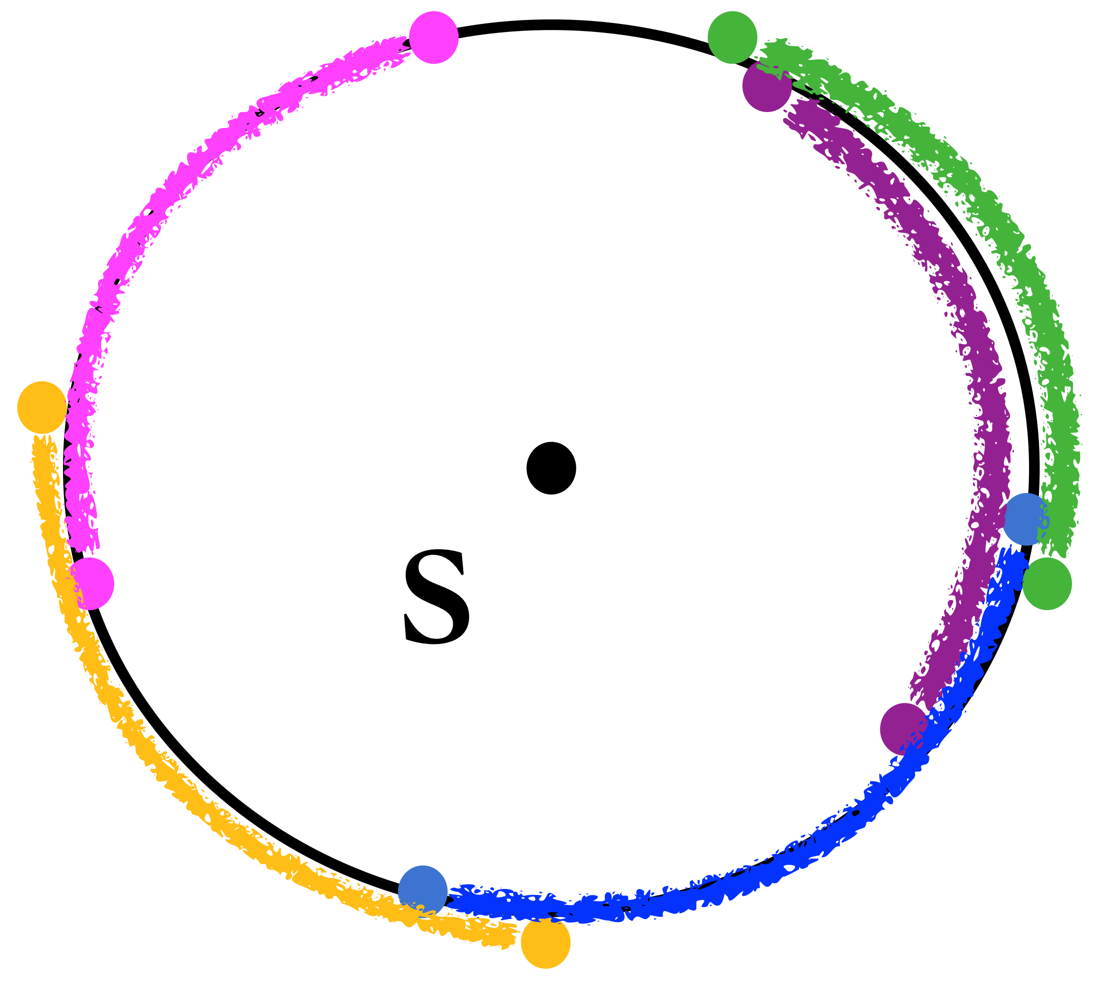
Compute how many arcs are pierced by every endpoint.



Piercing 2 arcs: MaxCount = 3

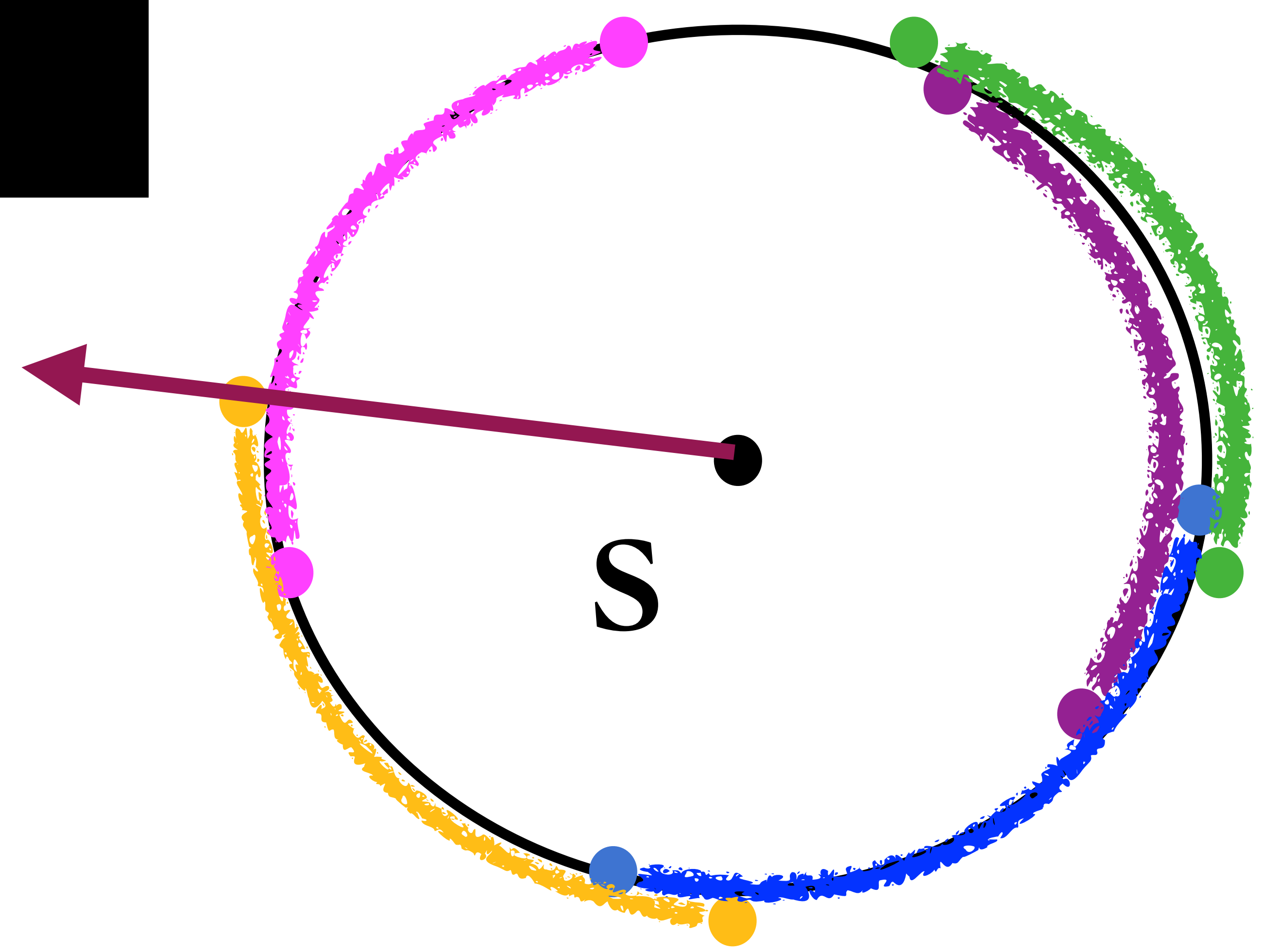
Straightforward Way for Arc Piercing Problem

Compute how many arcs are pierced by every endpoint.



Straightforward Way for Arc Piercing Problem

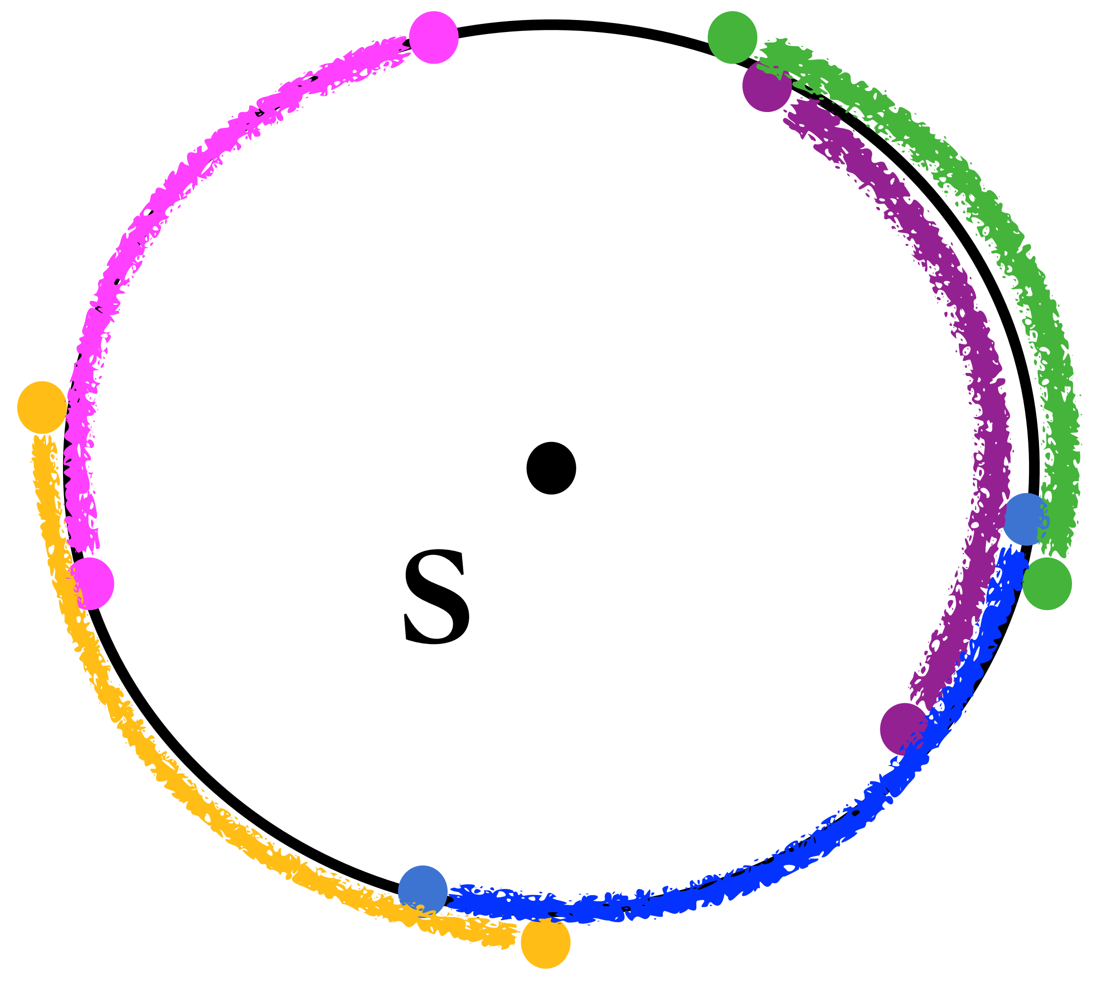
Compute how many arcs are pierced by every endpoint.



Piercing 2 arcs: MaxCount = 3

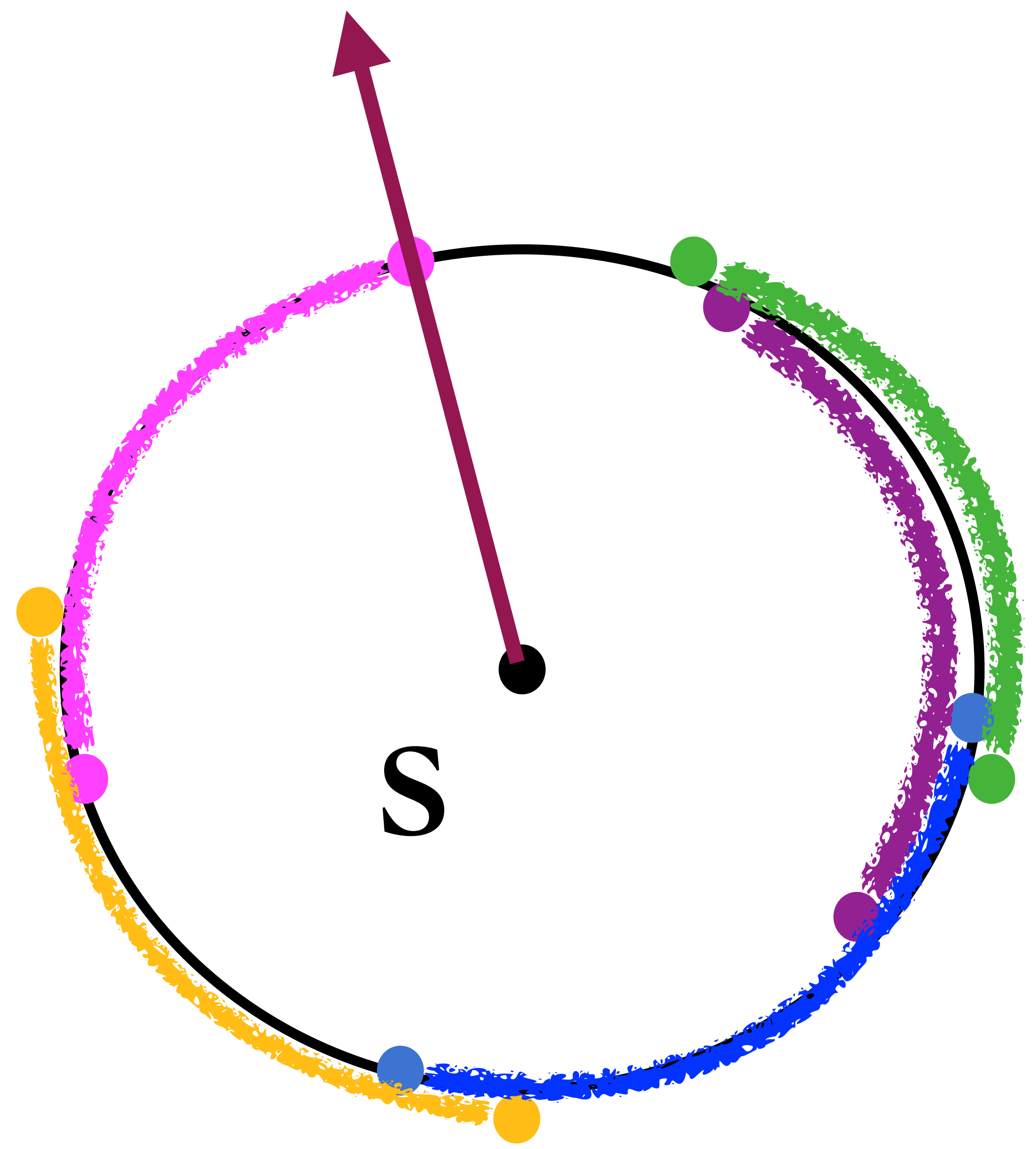
Straightforward Way for Arc Piercing Problem

Compute how many arcs are pierced by every endpoint.



Straightforward Way for Arc Piercing Problem

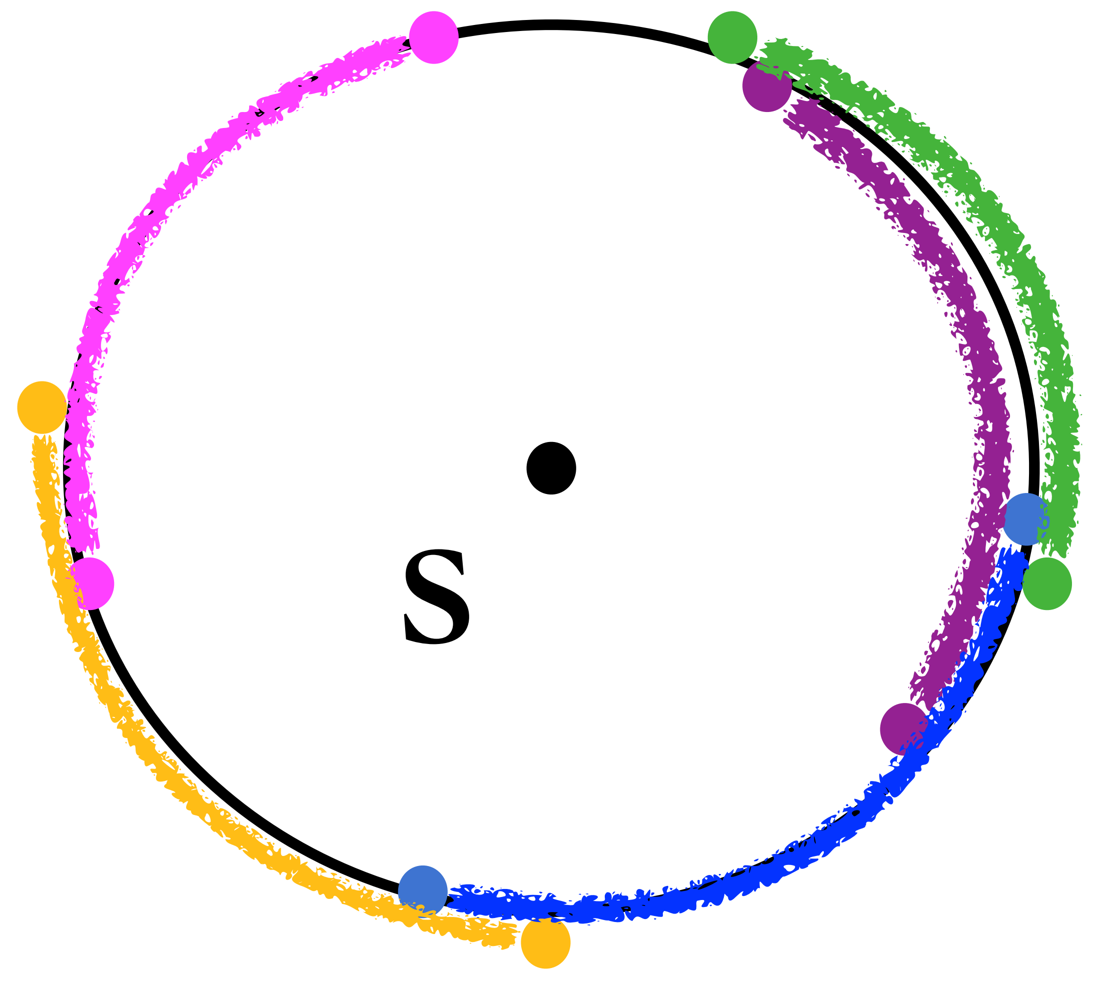
Compute how many arcs are pierced by every endpoint.



Piercing 1 arcs: MaxCount = 3

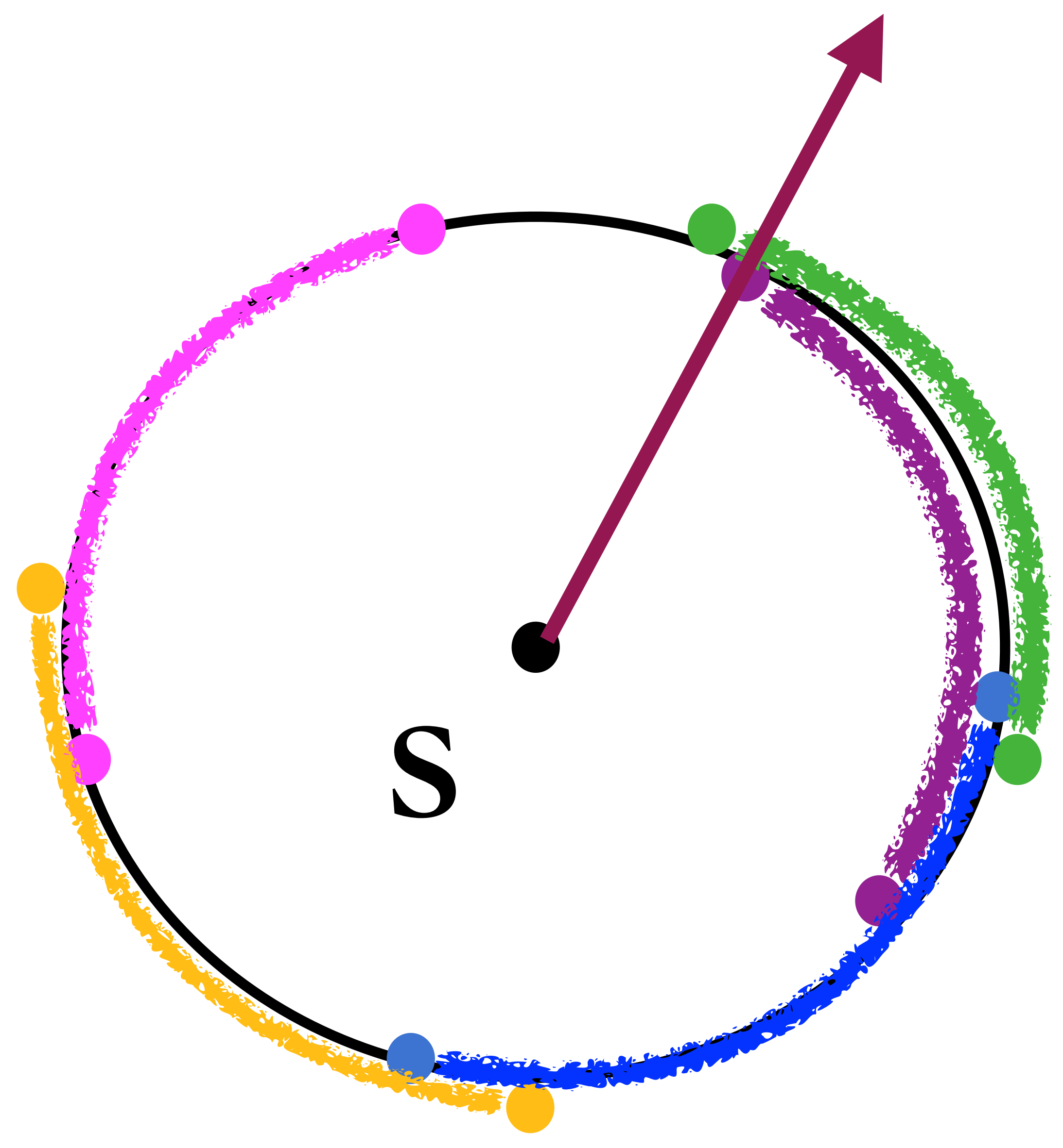
Straightforward Way for Arc Piercing Problem

Compute how many arcs are pierced by every endpoint.



Straightforward Way for Arc Piercing Problem

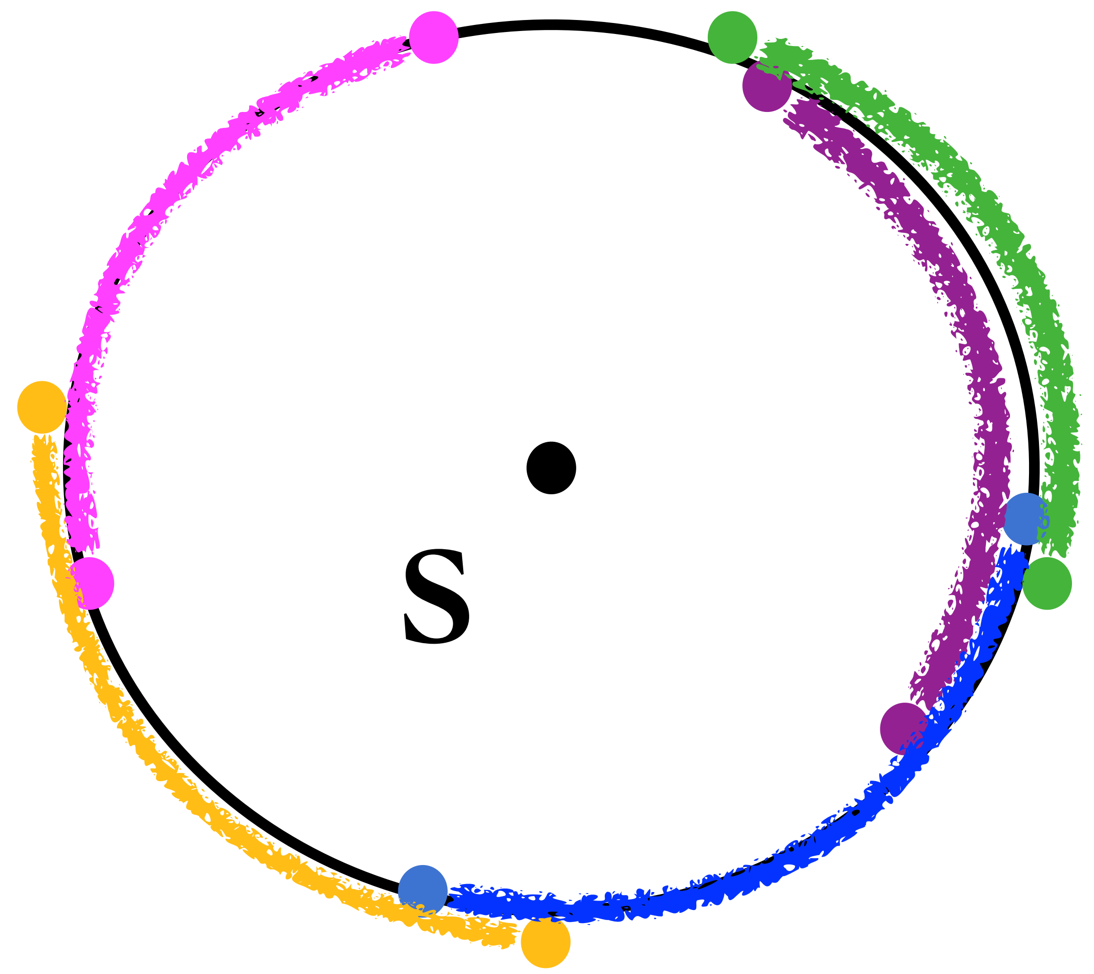
Compute how many arcs are pierced by every endpoint.



Piercing 2 arcs: MaxCount = 3

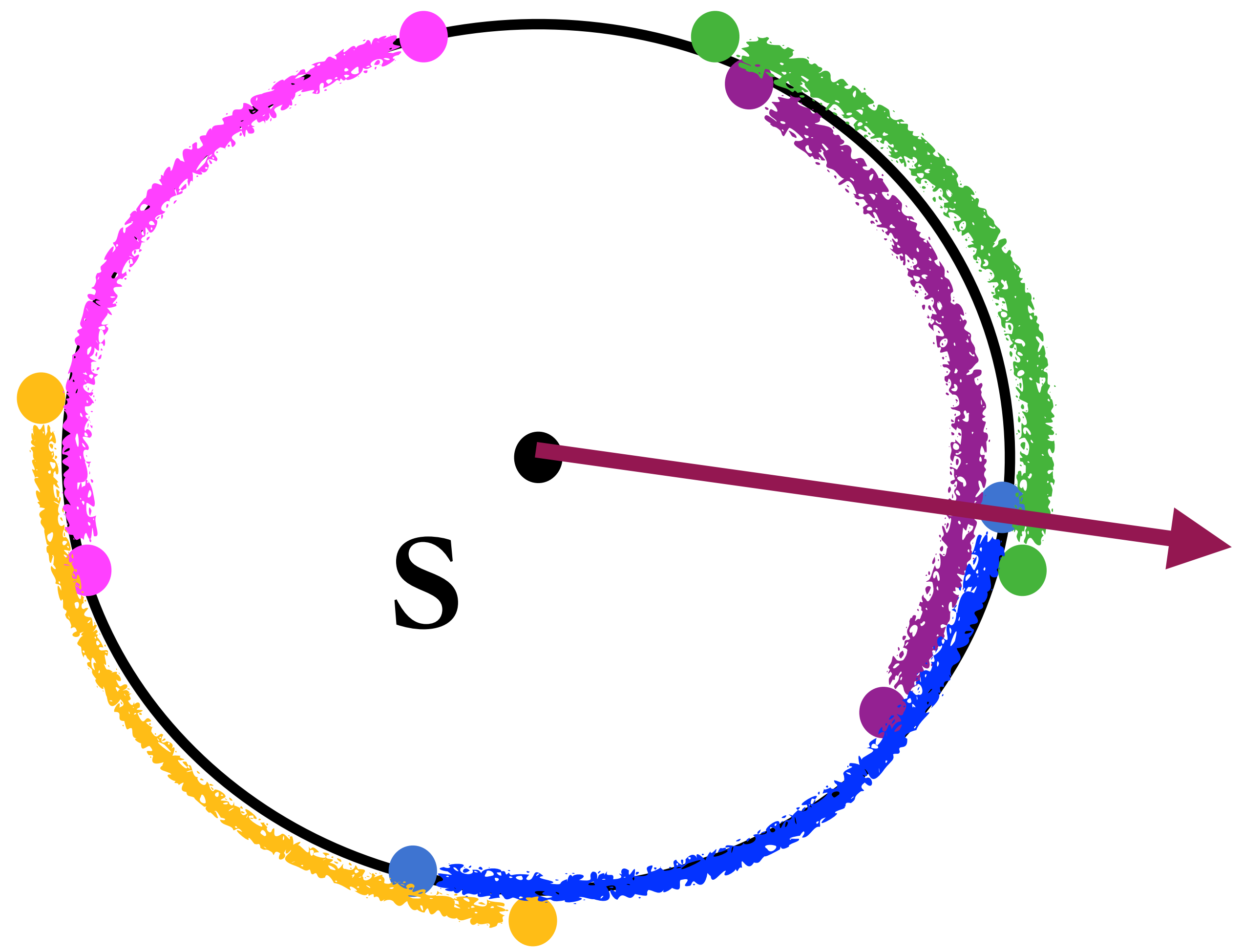
Straightforward Way for Arc Piercing Problem

Compute how many arcs are pierced by every endpoint.



Straightforward Way for Arc Piercing Problem

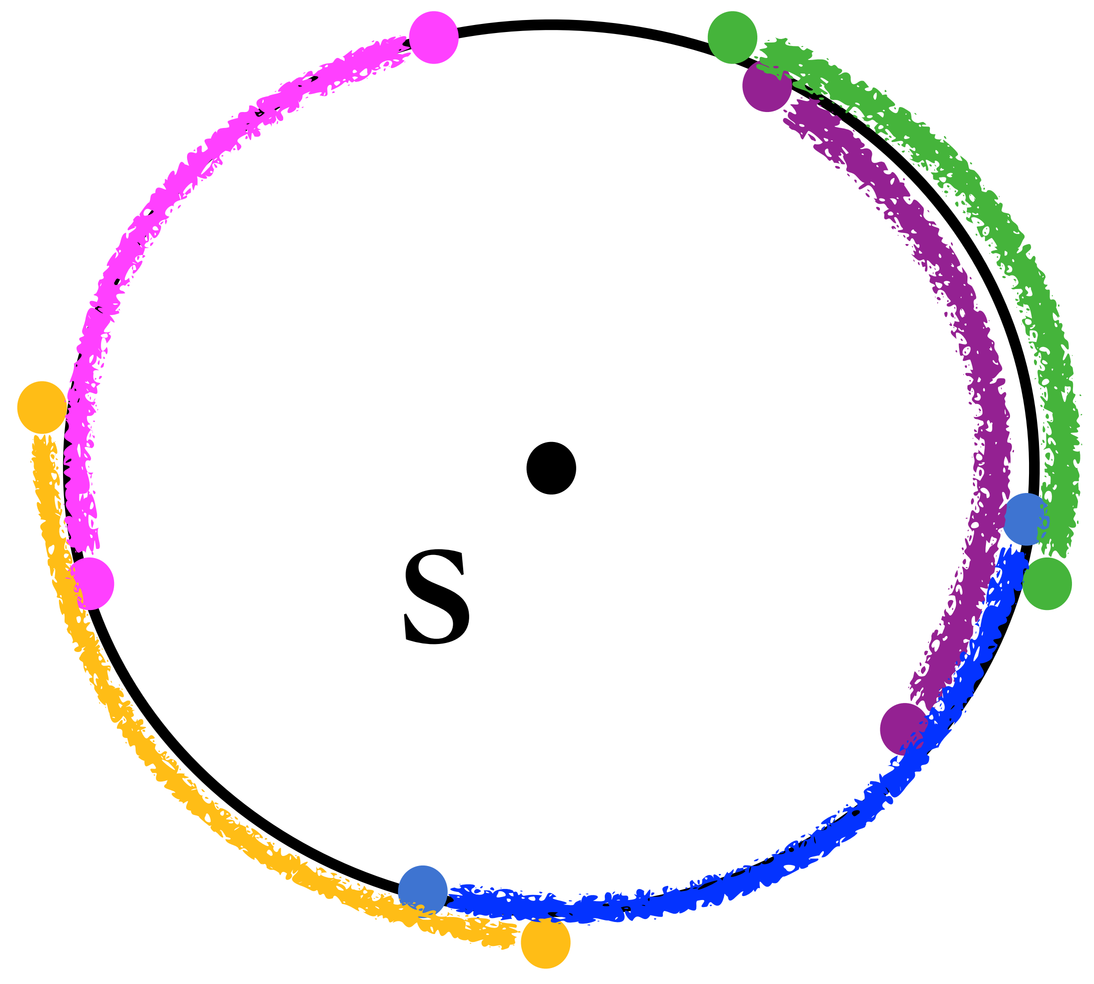
Compute how many arcs are pierced by every endpoint.



Piercing 3 arcs: MaxCount = 3

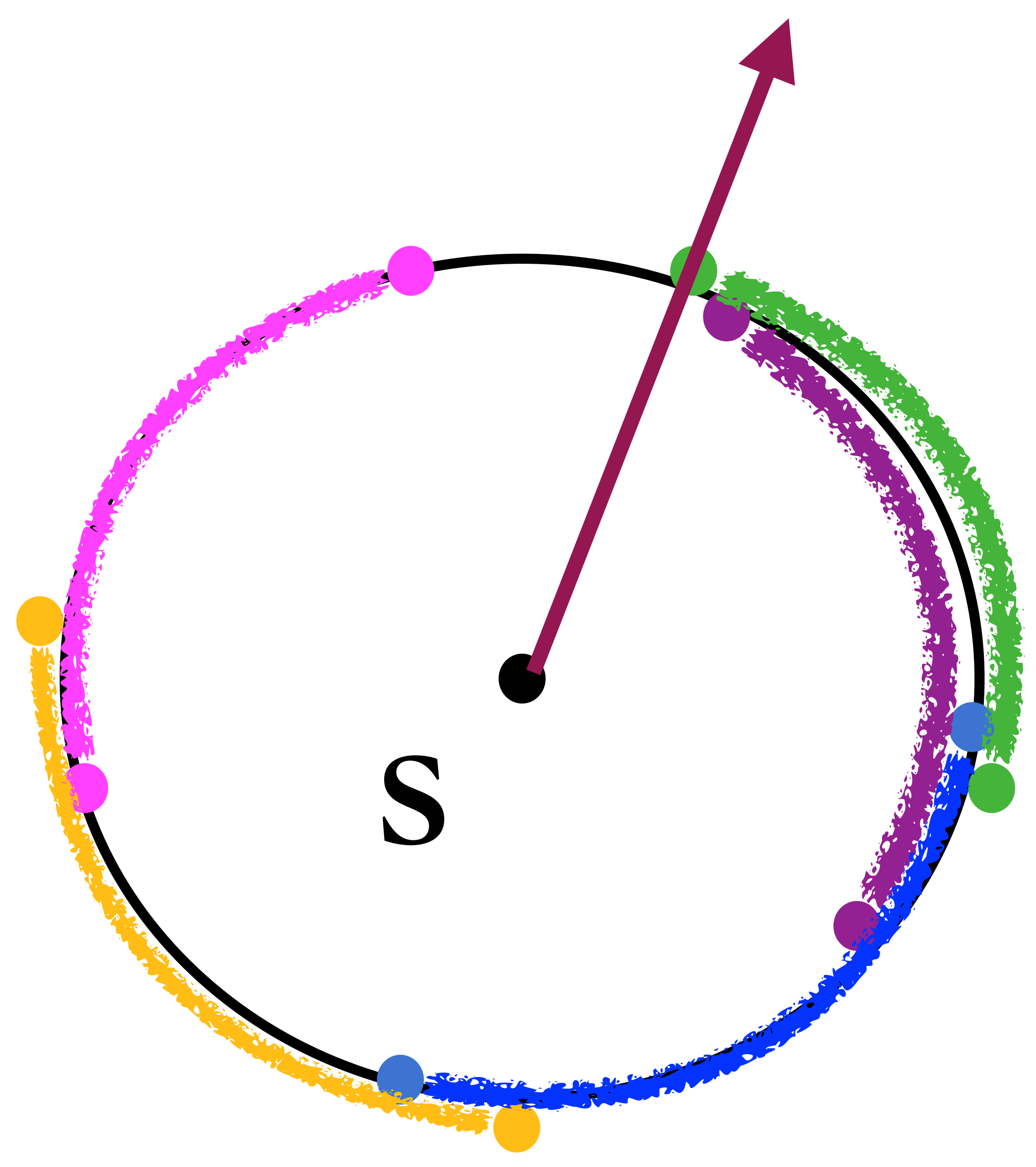
Straightforward Way for Arc Piercing Problem

Compute how many arcs are pierced by every endpoint.



Straightforward Way for Arc Piercing Problem

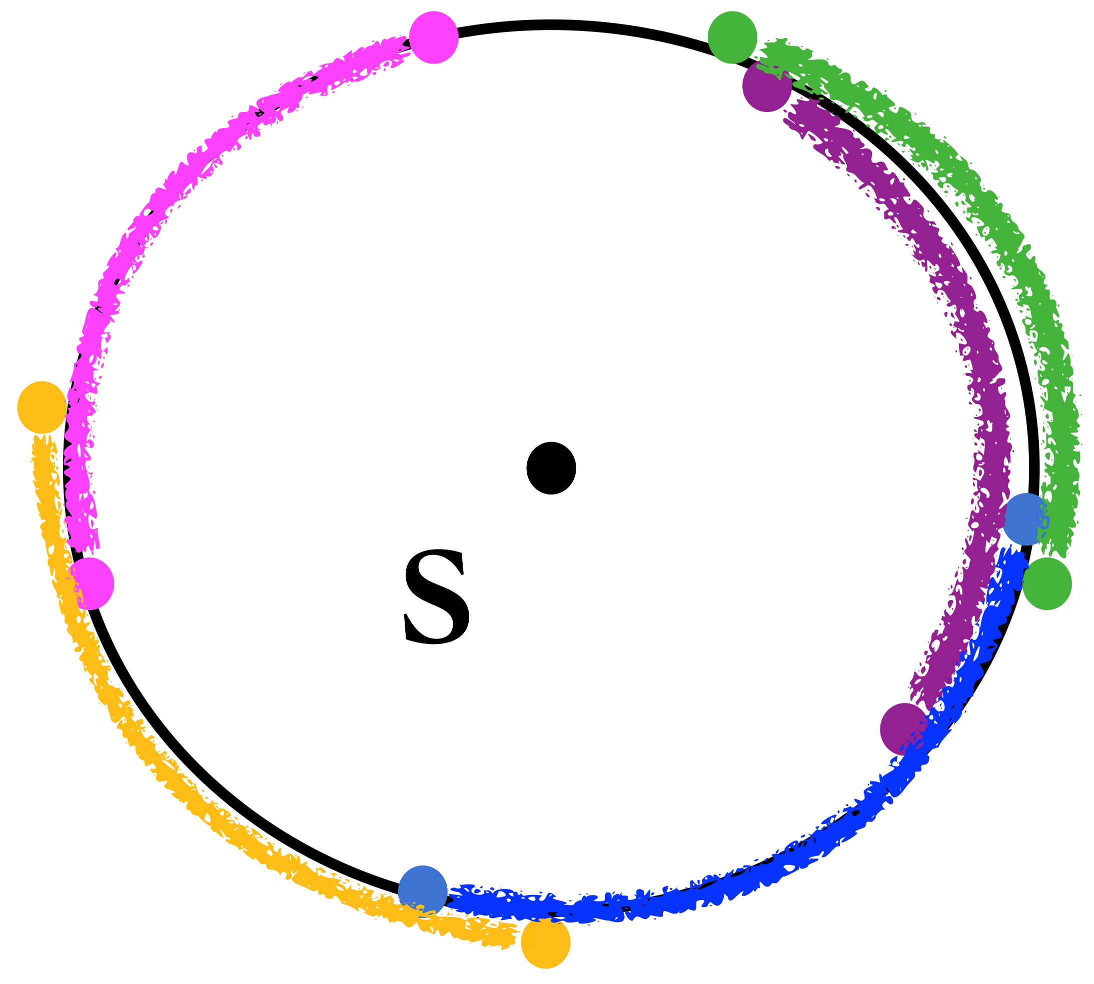
Compute how many arcs are pierced by every endpoint.



Piercing 1 arcs: MaxCount = 3

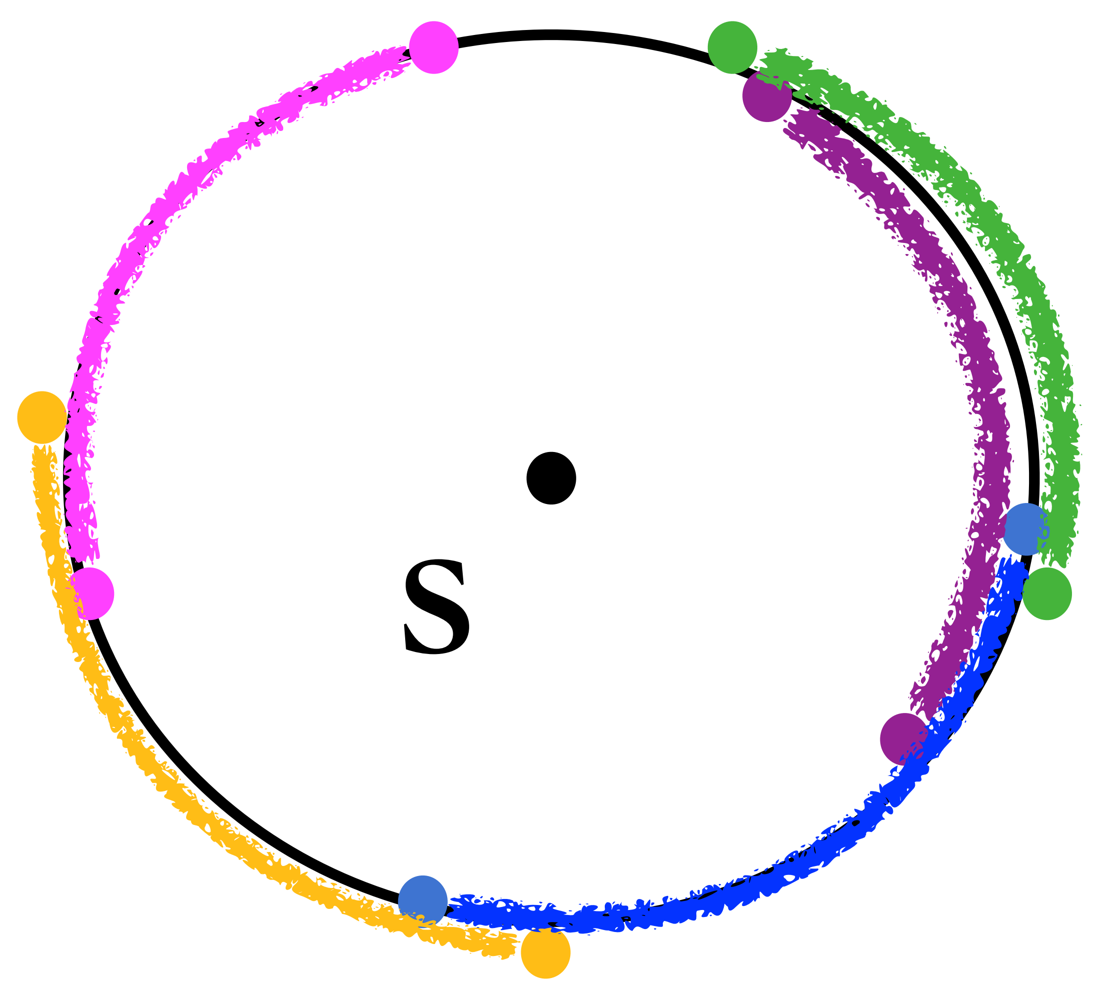
Straightforward Way for Arc Piercing Problem

Compute how many arcs are pierced by every endpoint.



Straightforward Way for Arc Piercing Problem

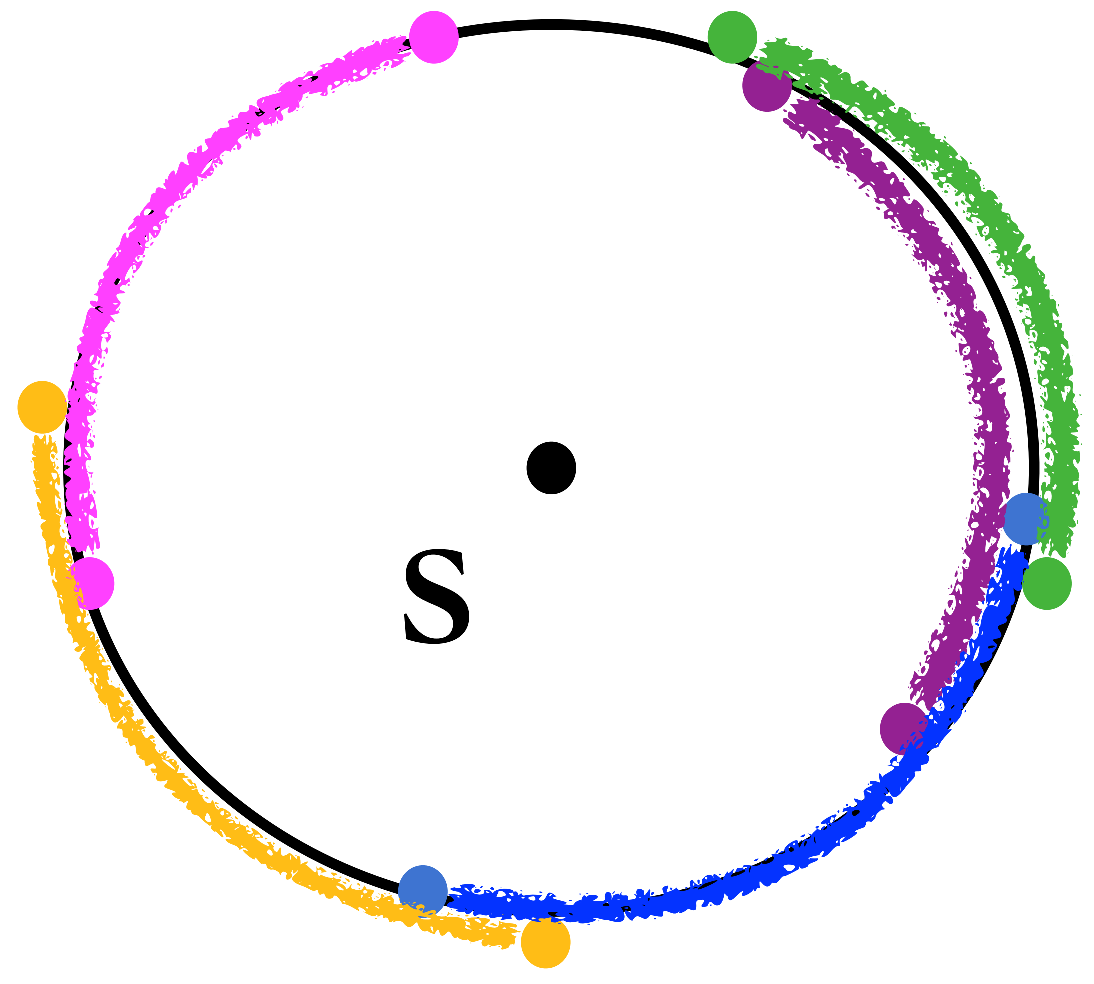
Compute how many arcs are pierced by every endpoint.



Time Complexity: $O(n^2)$

Straightforward Way for Arc Piercing Problem

Compute how many arcs are pierced by every endpoint.



Improve $O(n^2)$ to $O(n\log n)$: Computing # of arcs pierced by a point in $O(1)$ time.

Time Complexity: $O(n^2)$

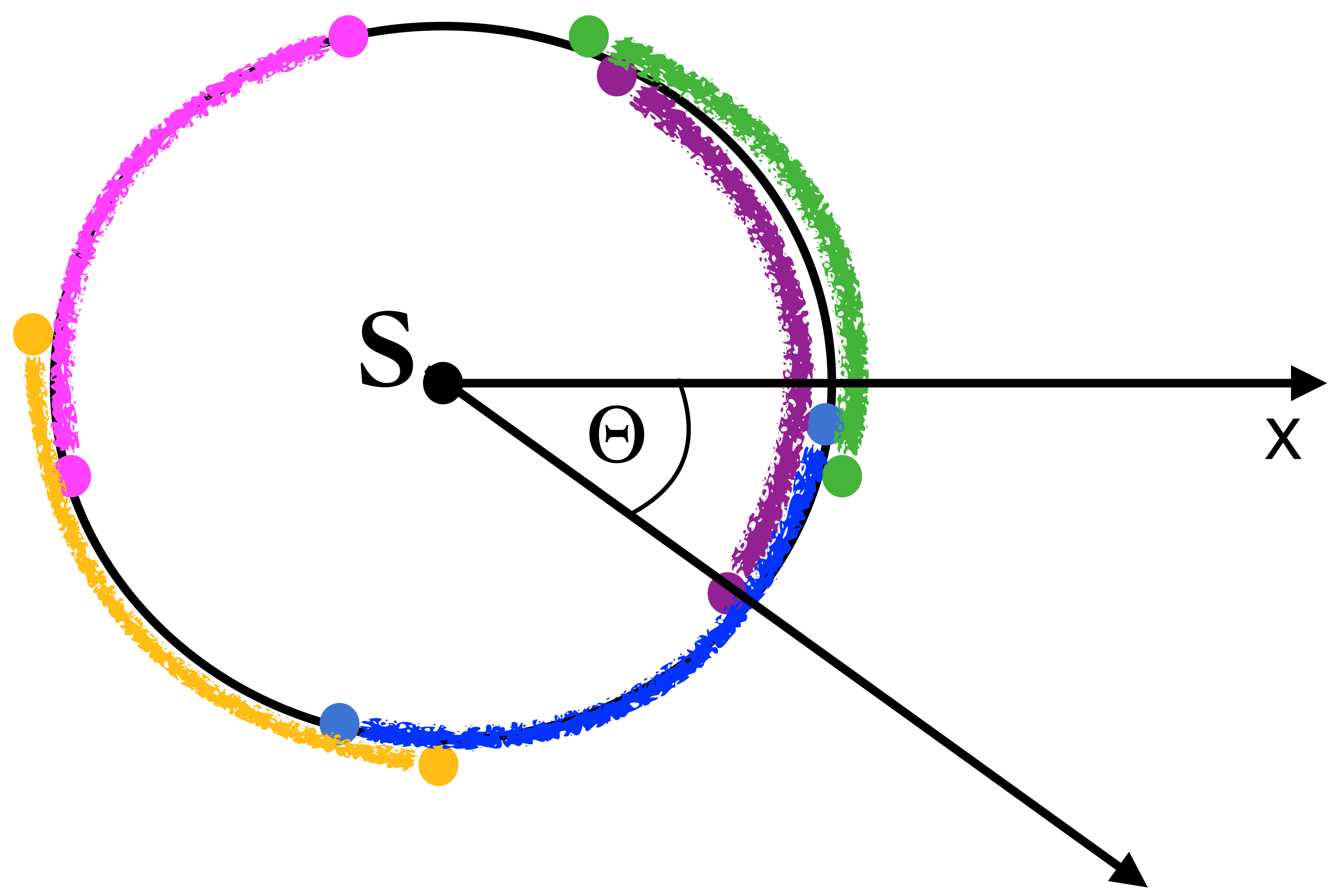
The Preprocessing Work

The O(nlog n) Preprocessing Work:

1. Sort all endpoints of arcs in clockwise order;

2. Mark Entry and Exit Endpoints of every arc in clockwise order;

3. Set MaxCount = 0 and count = 1.



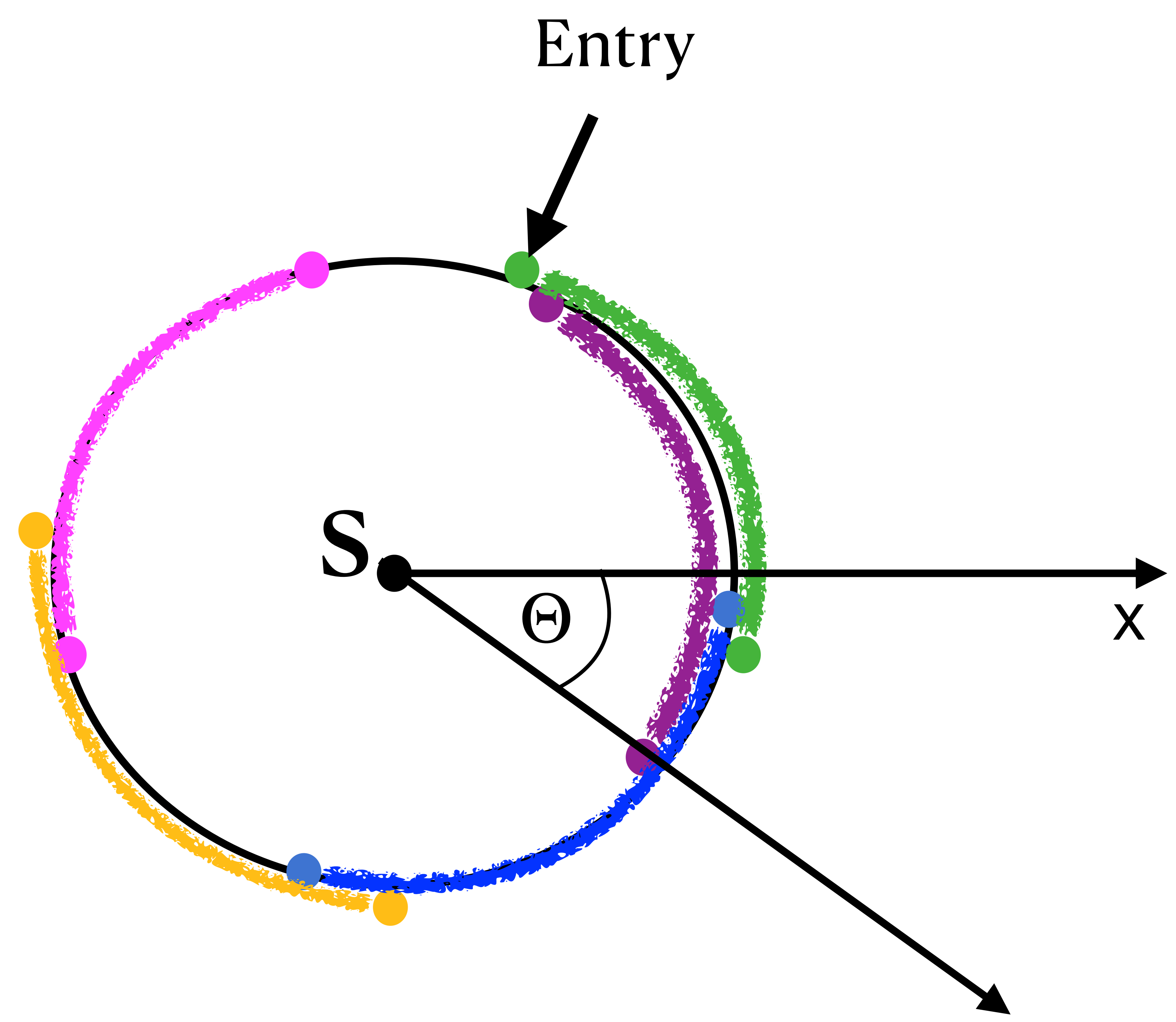
The Preprocessing Work

The O(nlog n) Preprocessing Work:

1. Sort all endpoints of arcs in clockwise order;

2. Mark Entry and Exit Endpoints of every arc in clockwise order;

3. Set MaxCount = 0 and count = 1.



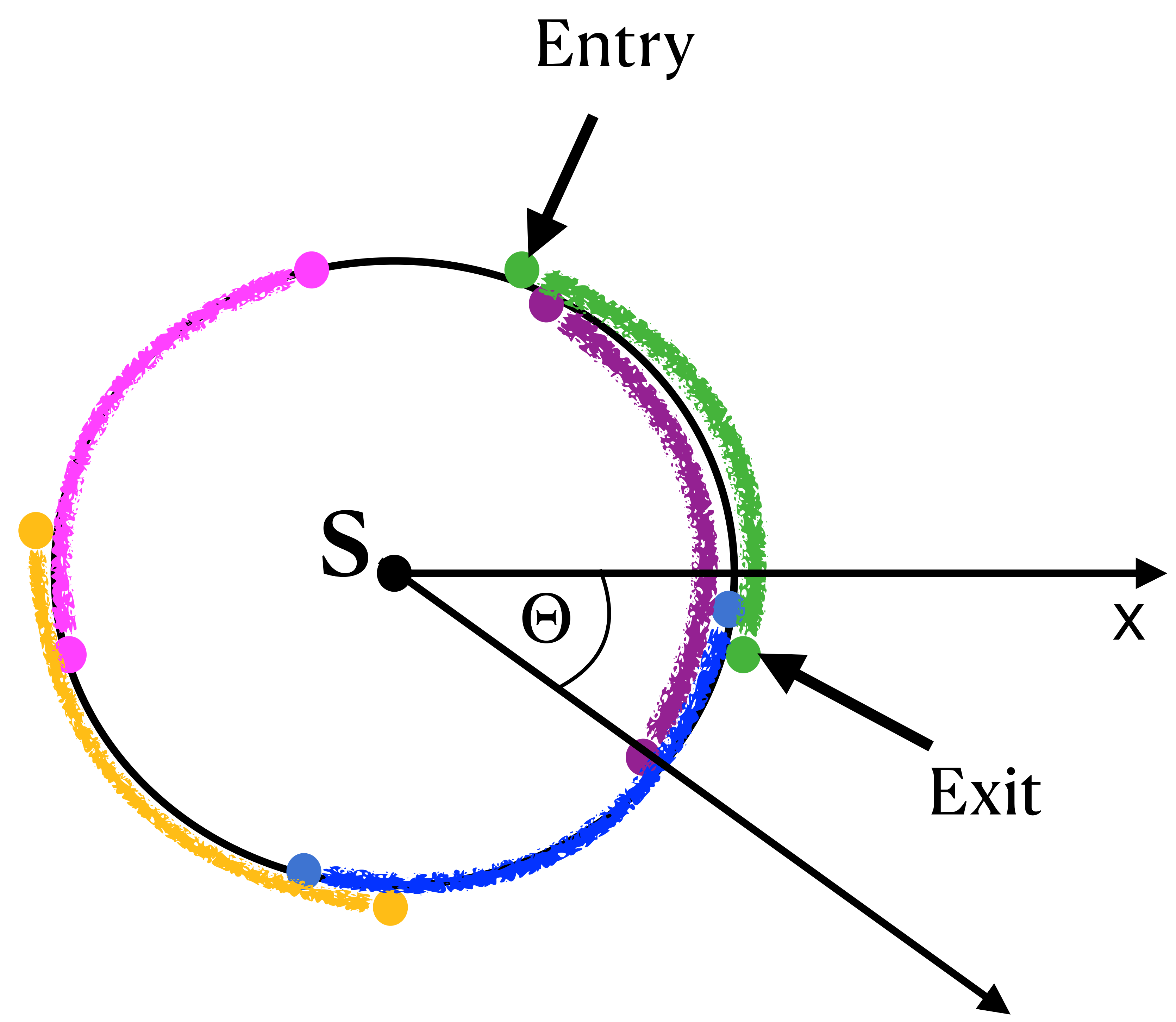
The Preprocessing Work

The O(nlog n) Preprocessing Work:

1. Sort all endpoints of arcs in clockwise order;

2. Mark Entry and Exit Endpoints of every arc in clockwise order;

3. Set MaxCount = 0 and count = 1.

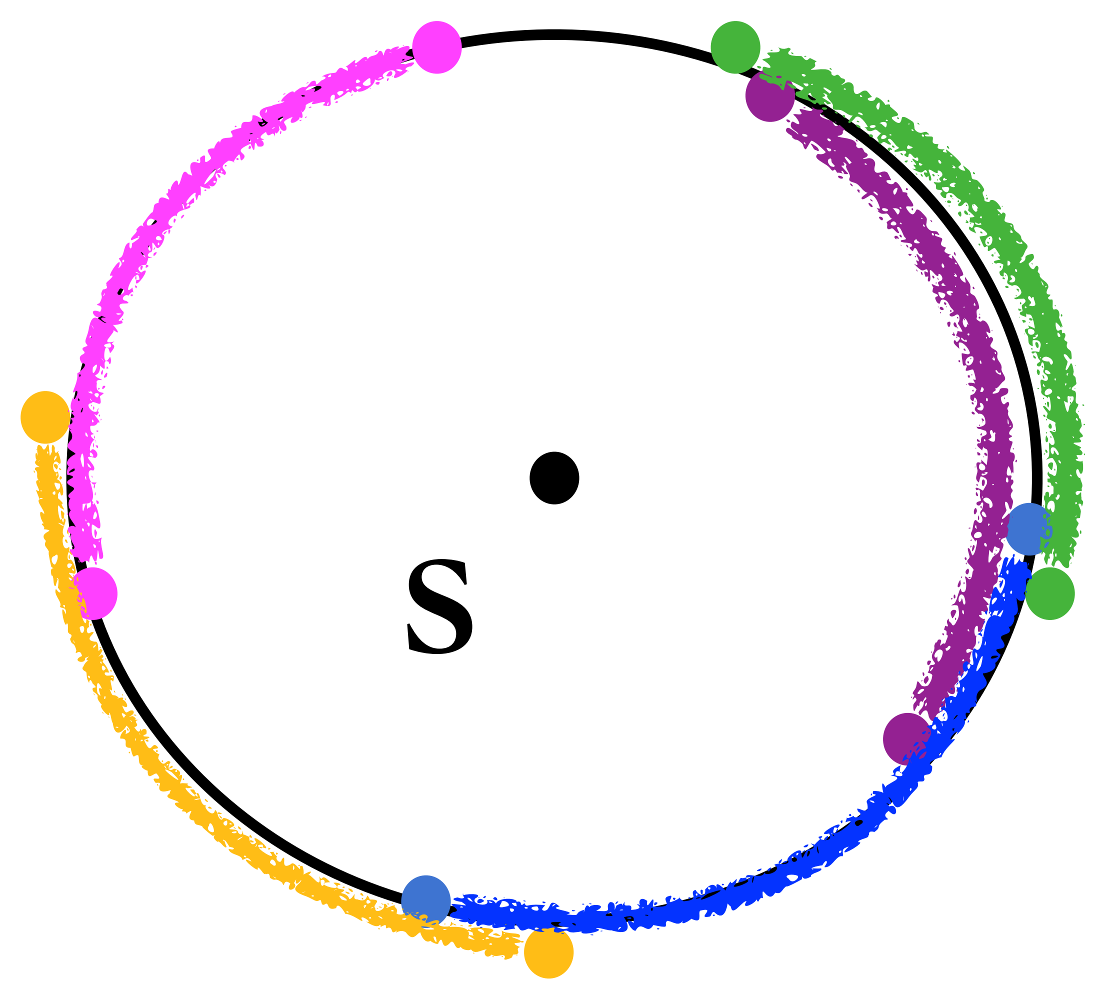


O(n)-Time Ray Sweeping Algorithm

Ray Sweeping:
Use a ray to sweep endpoints in clockwise oder to compute c

If it meets an Entry Point —— Event 1
Compute # of arcs it pierces in O(1) time

If it meets an Exit Point —— Event 2
Compute # of arcs it pierces in O(1) time



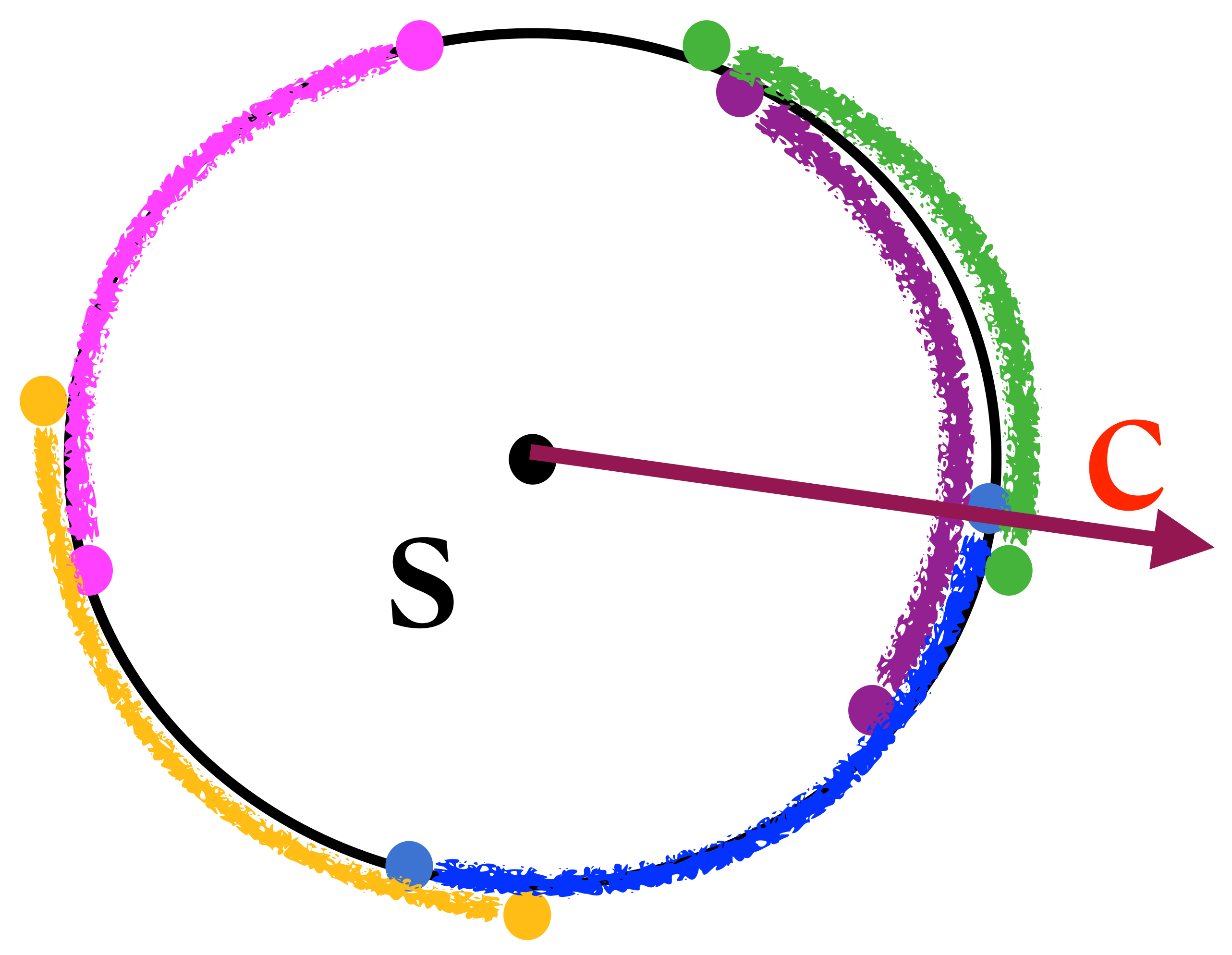
Time Complexity: O(n)

O(n)-Time Ray Sweeping Algorithm

Ray Sweeping:
Use a ray to sweep endpoints in clockwise oder to compute c

If it meets an Entry Point —— Event 1
Compute # of arcs it pierces in O(1) time

If it meets an Exit Point —— Event 2
Compute # of arcs it pierces in O(1) time



Time Complexity: O(n)

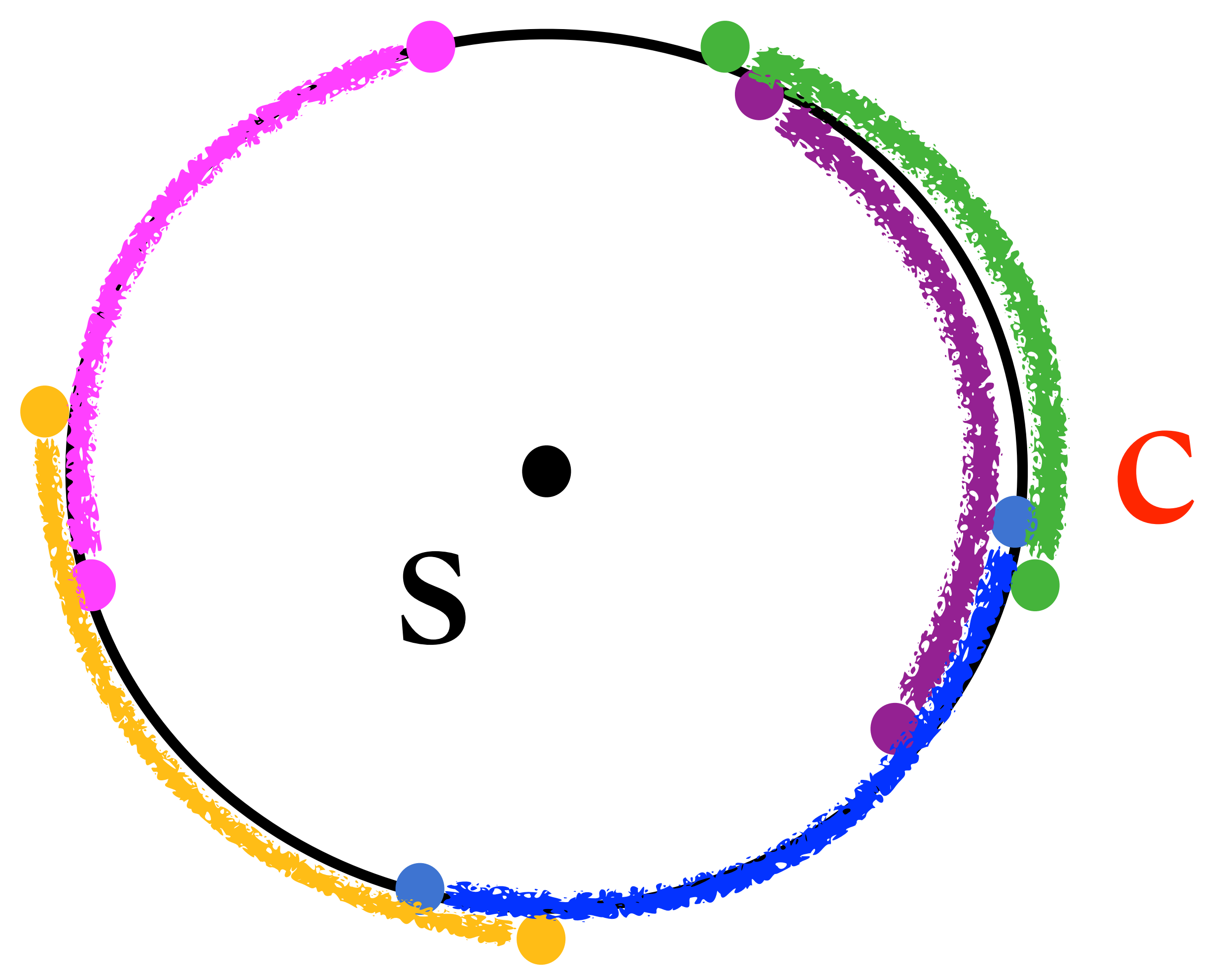
Piercing 3 arcs: MaxCount = 3

O(n)-Time Ray Sweeping Algorithm

Ray Sweeping:
Use a ray to sweep endpoints in clockwise oder to compute c

If it meets an Entry Point —— Event 1
Compute # of arcs it pierces in O(1) time

If it meets an Exit Point —— Event 2
Compute # of arcs it pierces in O(1) time



Time Complexity: O(n)

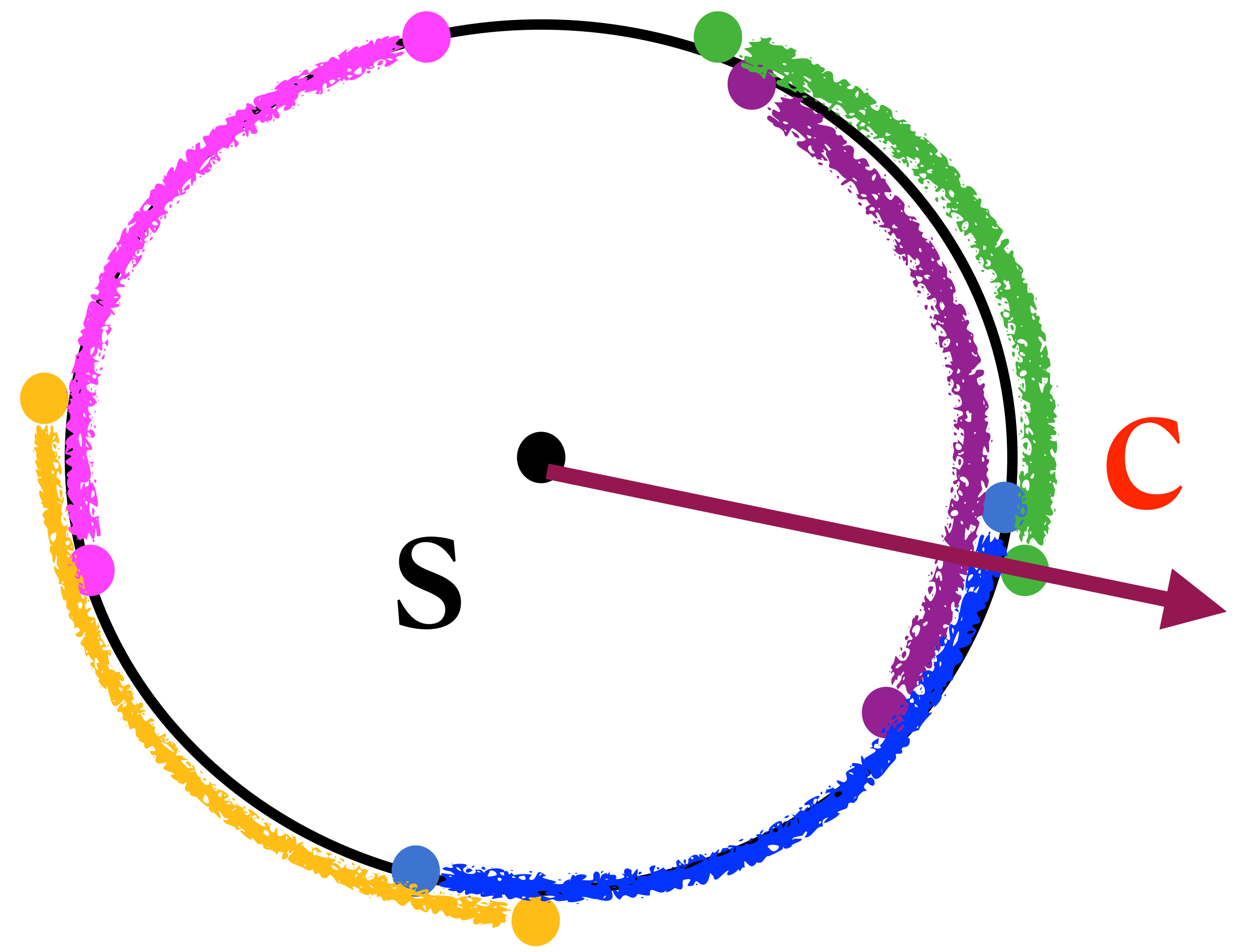
O(n)-Time Ray Sweeping Algorithm

Ray Sweeping:
Use a ray to sweep endpoints in clockwise oder to compute c

If it meets an Entry Point —— Event 1
Compute # of arcs it pierces in O(1) time

If it meets an Exit Point —— Event 2
Compute # of arcs it pierces in O(1) time

Time Complexity: O(n)



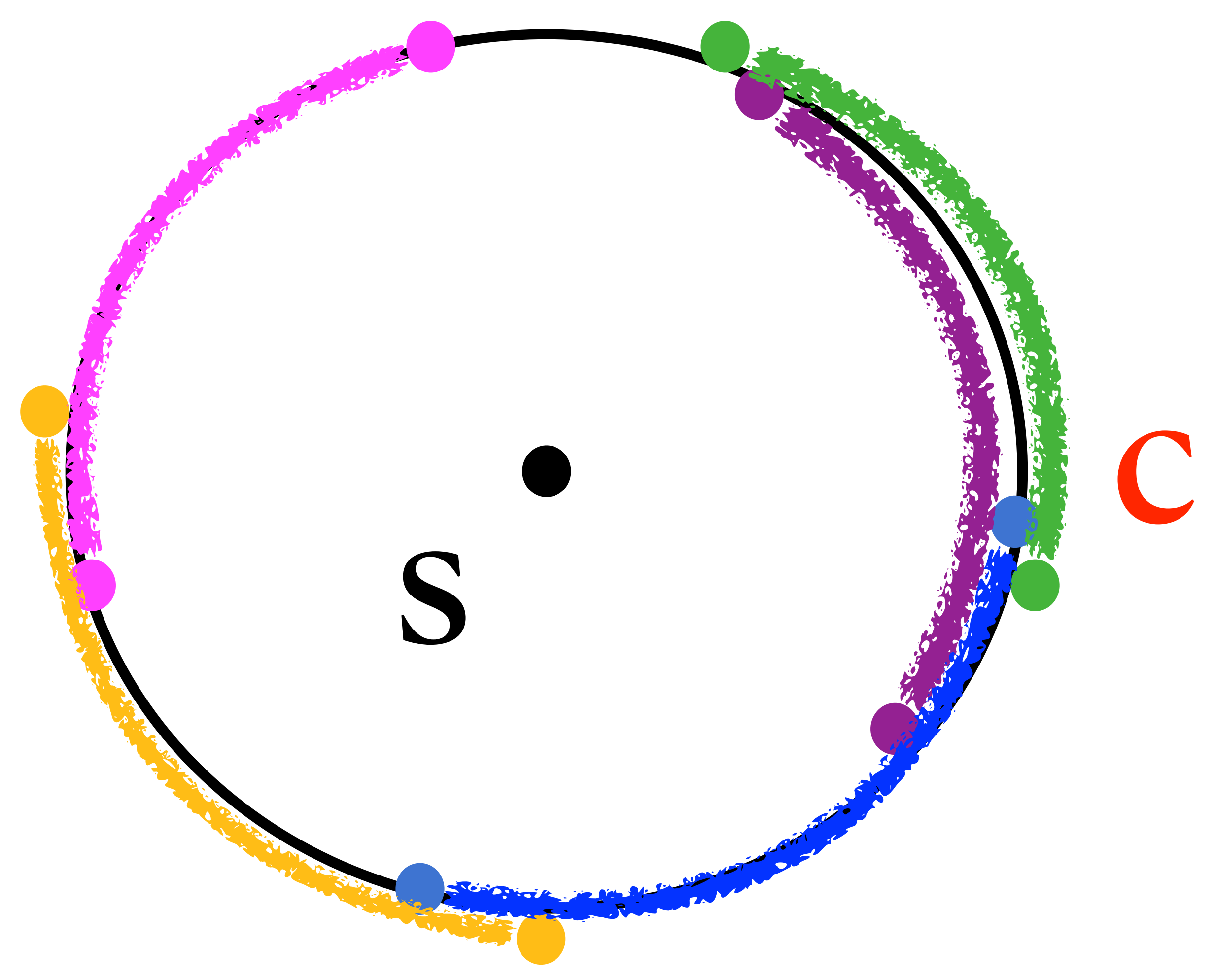
Piercing 3 arcs: MaxCount = 3

O(n)-Time Ray Sweeping Algorithm

Ray Sweeping:
Use a ray to sweep endpoints in clockwise oder to compute c

If it meets an Entry Point —— Event 1
    Compute # of arcs it pierces in O(1) time

If it meets an Exit Point —— Event 2
    Compute # of arcs it pierces in O(1) time



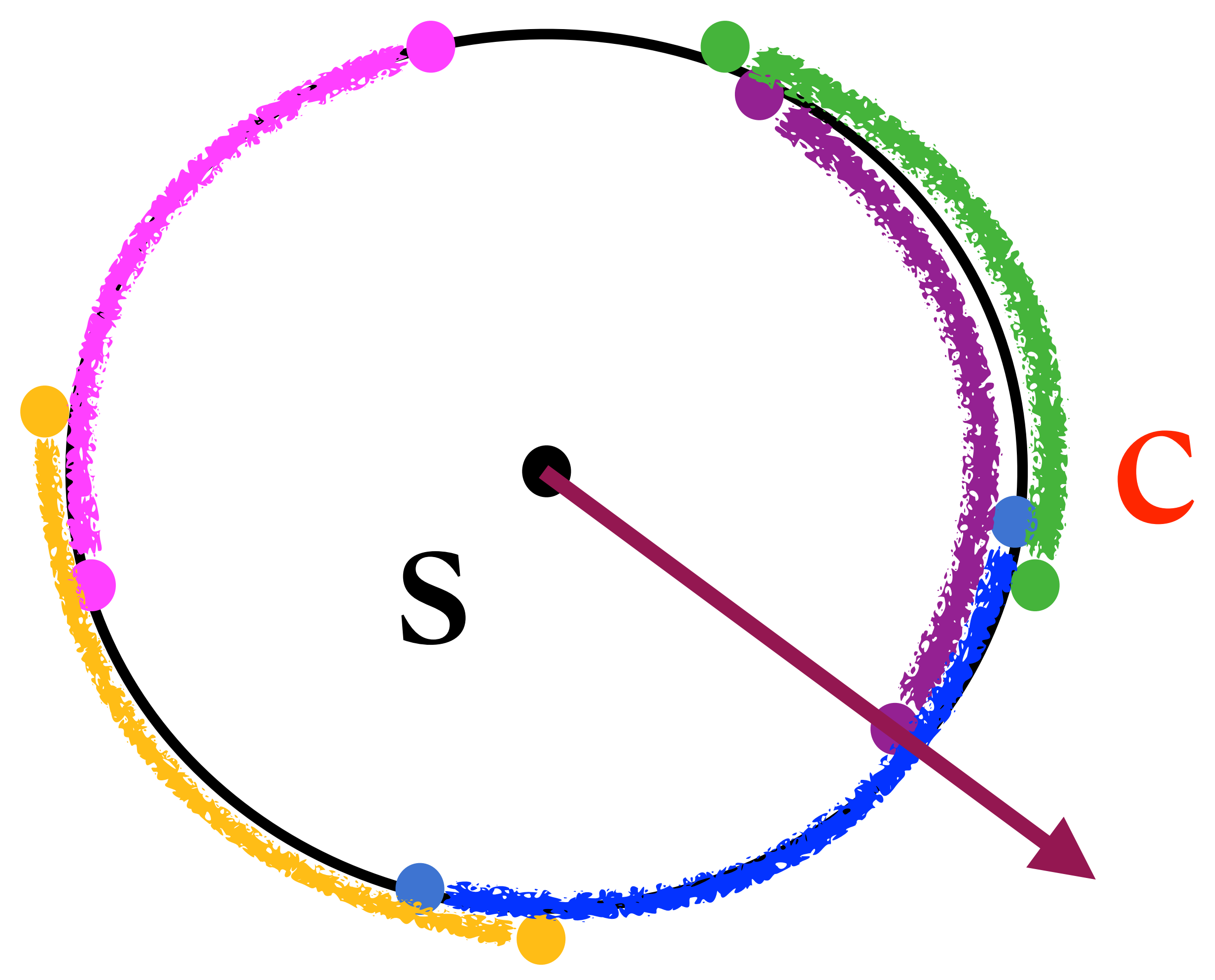
Time Complexity: O(n)

O(n)-Time Ray Sweeping Algorithm

Ray Sweeping:
Use a ray to sweep endpoints in clockwise oder to compute c

If it meets an Entry Point —— Event 1
Compute # of arcs it pierces in O(1) time

If it meets an Exit Point —— Event 2
Compute # of arcs it pierces in O(1) time



Time Complexity: O(n)

Piercing 2 arcs: MaxCount = 3

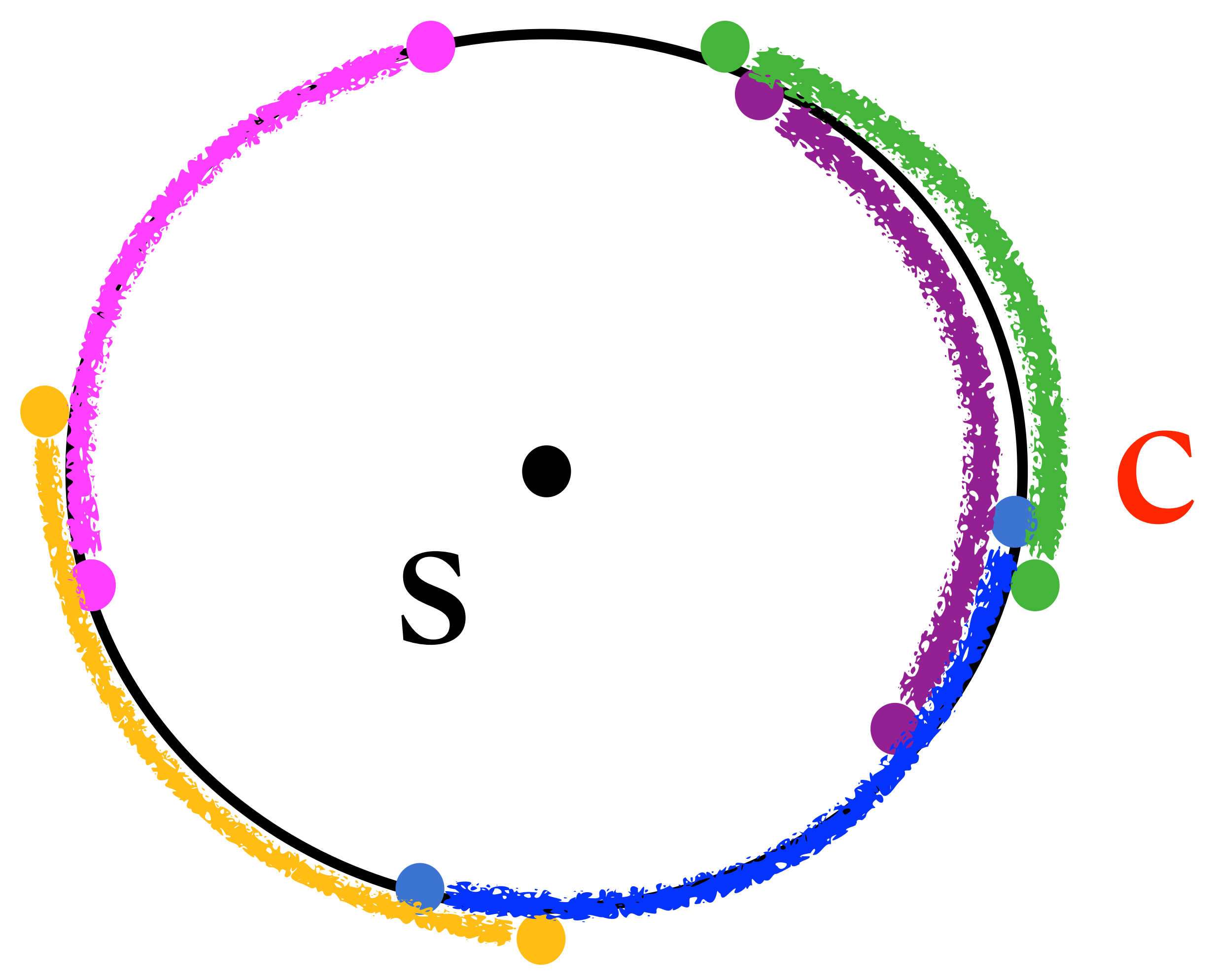
O(n)-Time Ray Sweeping Algorithm

Ray Sweeping:
Use a ray to sweep endpoints in clockwise oder to compute c

If it meets an Entry Point —— Event 1
Compute # of arcs it pierces in O(1) time

If it meets an Exit Point —— Event 2
Compute # of arcs it pierces in O(1) time

Time Complexity: O(n)



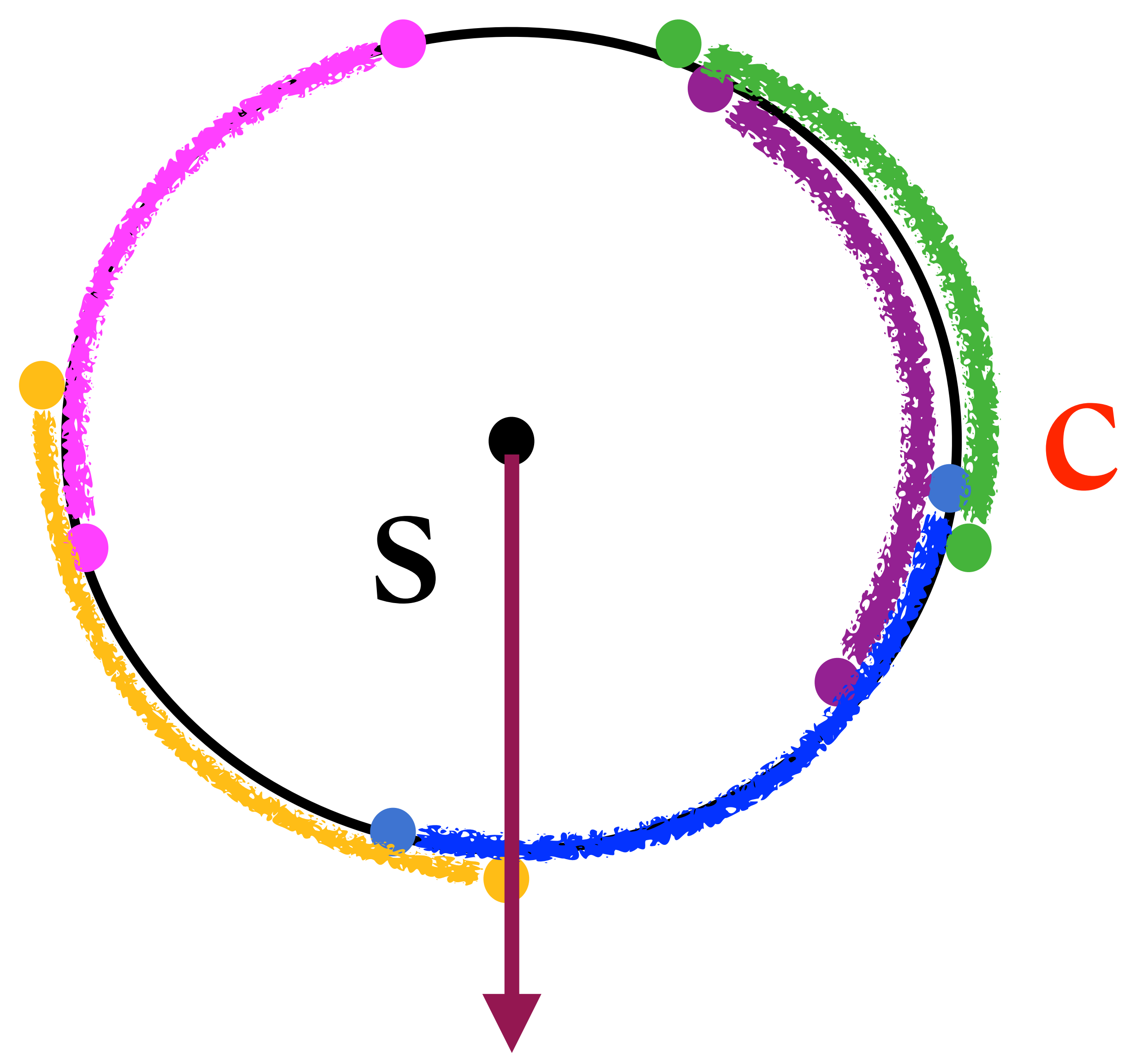
O(n)-Time Ray Sweeping Algorithm

Ray Sweeping:
Use a ray to sweep endpoints in clockwise oder to compute c

If it meets an Entry Point —— Event 1
Compute # of arcs it pierces in O(1) time

If it meets an Exit Point —— Event 2
Compute # of arcs it pierces in O(1) time

Time Complexity: O(n)



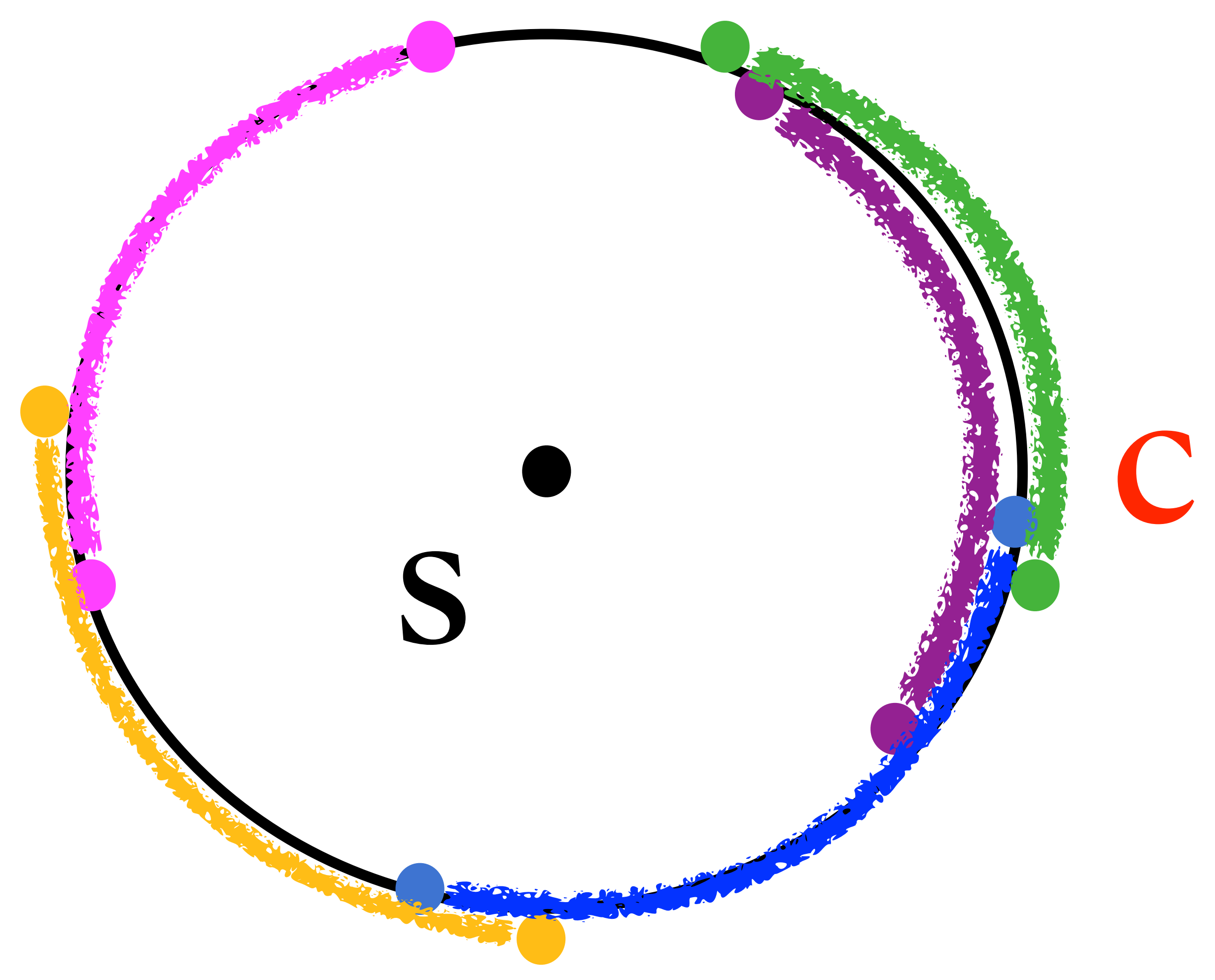
Piercing 2 arcs: MaxCount = 3

O(n)-Time Ray Sweeping Algorithm

Ray Sweeping:
Use a ray to sweep endpoints in clockwise oder to compute c

If it meets an Entry Point —— Event 1
    Compute # of arcs it pierces in O(1) time

If it meets an Exit Point —— Event 2
    Compute # of arcs it pierces in O(1) time



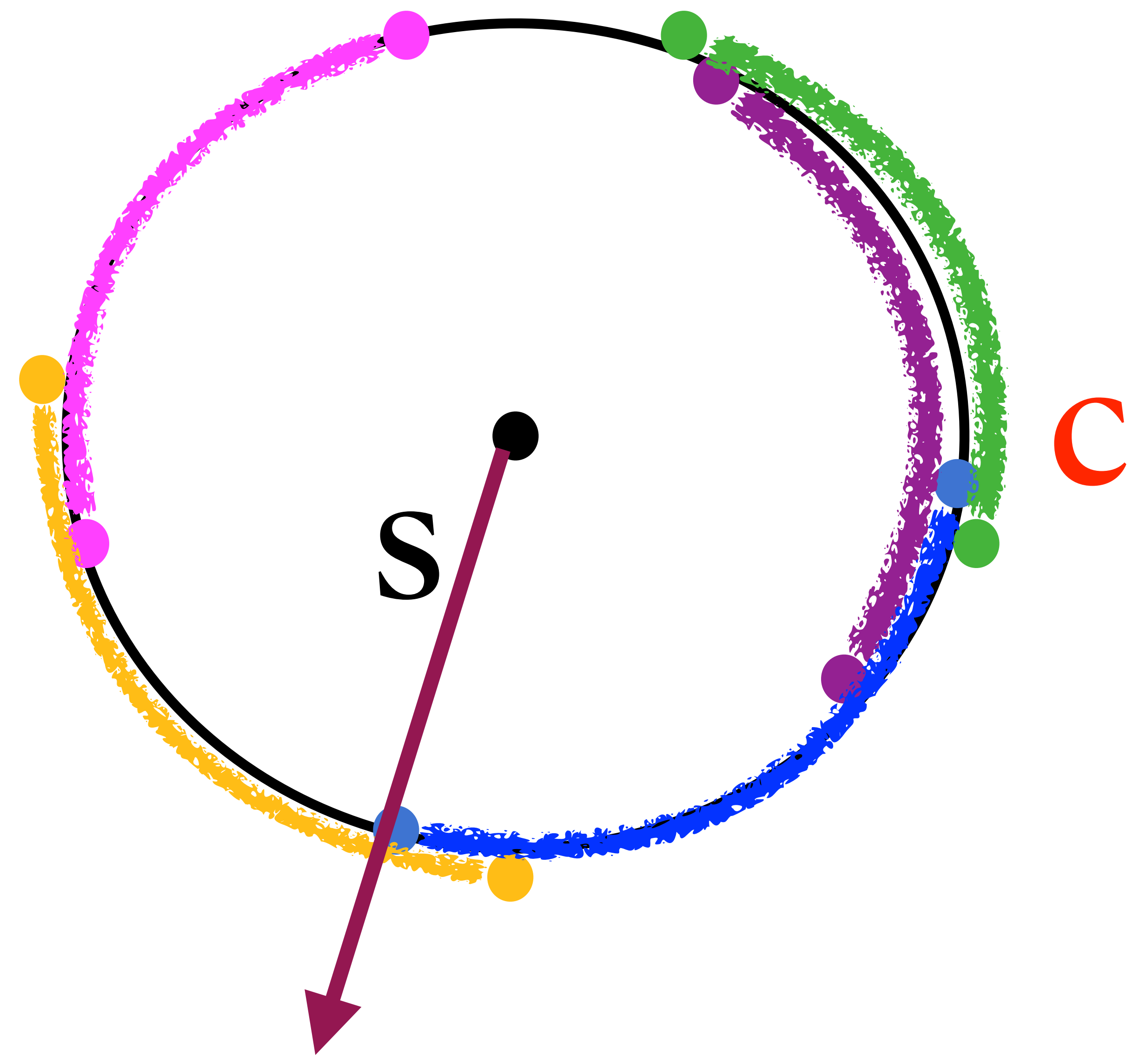
Time Complexity: O(n)

O(n)-Time Ray Sweeping Algorithm

Ray Sweeping:
Use a ray to sweep endpoints in clockwise oder to compute c

If it meets an Entry Point —— Event 1
Compute # of arcs it pierces in O(1) time

If it meets an Exit Point —— Event 2
Compute # of arcs it pierces in O(1) time



Time Complexity: O(n)

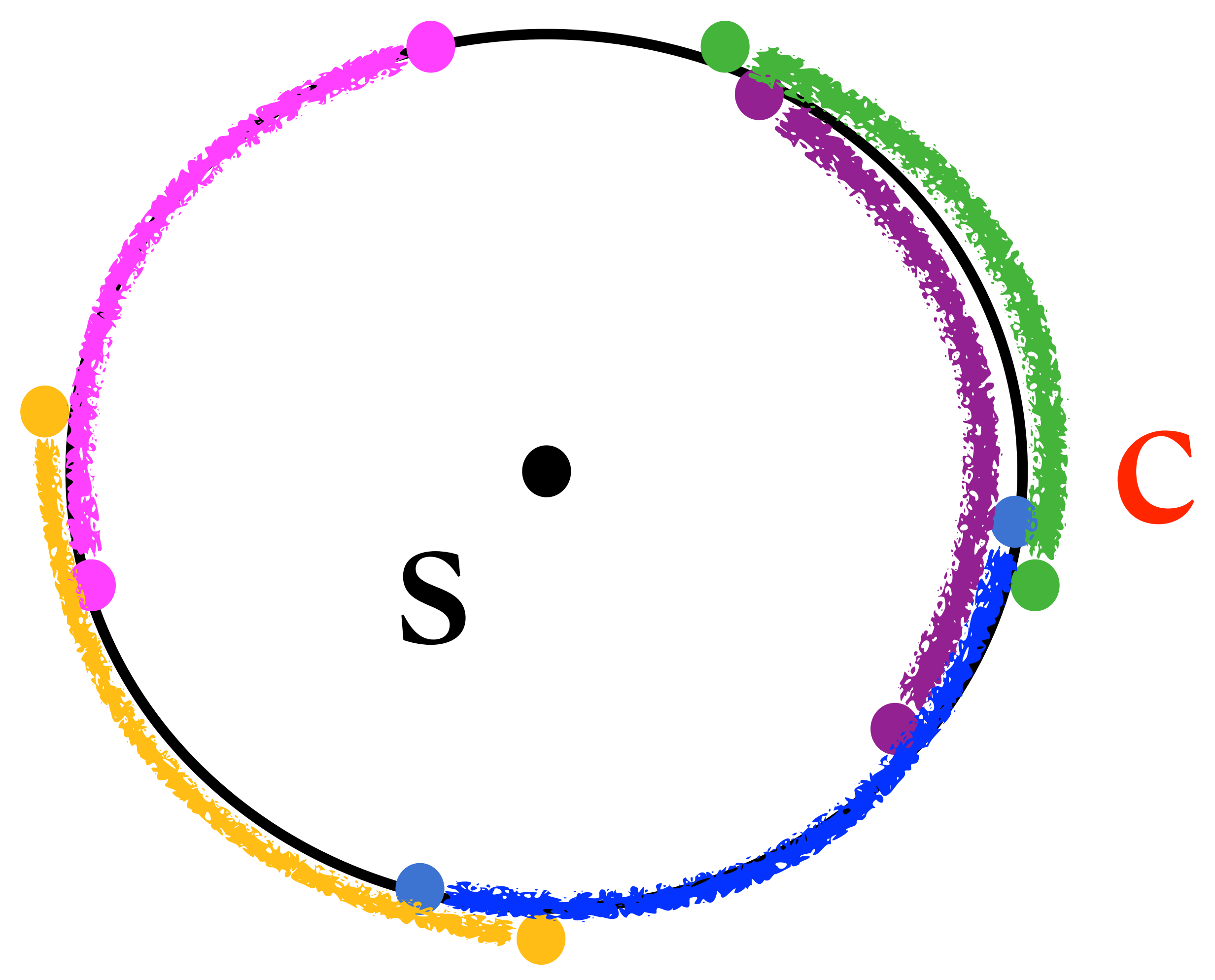
Piercing 2 arcs: MaxCount = 3

O(n)-Time Ray Sweeping Algorithm

Ray Sweeping:
Use a ray to sweep endpoints in clockwise oder to compute c

If it meets an Entry Point —— Event 1
    Compute # of arcs it pierces in O(1) time

If it meets an Exit Point —— Event 2
    Compute # of arcs it pierces in O(1) time



Time Complexity: O(n)

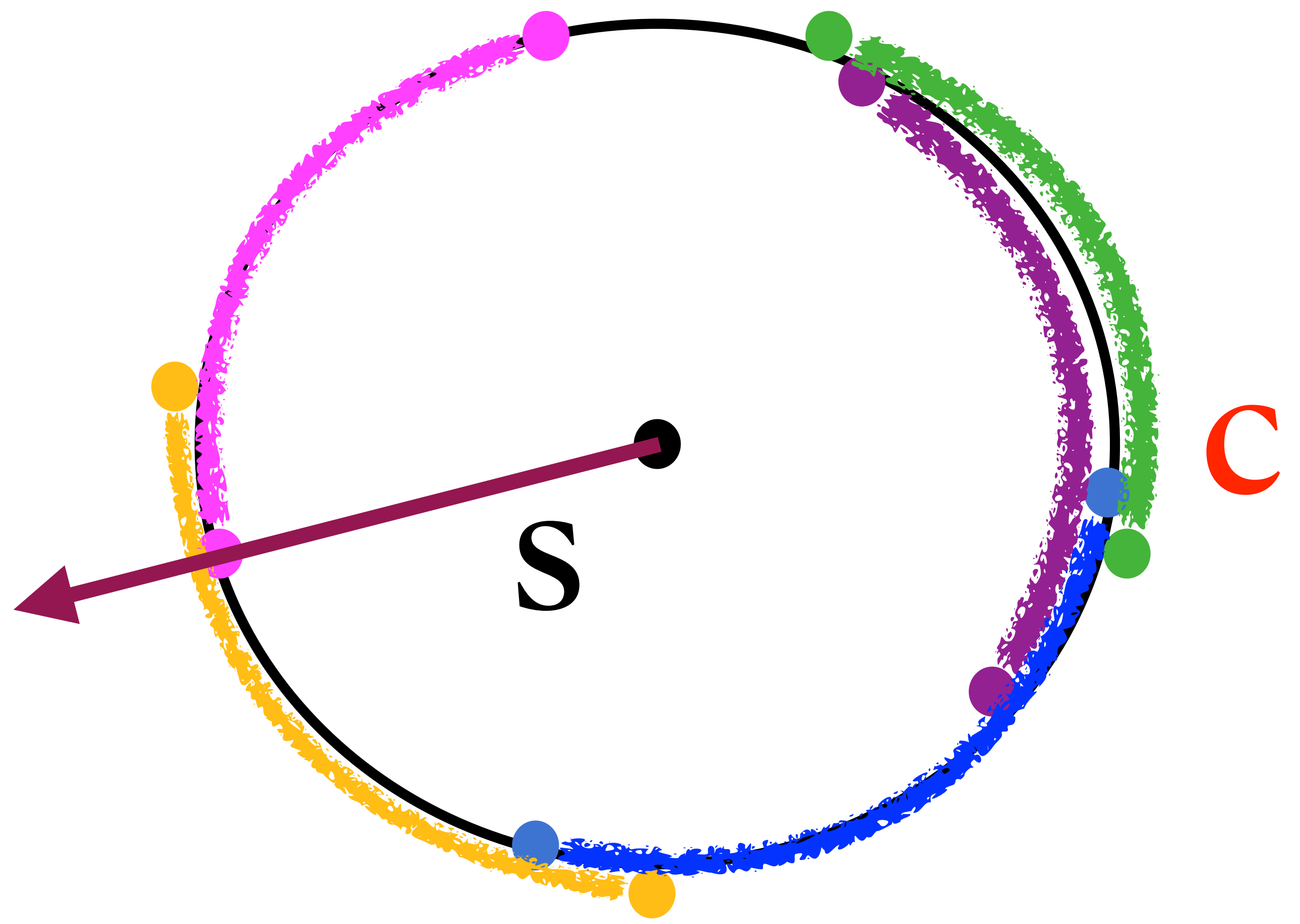
O(n)-Time Ray Sweeping Algorithm

Ray Sweeping:
Use a ray to sweep endpoints in clockwise oder to compute c

If it meets an Entry Point —— Event 1
Compute # of arcs it pierces in O(1) time

If it meets an Exit Point —— Event 2
Compute # of arcs it pierces in O(1) time

Time Complexity: O(n)



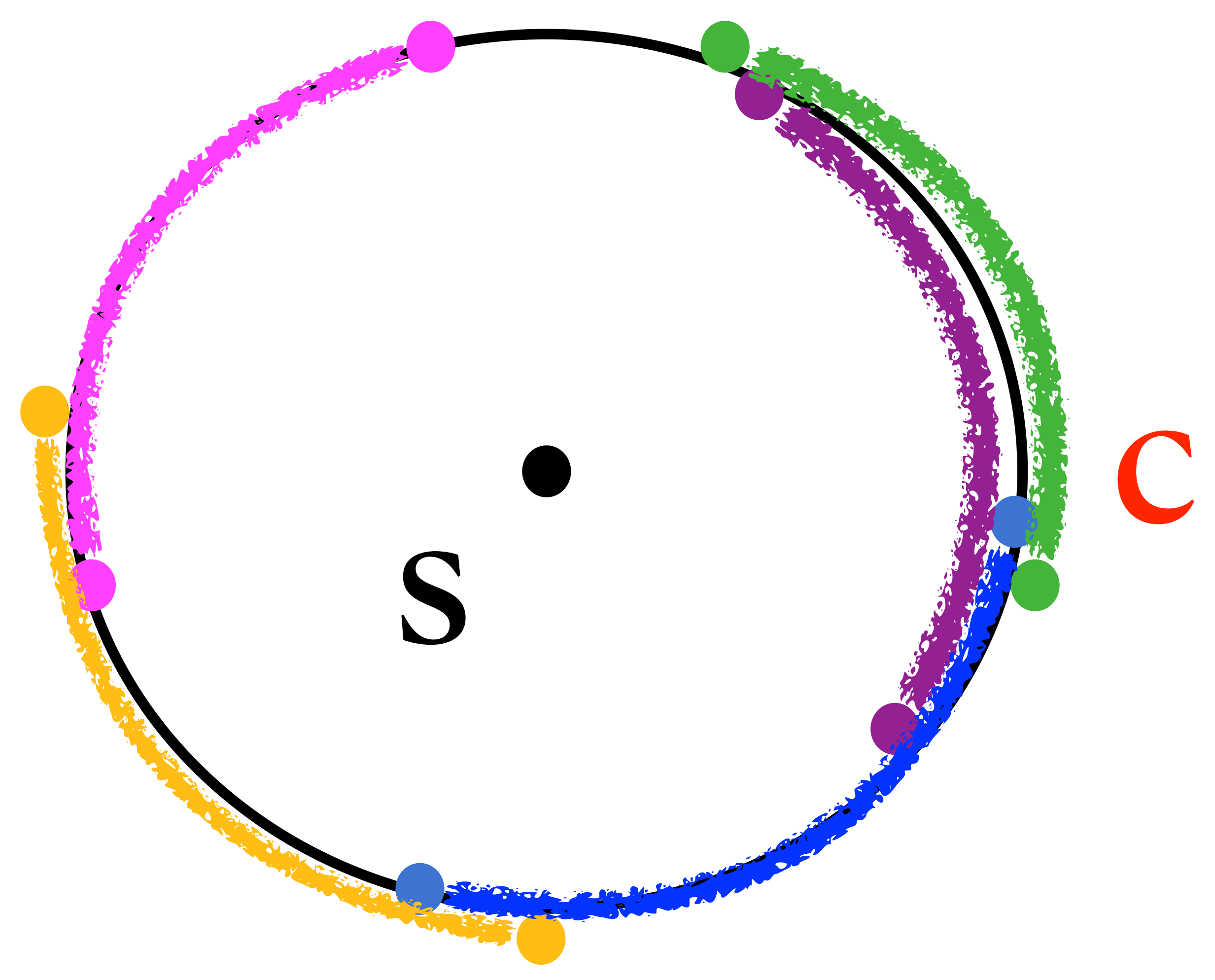
Piercing 2 arcs: MaxCount = 3

O(n)-Time Ray Sweeping Algorithm

Ray Sweeping:
Use a ray to sweep endpoints in clockwise oder to compute c

If it meets an Entry Point —— Event 1
Compute # of arcs it pierces in O(1) time

If it meets an Exit Point —— Event 2
Compute # of arcs it pierces in O(1) time



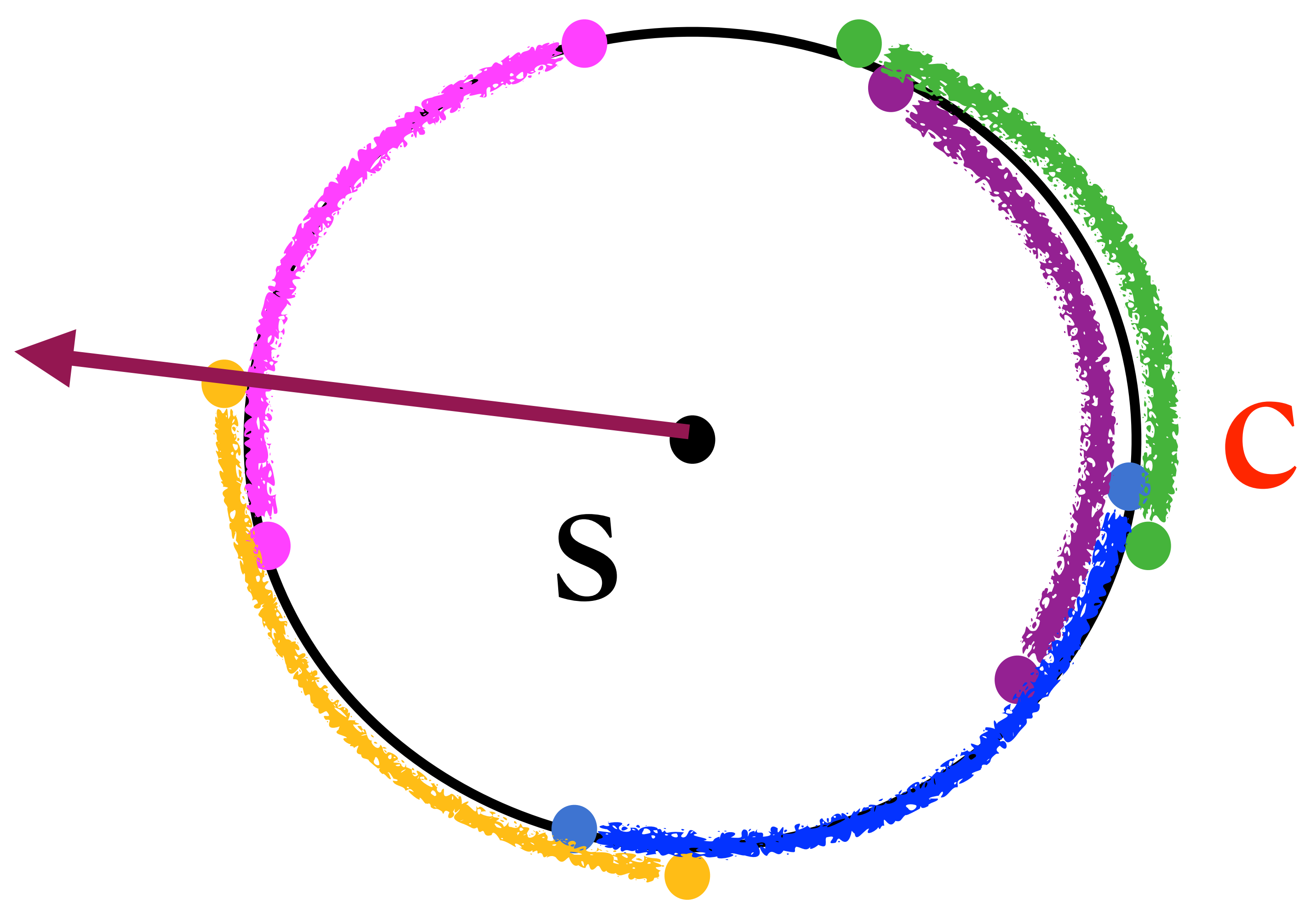
Time Complexity: O(n)

O(n)-Time Ray Sweeping Algorithm

Ray Sweeping:
Use a ray to sweep endpoints in clockwise oder to compute c

If it meets an Entry Point —— Event 1
    Compute # of arcs it pierces in O(1) time

If it meets an Exit Point —— Event 2
    Compute # of arcs it pierces in O(1) time



Time Complexity: O(n)

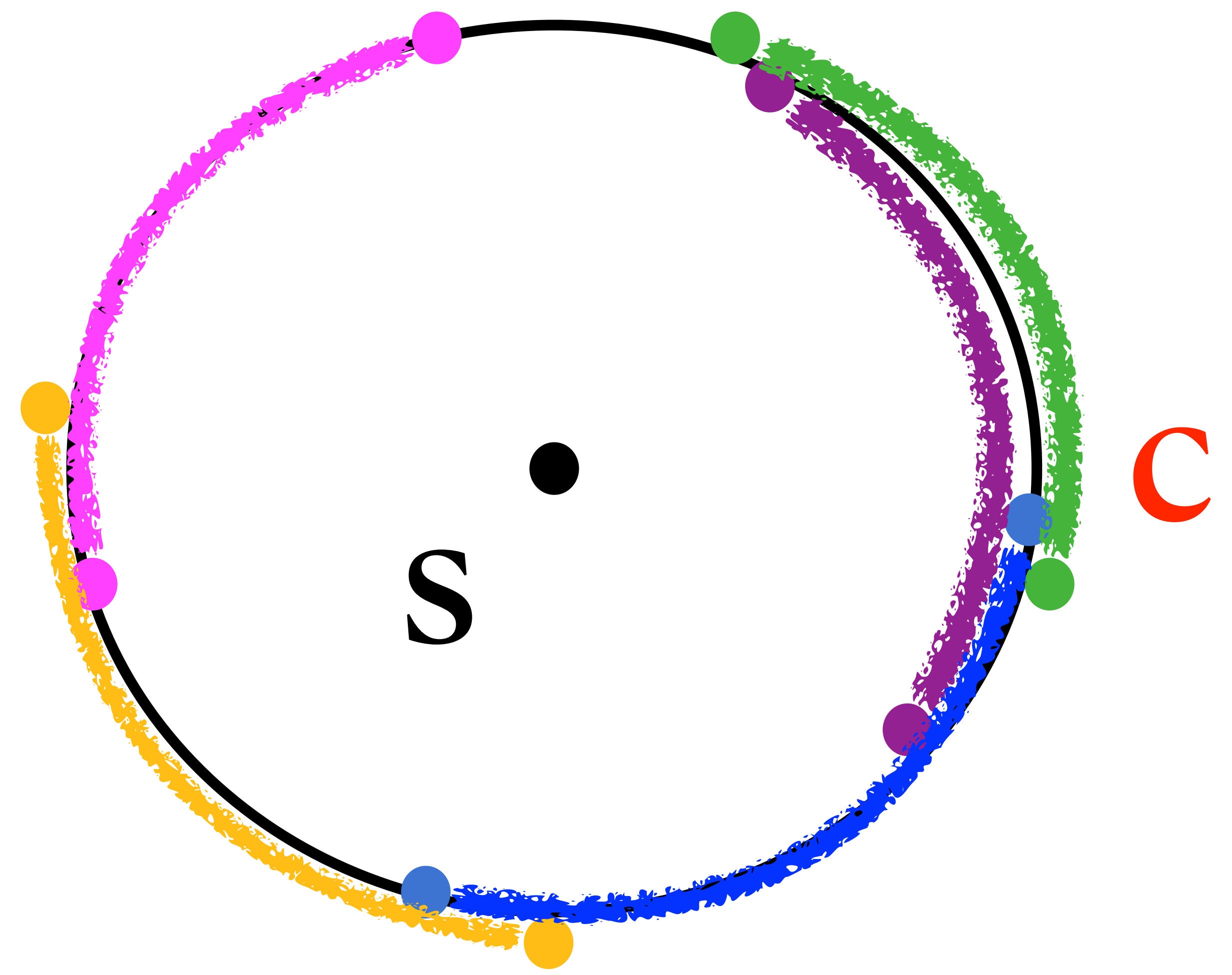
Piercing 2 arcs: MaxCount = 3

O(n)-Time Ray Sweeping Algorithm

Ray Sweeping:
Use a ray to sweep endpoints in clockwise oder to compute c

If it meets an Entry Point —— Event 1
    Compute # of arcs it pierces in O(1) time

If it meets an Exit Point —— Event 2
    Compute # of arcs it pierces in O(1) time



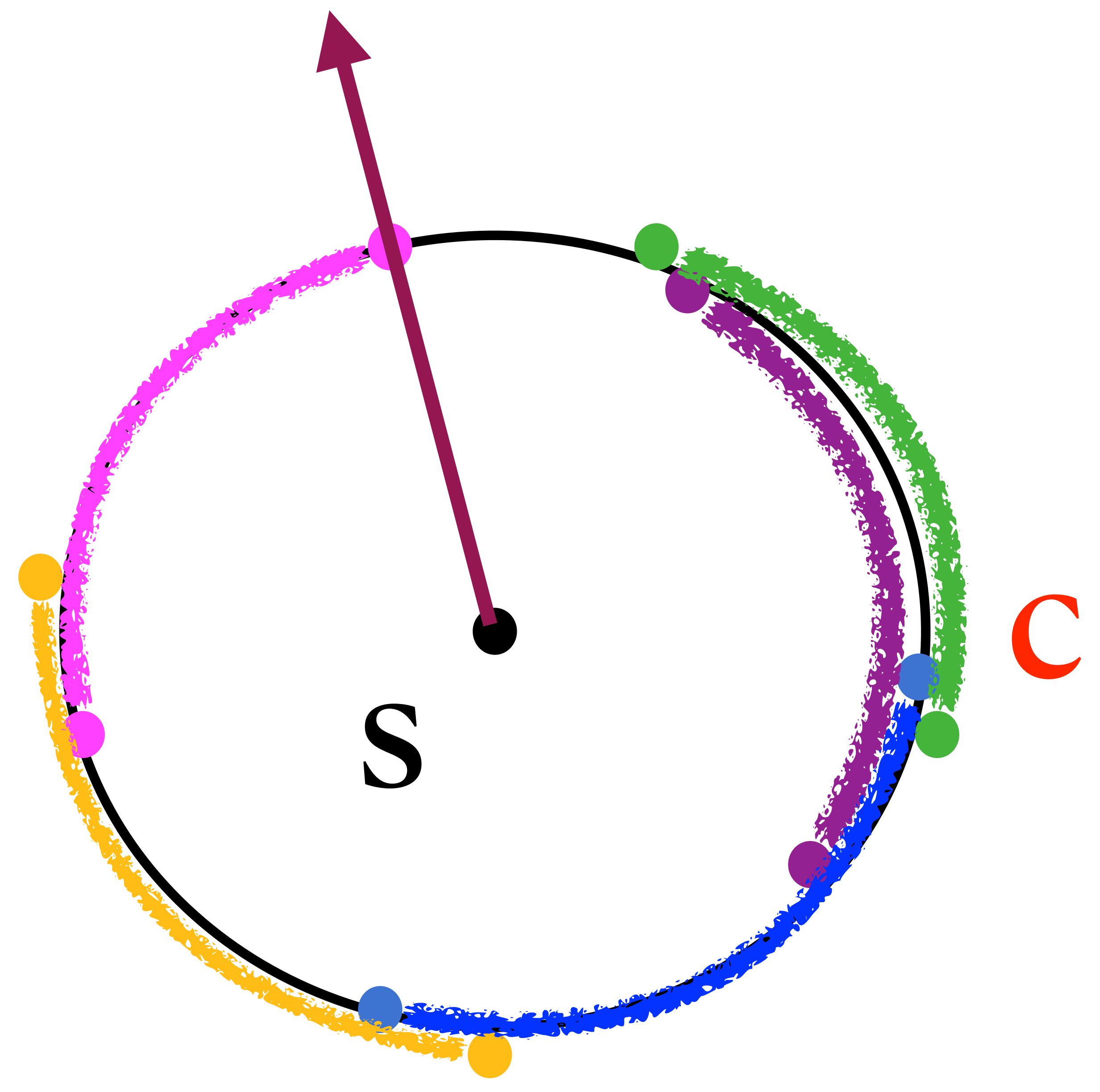
Time Complexity: O(n)

O(n)-Time Ray Sweeping Algorithm

Ray Sweeping:
Use a ray to sweep endpoints in clockwise oder to compute c

If it meets an Entry Point —— Event 1
Compute # of arcs it pierces in O(1) time

If it meets an Exit Point —— Event 2
Compute # of arcs it pierces in O(1) time



Time Complexity: O(n)

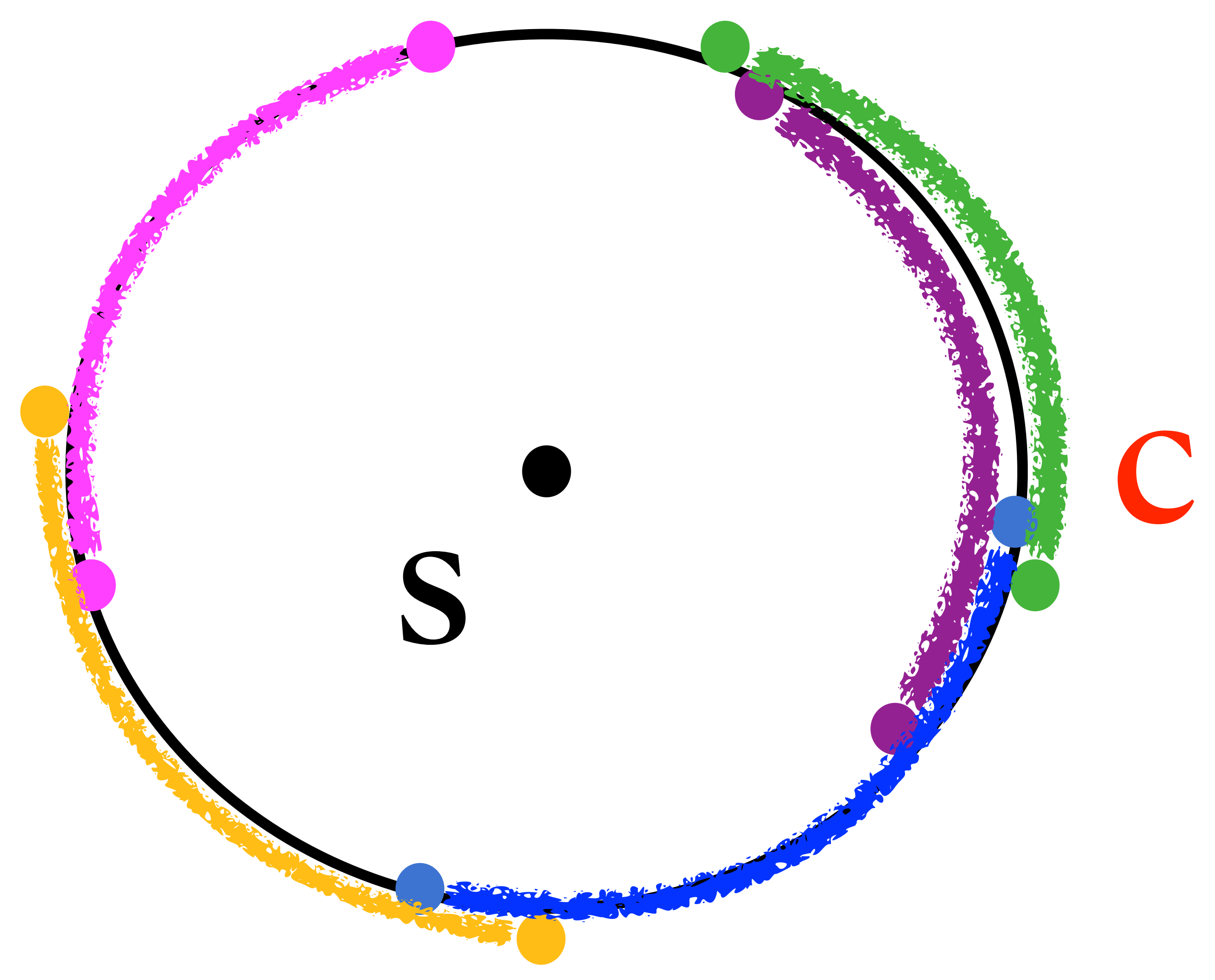
Piercing 1 arcs: MaxCount = 3

O(n)-Time Ray Sweeping Algorithm

Ray Sweeping:
Use a ray to sweep endpoints in clockwise oder to compute c

If it meets an Entry Point —— Event 1
    Compute # of arcs it pierces in O(1) time

If it meets an Exit Point —— Event 2
    Compute # of arcs it pierces in O(1) time



Time Complexity: O(n)

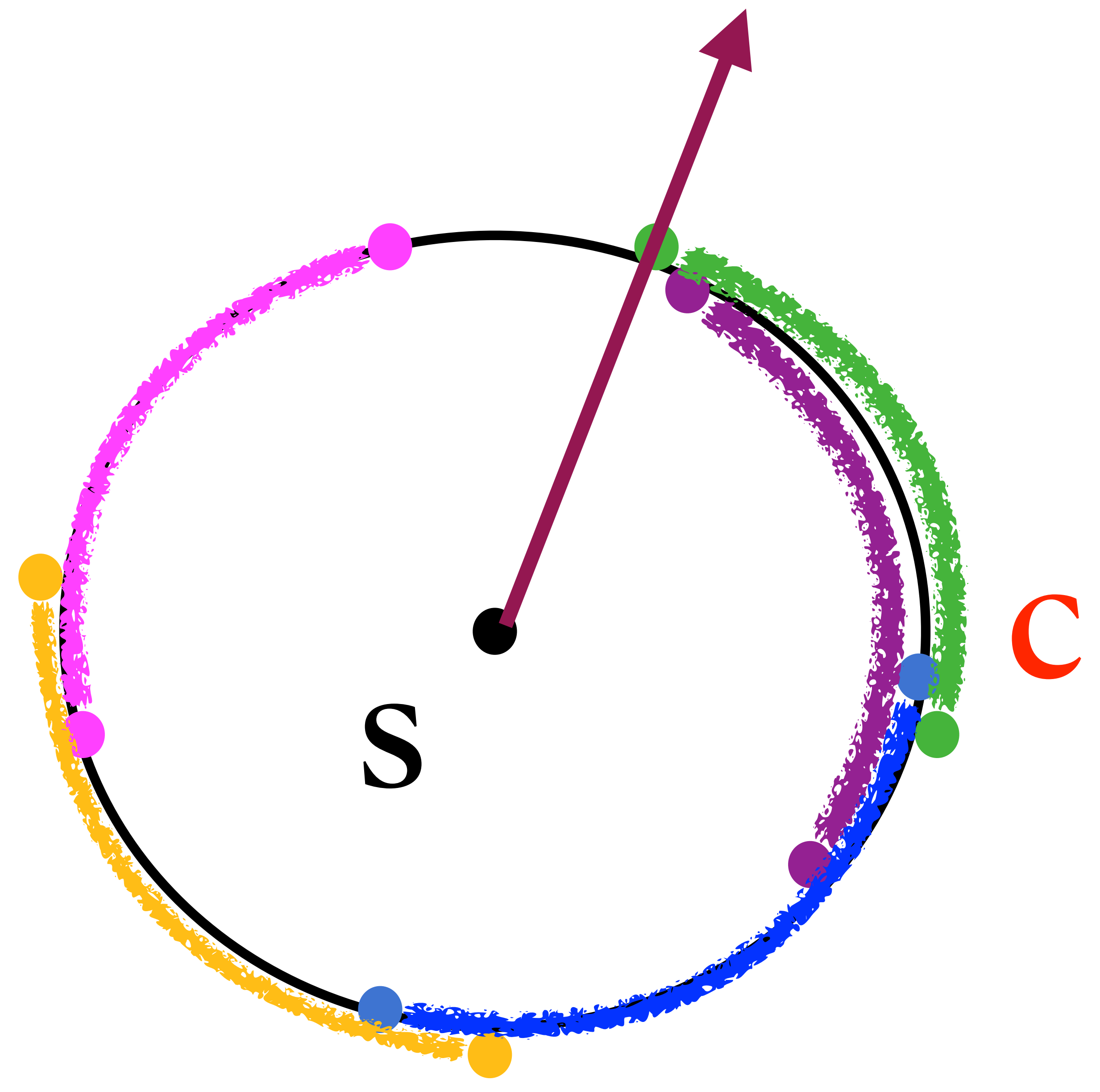
O(n)-Time Ray Sweeping Algorithm

Ray Sweeping:
Use a ray to sweep endpoints in clockwise oder to compute c

If it meets an Entry Point —— Event 1
Compute # of arcs it pierces in O(1) time

If it meets an Exit Point —— Event 2
Compute # of arcs it pierces in O(1) time

Time Complexity: O(n)



Piercing 1 arcs: MaxCount = 3

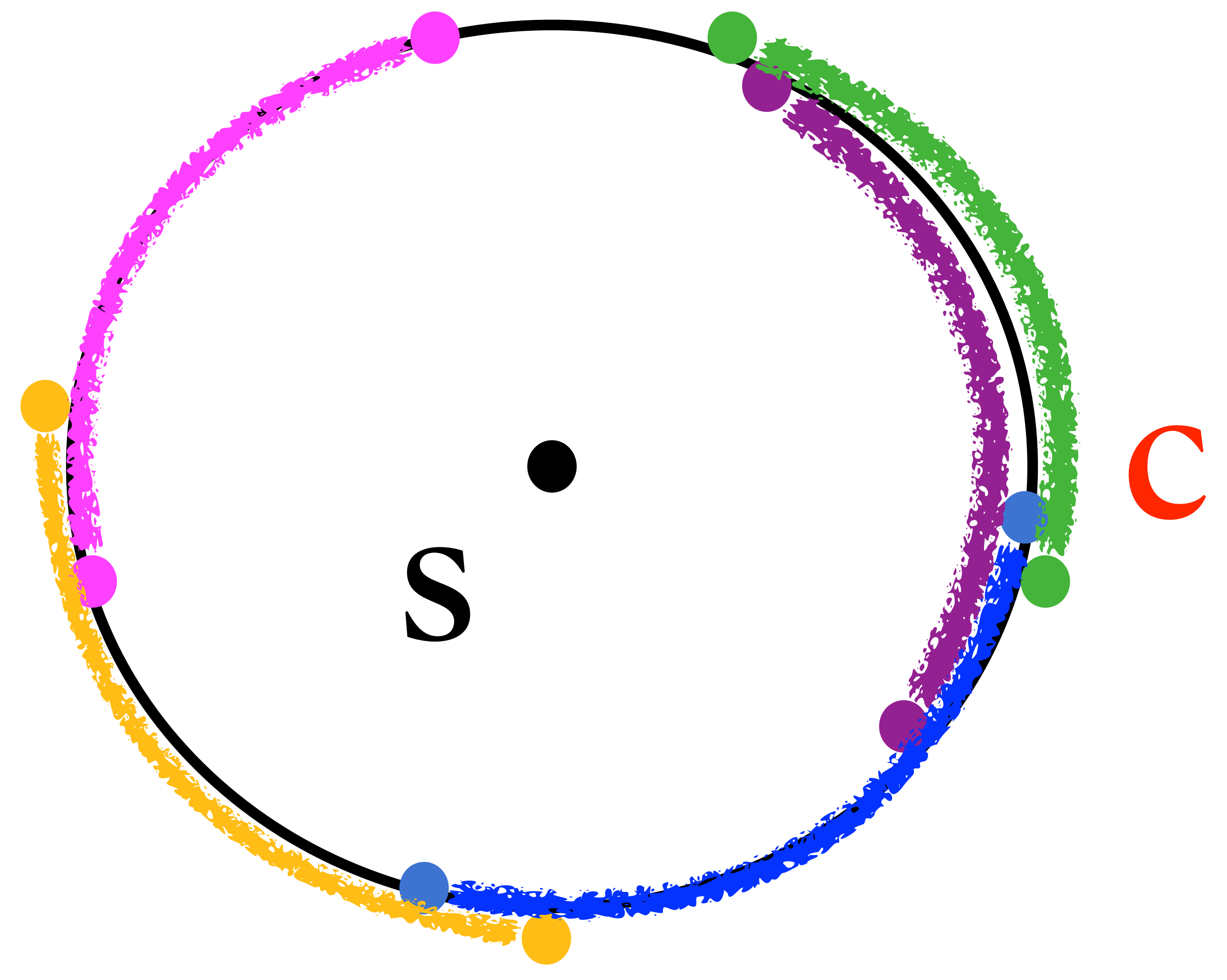
O(n)-Time Ray Sweeping Algorithm

Ray Sweeping:
Use a ray to sweep endpoints in clockwise oder to compute c

If it meets an Entry Point —— Event 1
Compute # of arcs it pierces in O(1) time

If it meets an Exit Point —— Event 2
Compute # of arcs it pierces in O(1) time

Time Complexity: O(n)



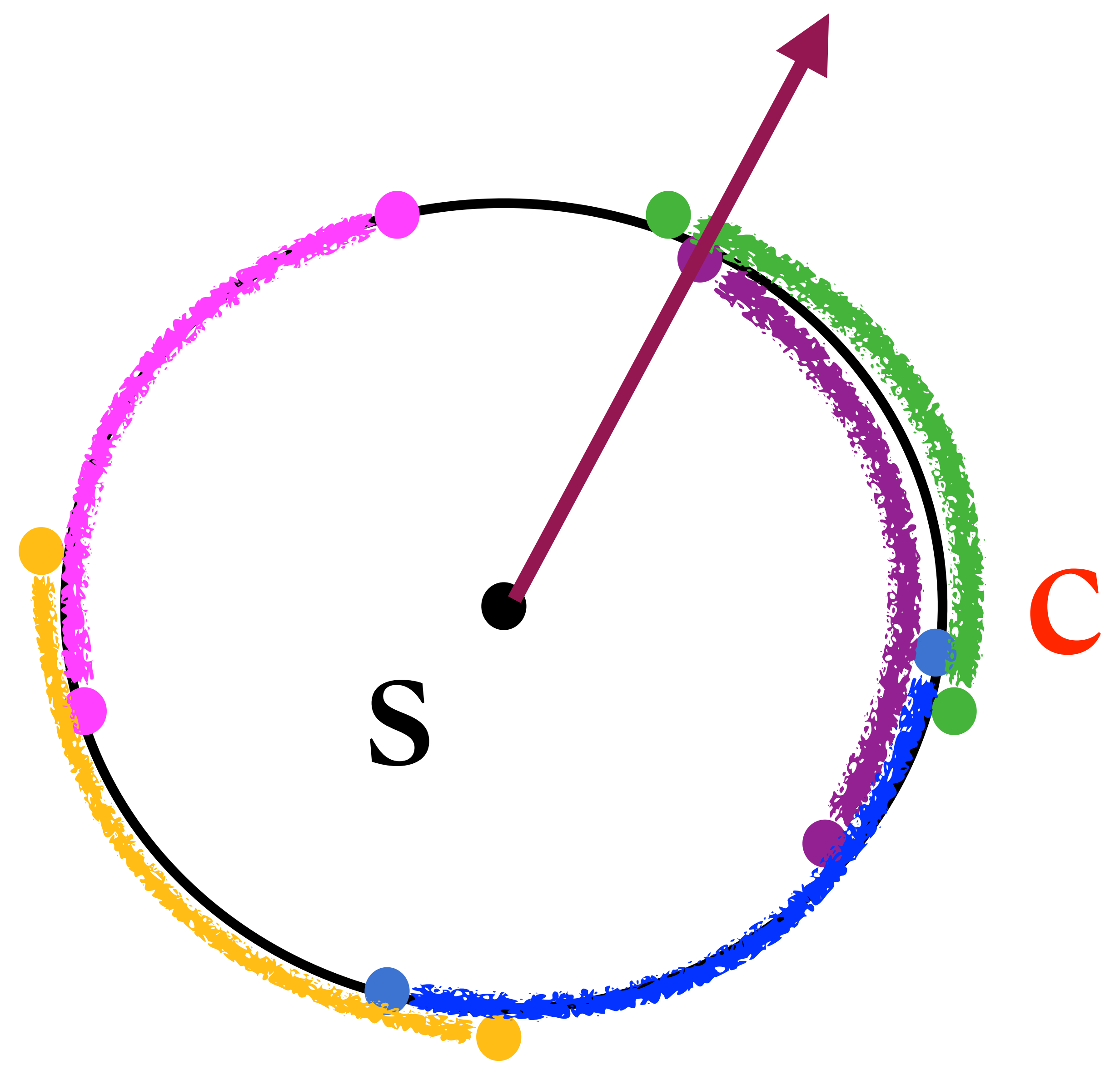
O(n)-Time Ray Sweeping Algorithm

Ray Sweeping:
Use a ray to sweep endpoints in clockwise oder to compute c

If it meets an Entry Point —— Event 1
Compute # of arcs it pierces in O(1) time

If it meets an Exit Point —— Event 2
Compute # of arcs it pierces in O(1) time

Time Complexity: O(n)



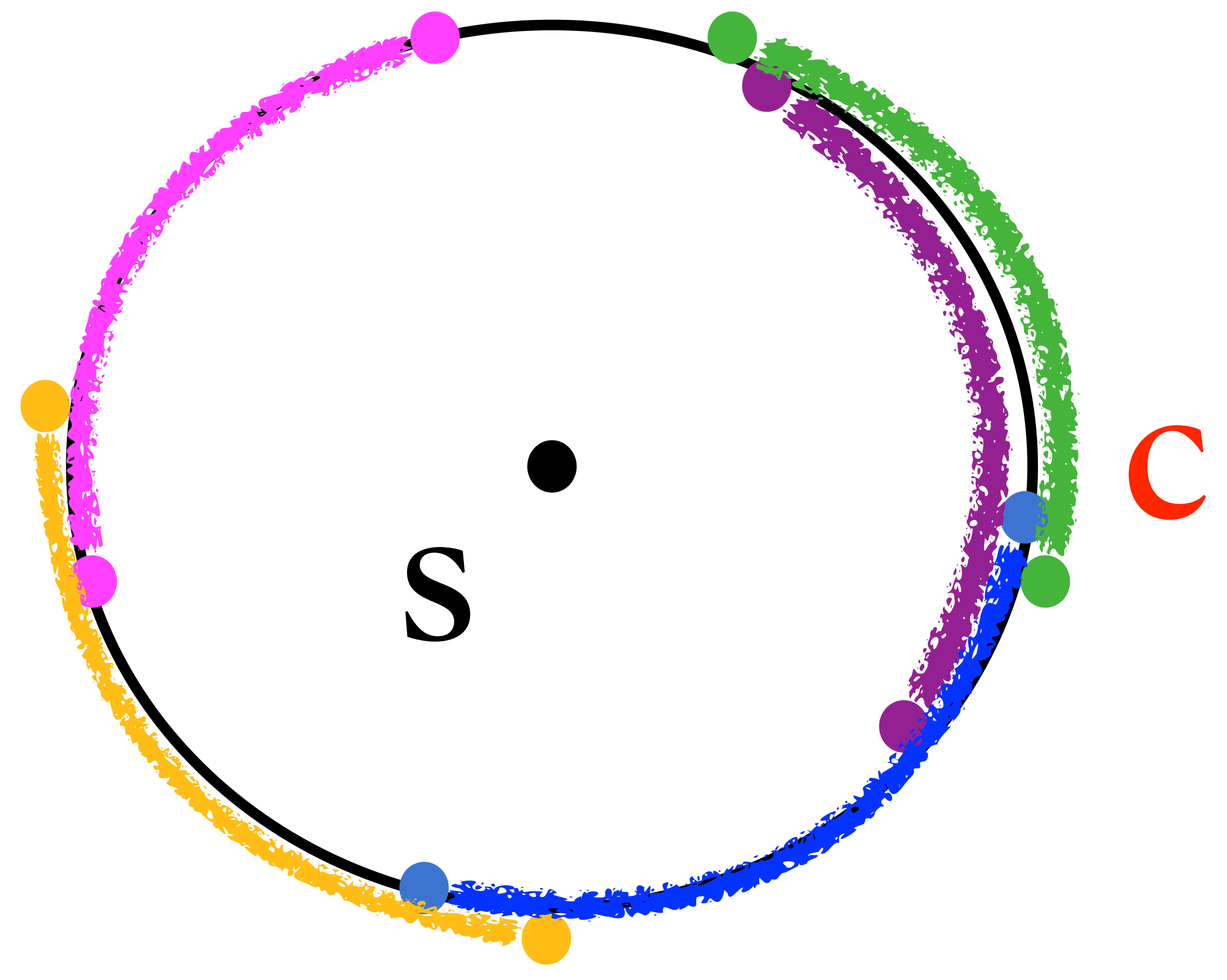
Piercing 2 arcs: MaxCount = 3

O(n)-Time Ray Sweeping Algorithm

Ray Sweeping:
Use a ray to sweep endpoints in clockwise oder to compute c

If it meets an Entry Point —— Event 1
    Compute # of arcs it pierces in O(1) time

If it meets an Exit Point —— Event 2
    Compute # of arcs it pierces in O(1) time



Time Complexity: O(n)

Event 1: The Sweeping Ray is Through an Entry Point

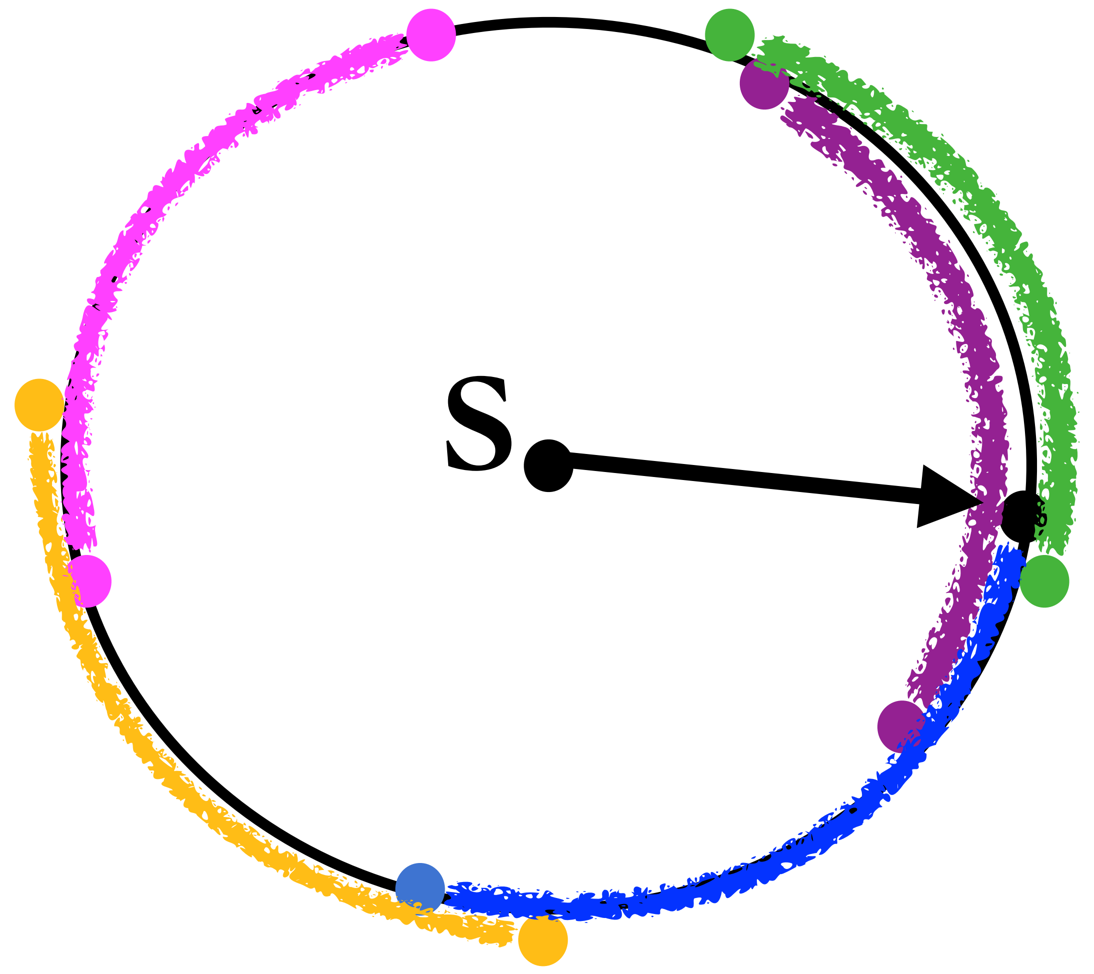
Event 1:

if the ray is through an Entry point

count++

update MaxCount and set c

set the arc's flag as true



Event 2: The Sweeping Ray is Through an Exit Point

Event 2:

if the current point is an exit point

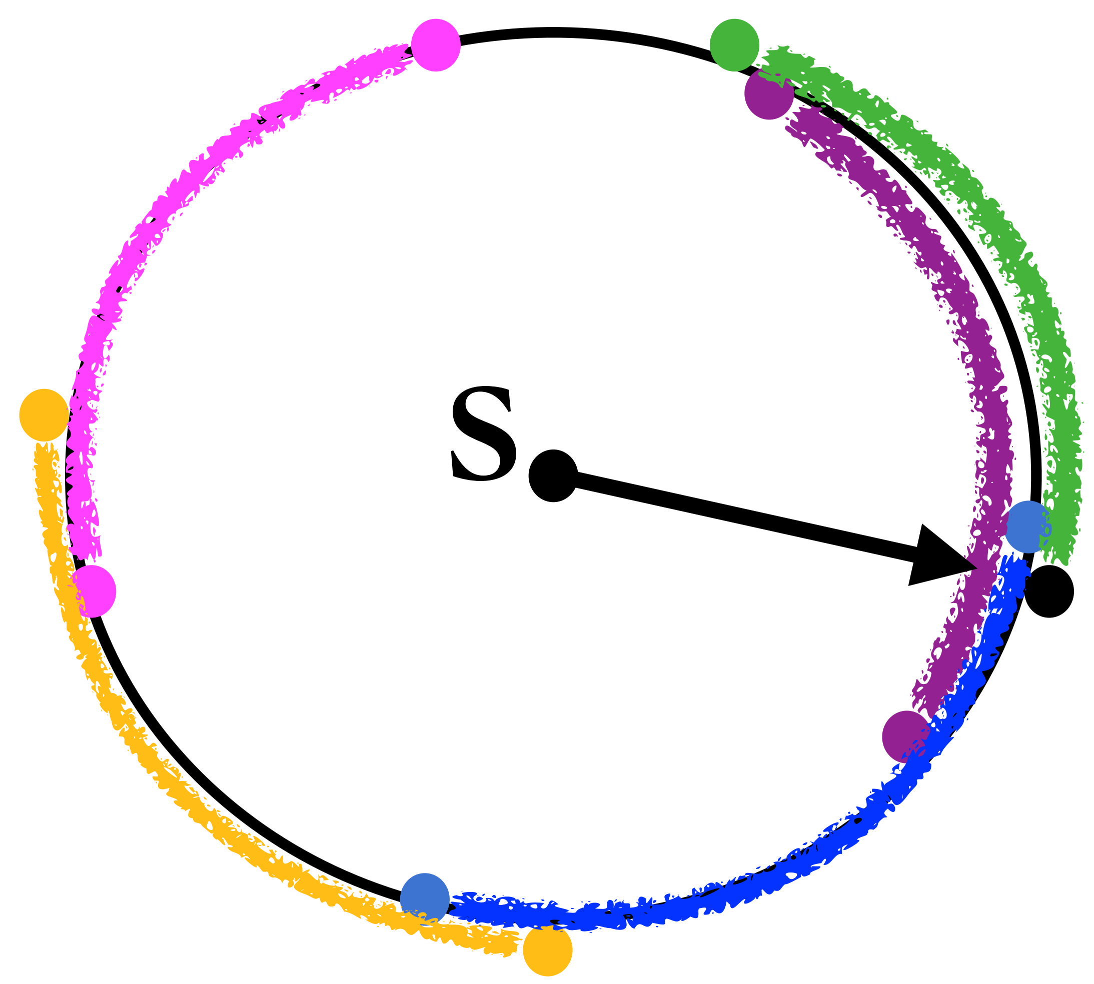
if the arc flag is false

count + +

update MaxCount and set c

set the arc flag as false

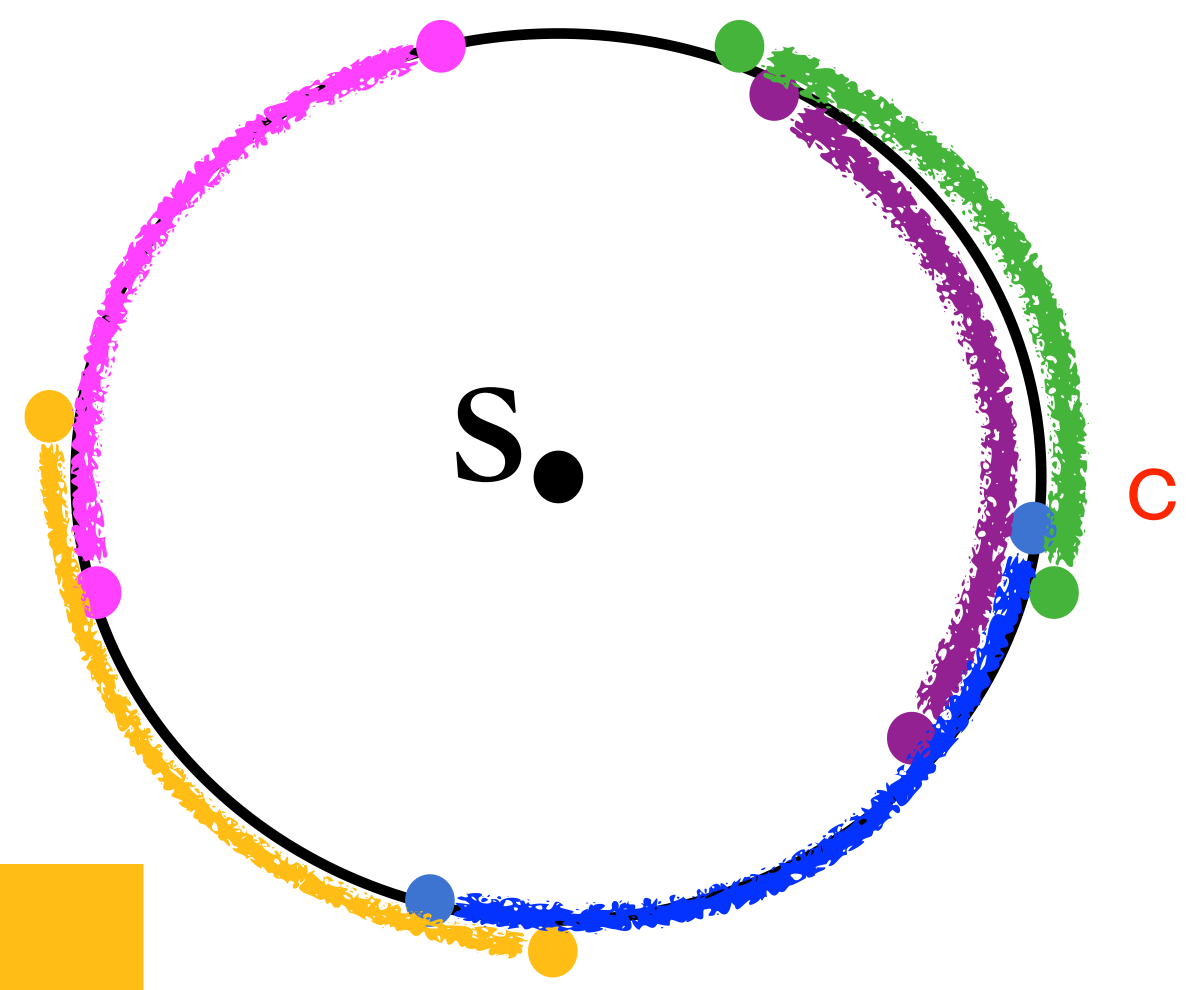
count - -



Arc Piercing Problem

Input: n arcs on a cycle

Output: the point piercing most arcs.

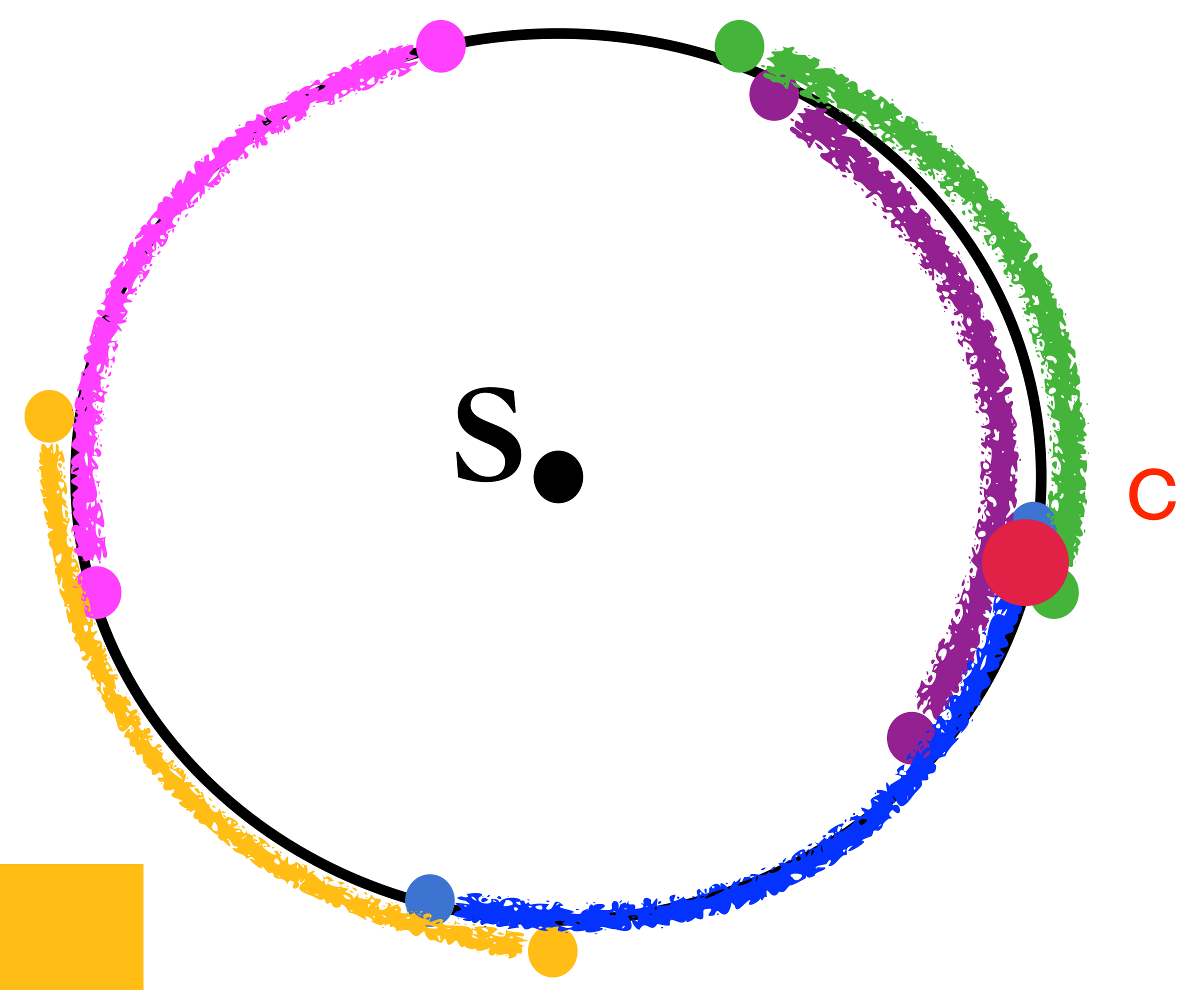


After O(nlog n) preprocessing work, the point piercing most arcs can be computed in O(n) time.

Arc Piercing Problem

Input: n arcs on a cycle

Output: the point piercing most arcs.



After O(nlog n) preprocessing work, the point piercing most arcs can be computed in O(n) time.

The Constrained Version

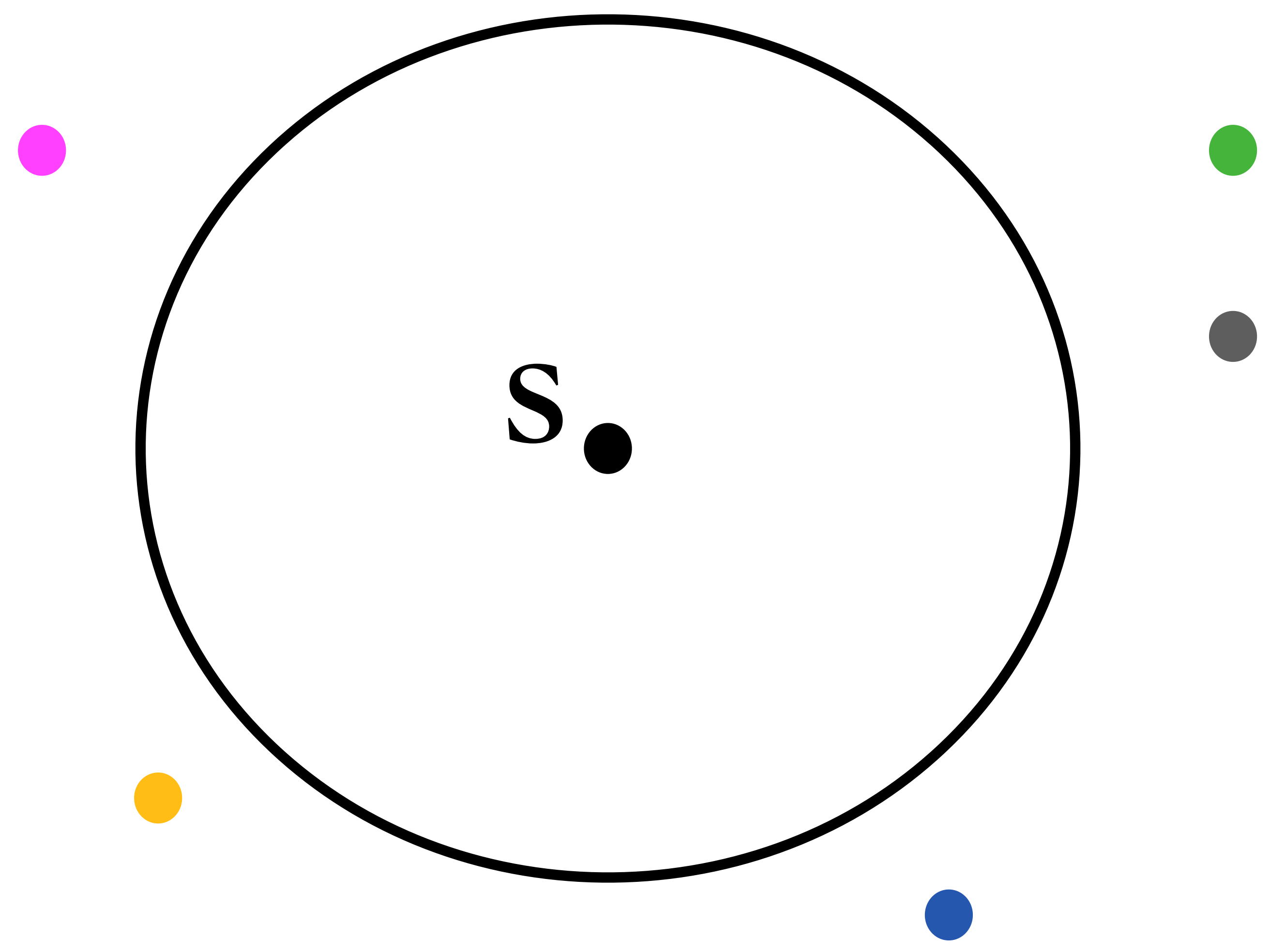
Input: constraint point s

n points

radius $r > 0$

Output: The center c of the cycle of radius r enclosing most points s.t c lying on the r-cycle of s.

Solved in O(n log n) time



The Constrained Version

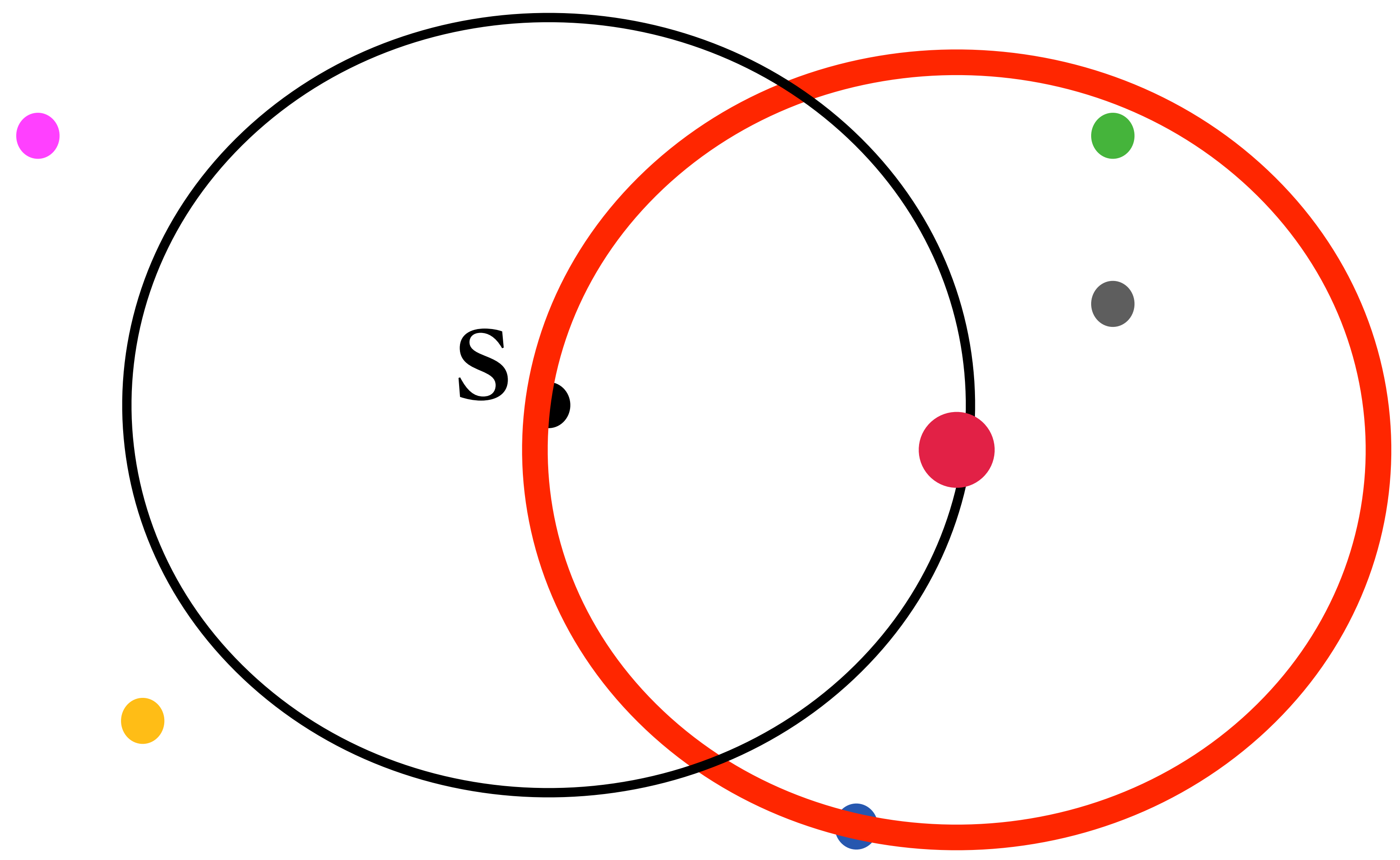
Input: constraint point s

n points

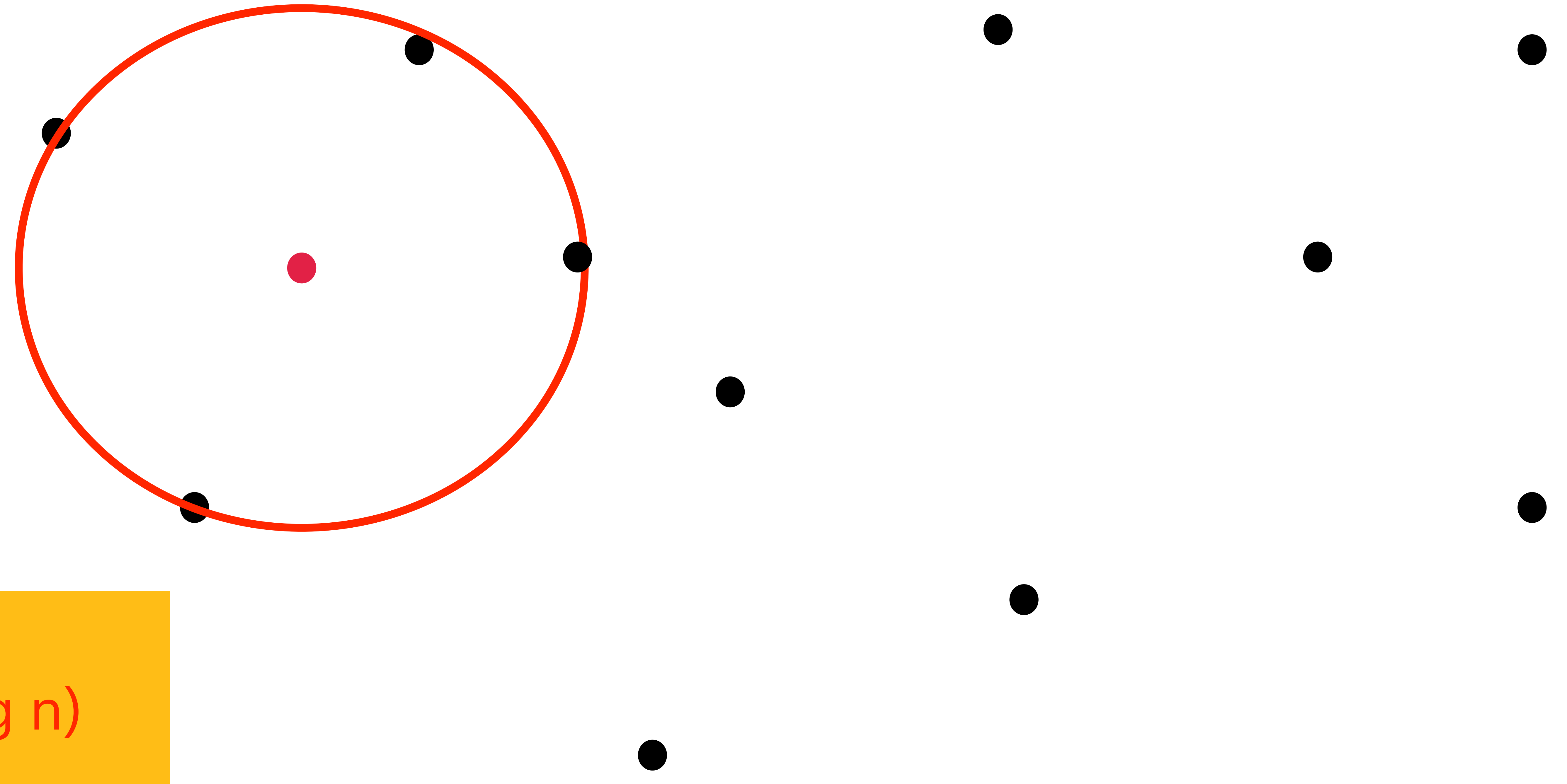
radius r > 0

Output: The center c of the cycle of radius r enclosing most points s.t c lying on the r-cycle of s.

Solved in O(n log n) time



Computing the r-Cycle Enclosing Most Points



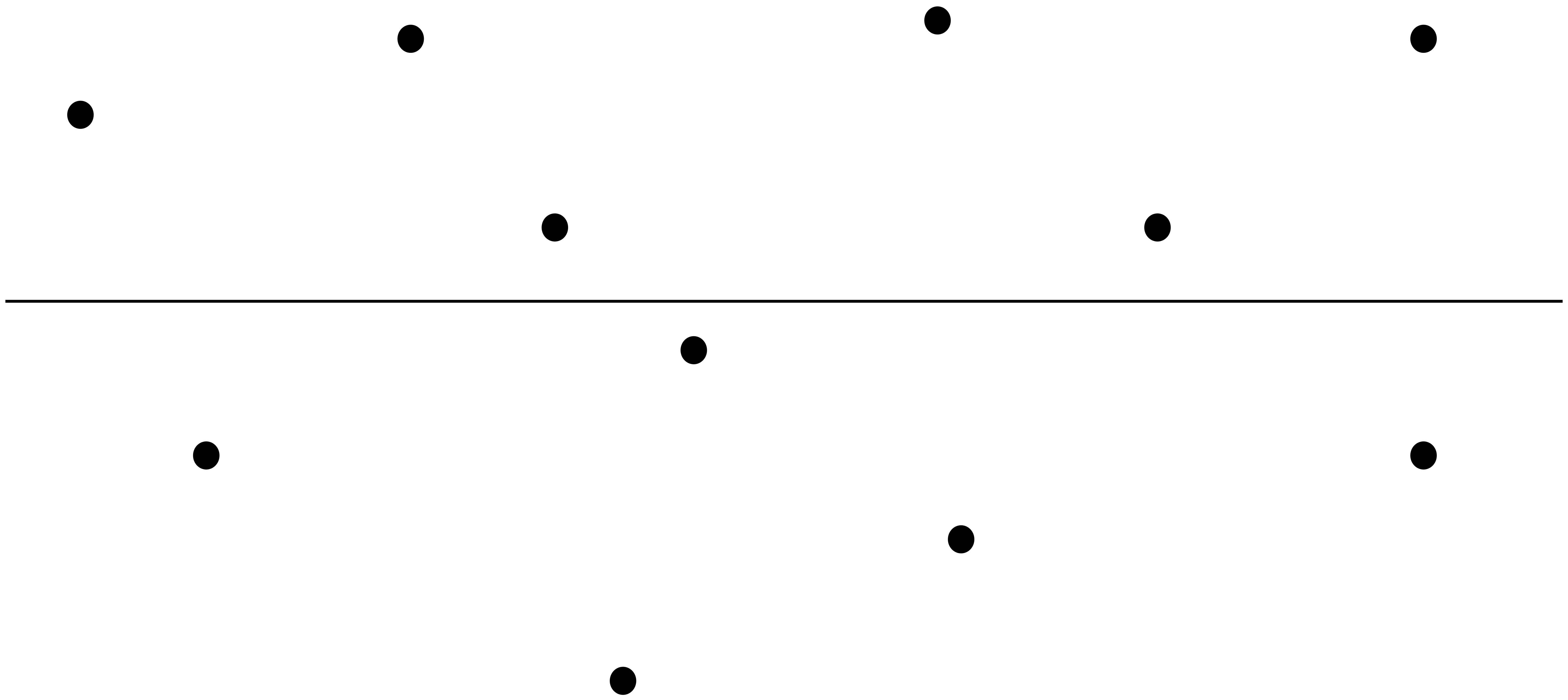
Ray Sweeping Algorithm ——— $O(n^2 \log n)$

For every input point p:

Compute the r-cycle enclosing most points centered at a point on its r-cycle.

The Line-Constrained Version

Computing Line-Constrained r-Cycle Enclosing Most Points



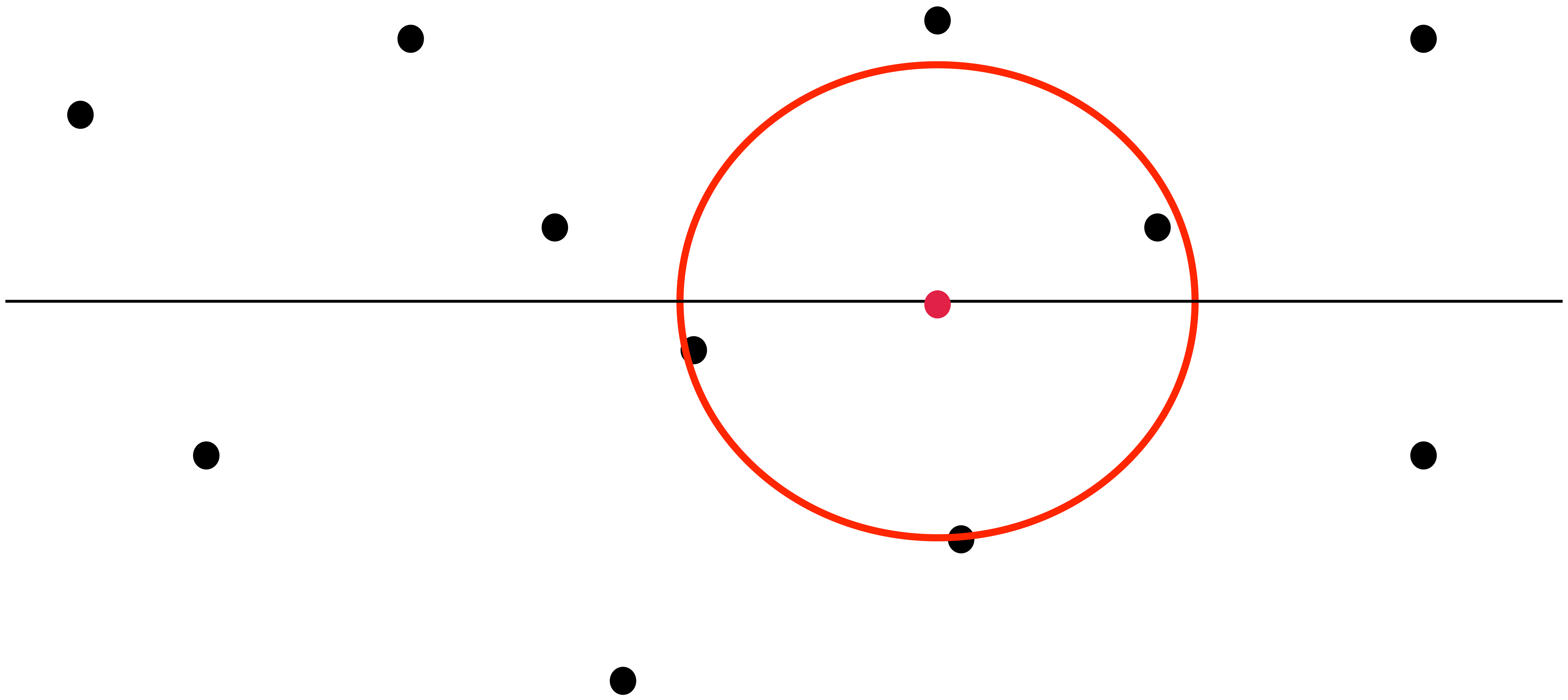
Input: n points

Radius r > 0

Line L

Output: The center c on L of the r-cycle enclosing most points.

Computing Line-Constrained r-Cycle Enclosing Most Points



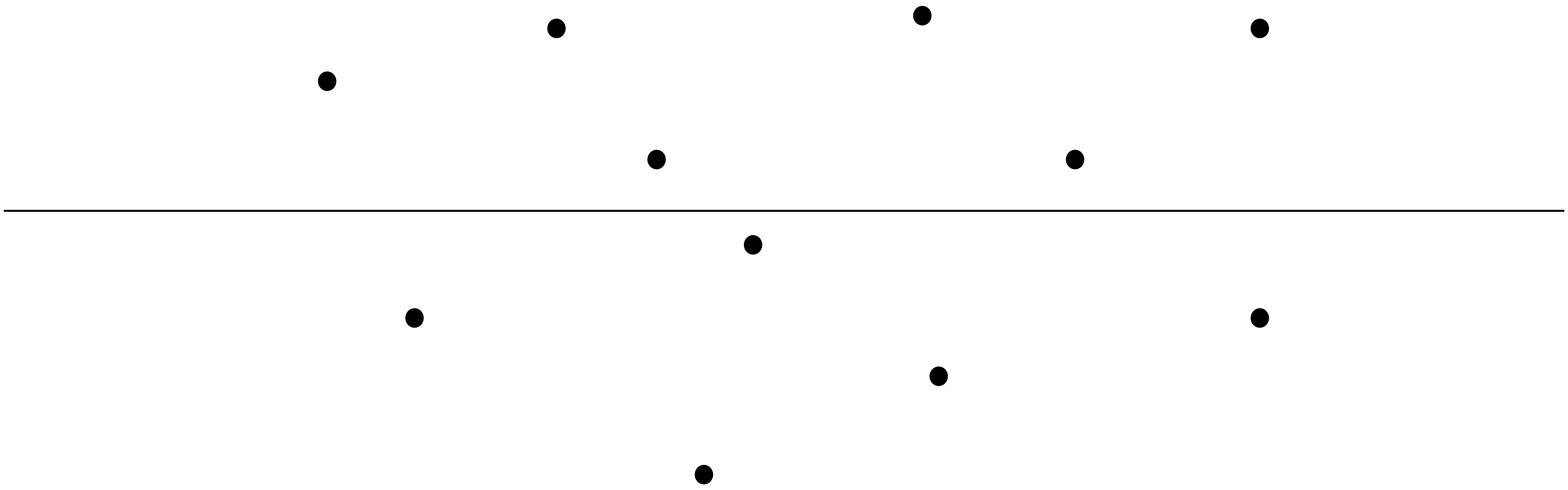
Input: n points

Radius r > 0

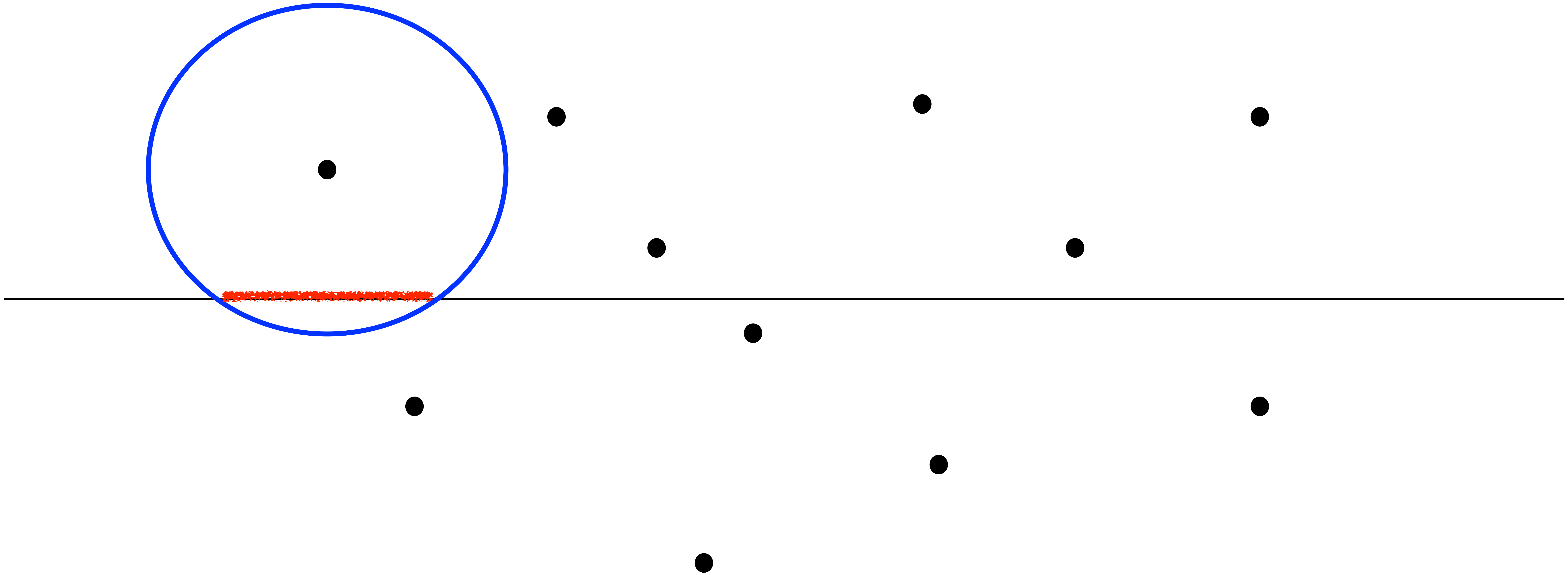
Line L

Output: The center c on L of the r-cycle enclosing most points.

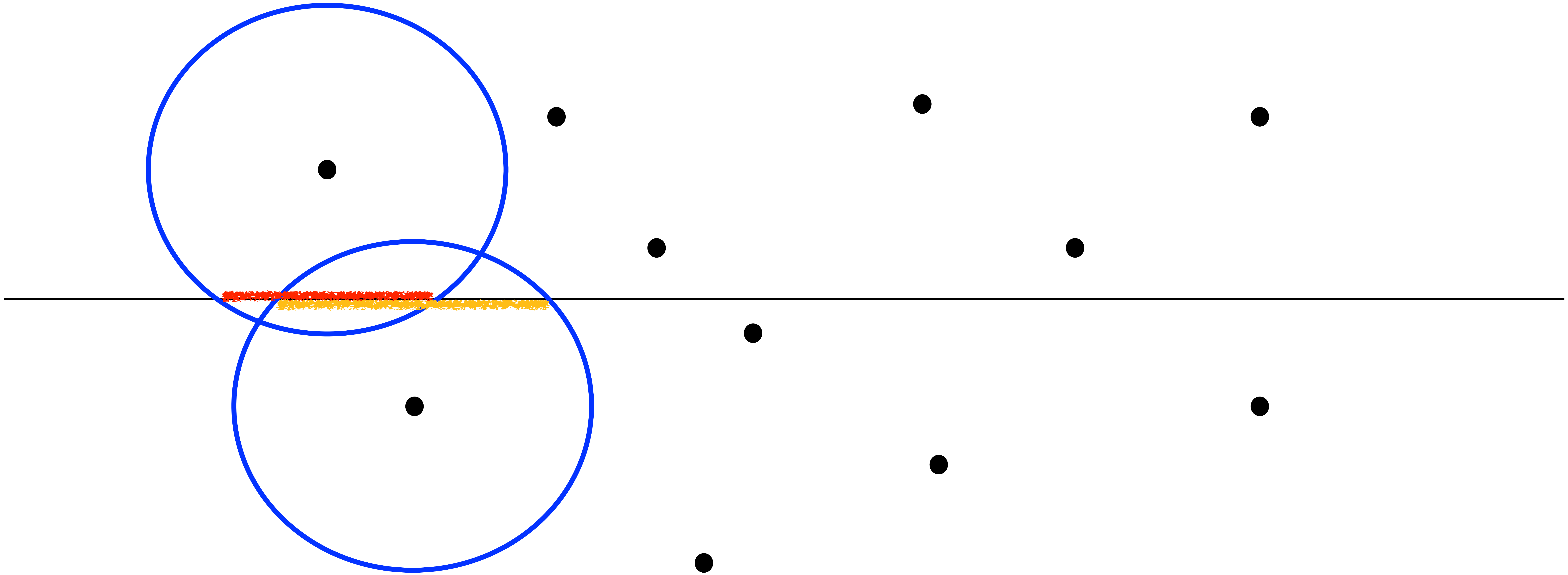
Computing the Line-Constrained r-Cycle Enclosing Most Points



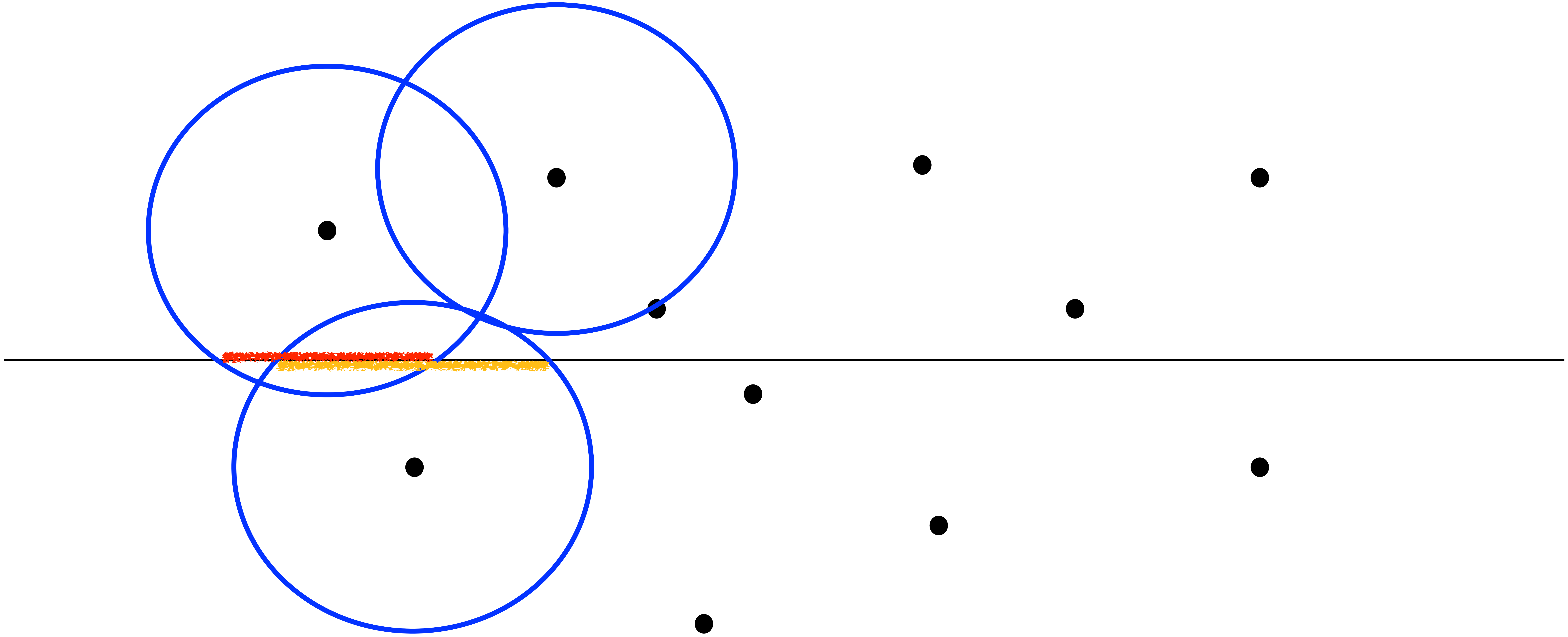
Computing the Line-Constrained r-Cycle Enclosing Most Points



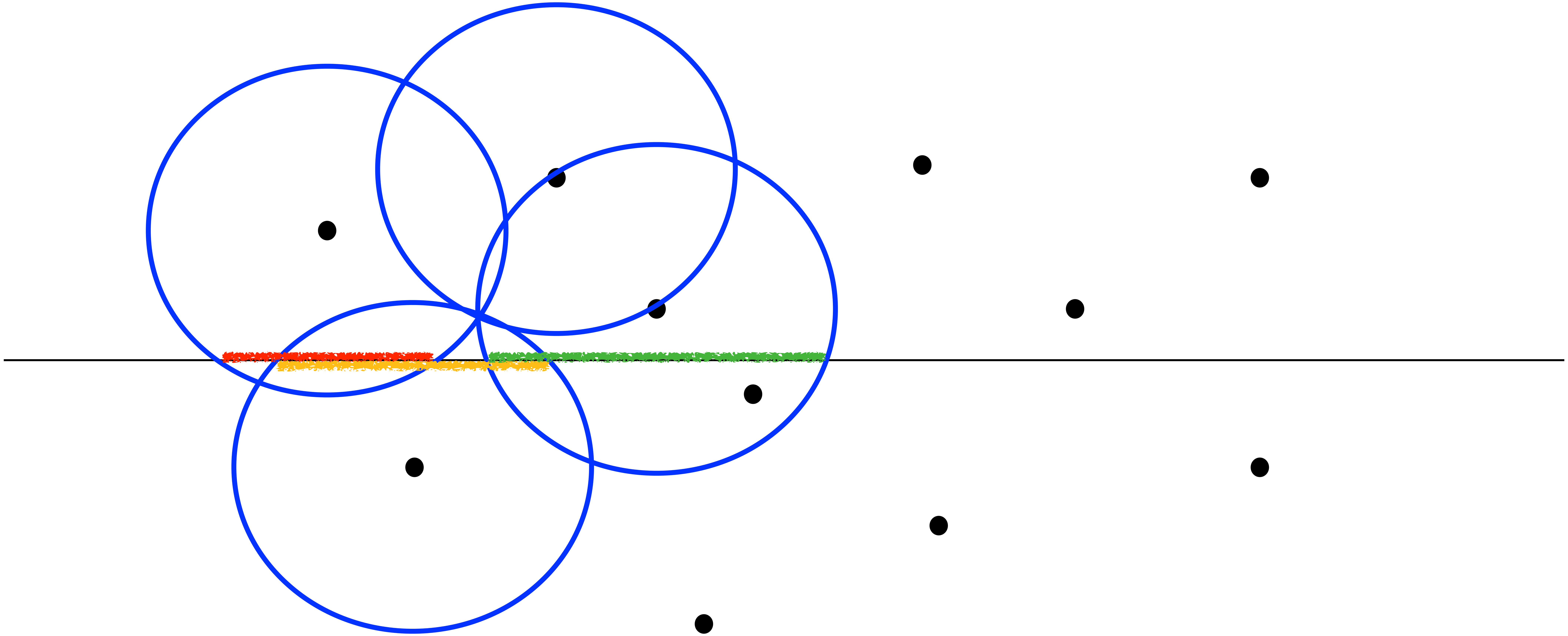
Computing the Line-Constrained r-Cycle Enclosing Most Points



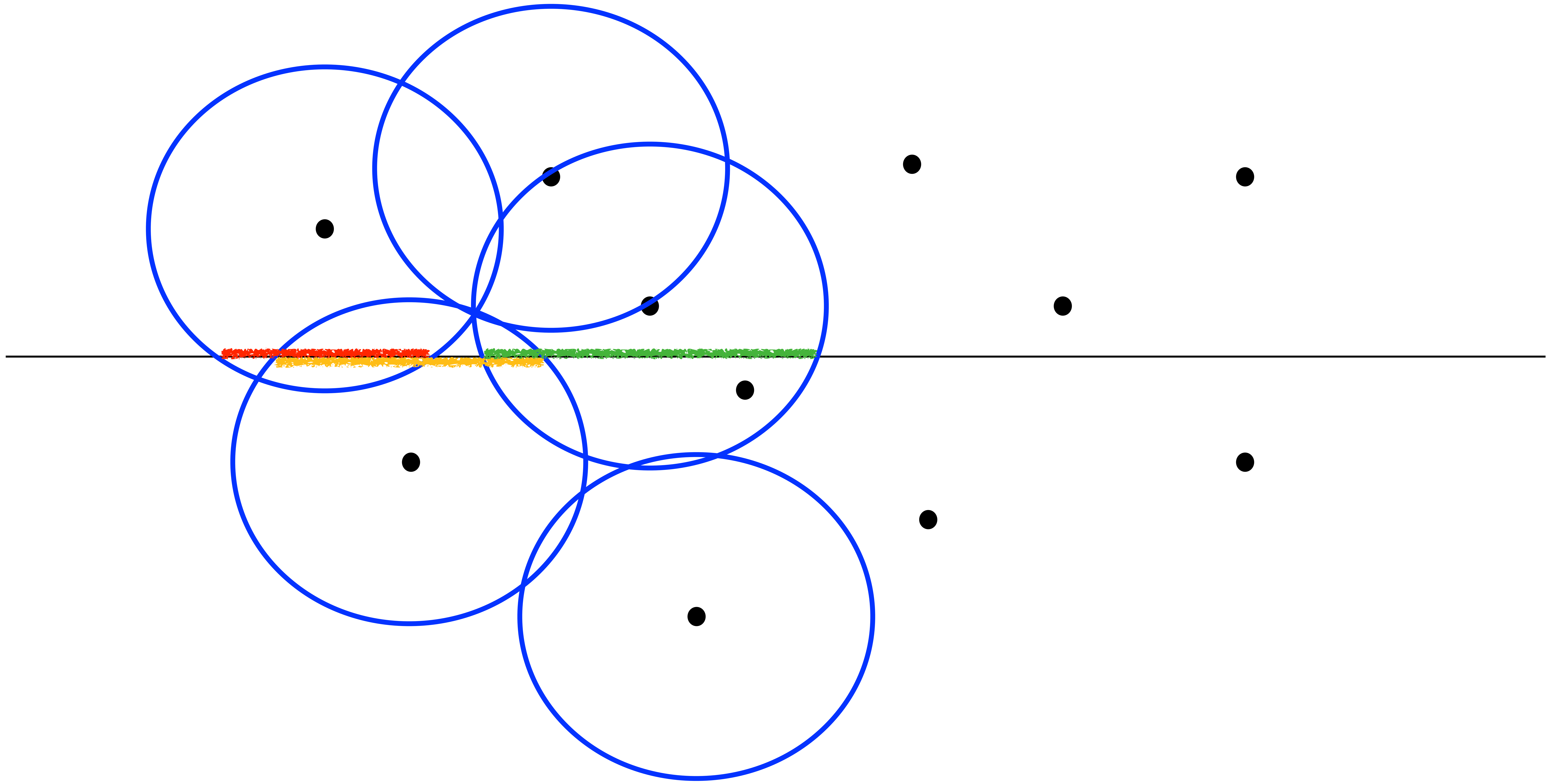
Computing the Line-Constrained r-Cycle Enclosing Most Points



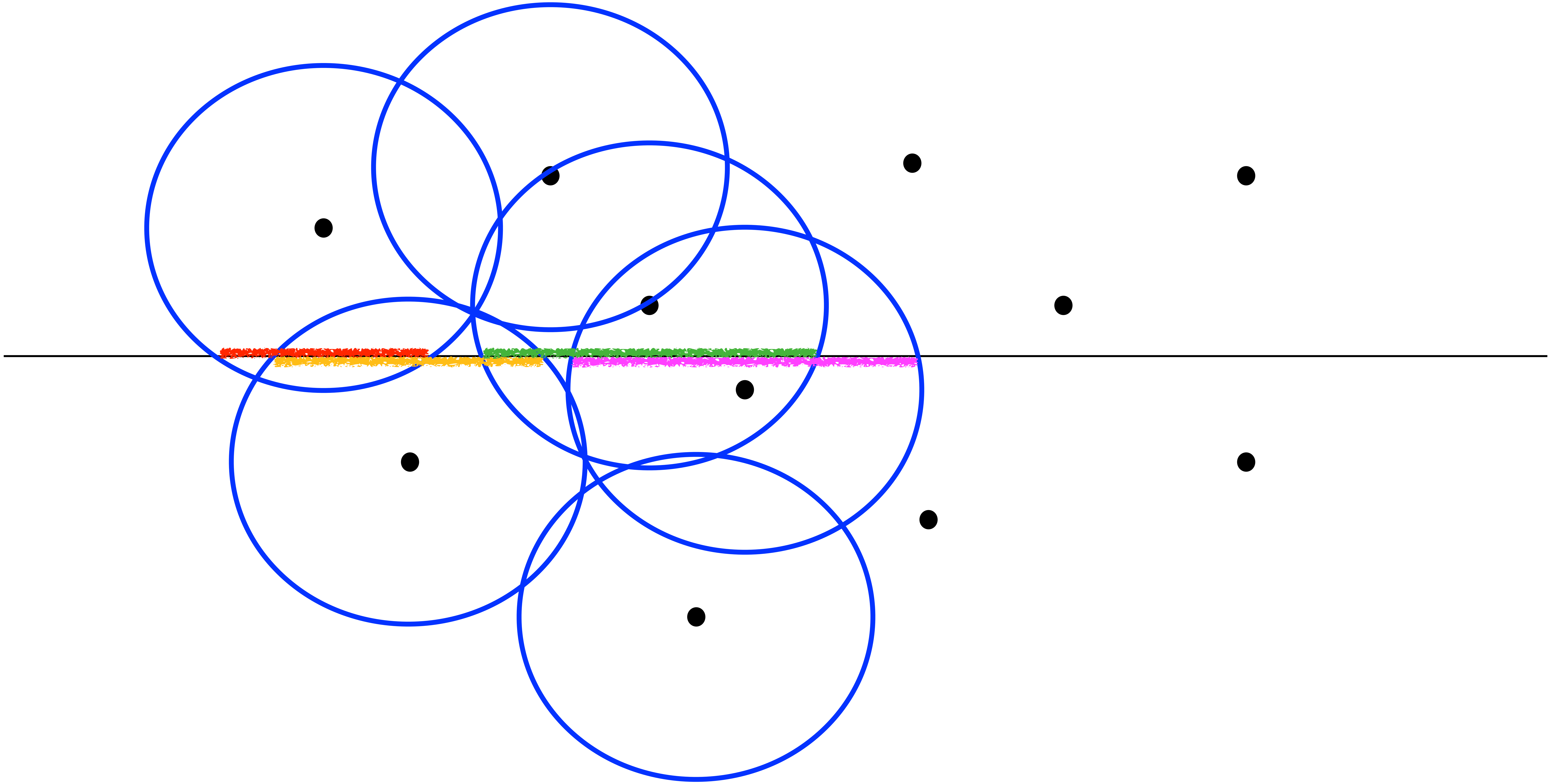
Computing the Line-Constrained r-Cycle Enclosing Most Points



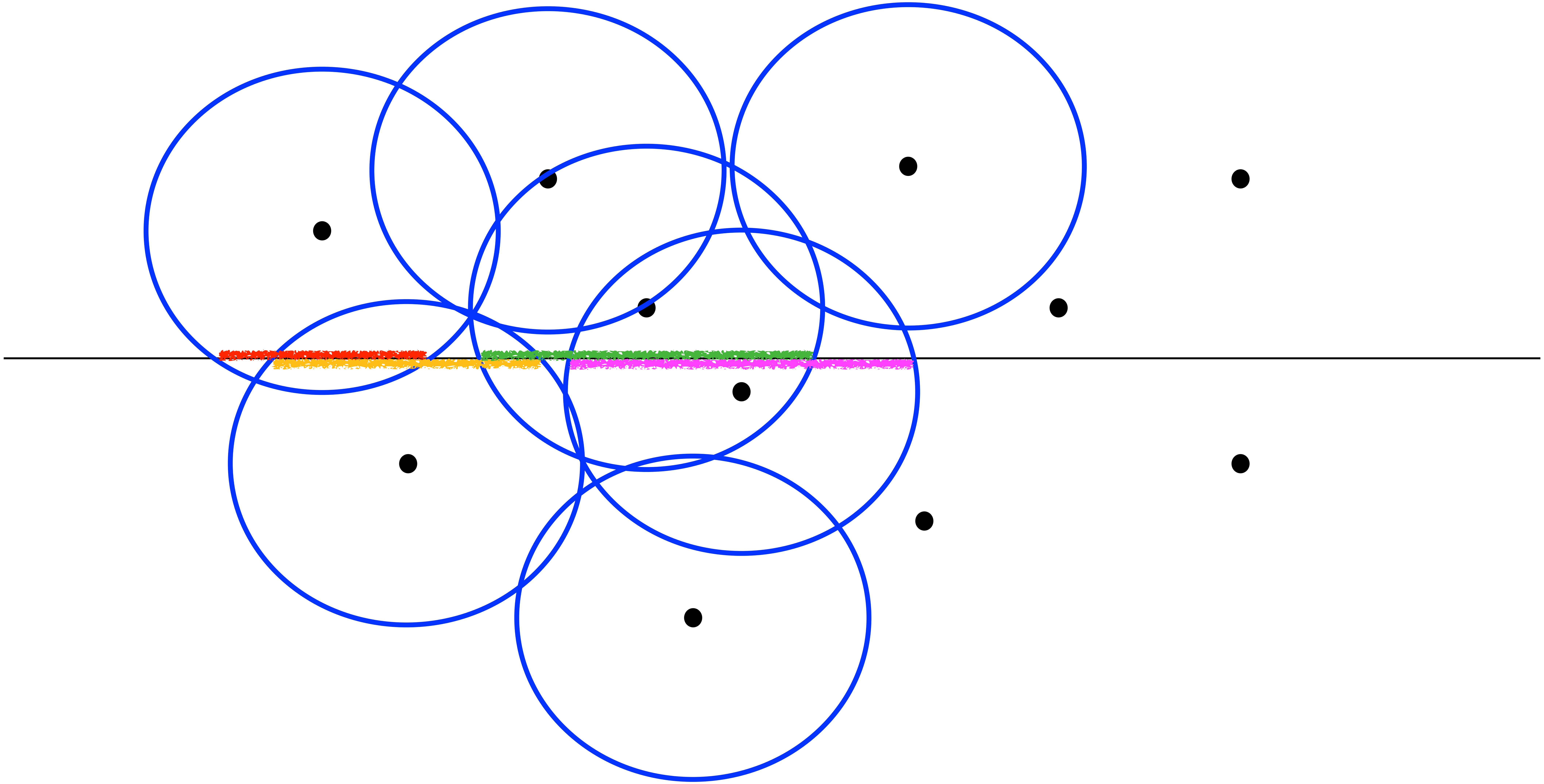
Computing the Line-Constrained r-Cycle Enclosing Most Points



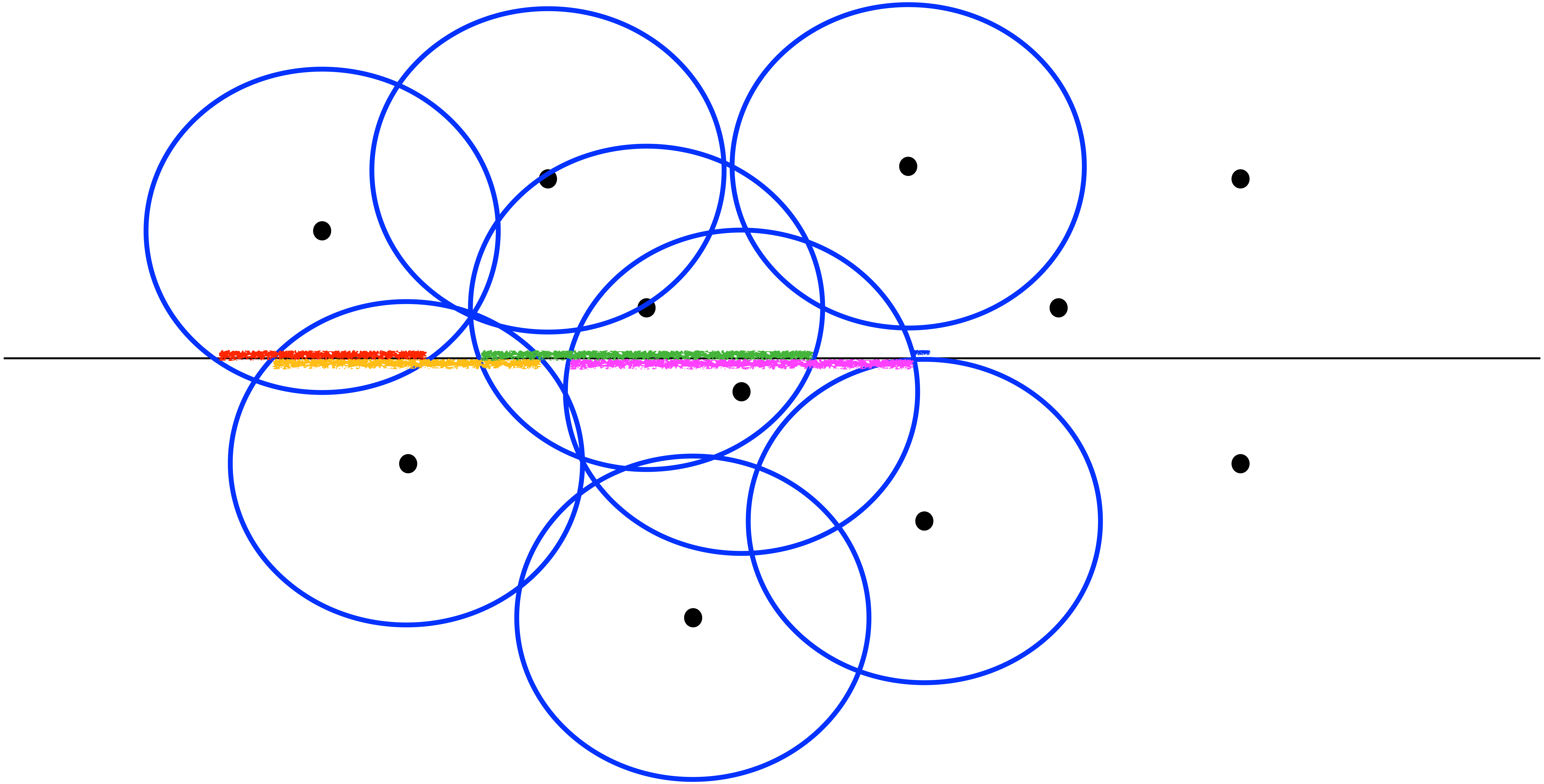
Computing the Line-Constrained r-Cycle Enclosing Most Points



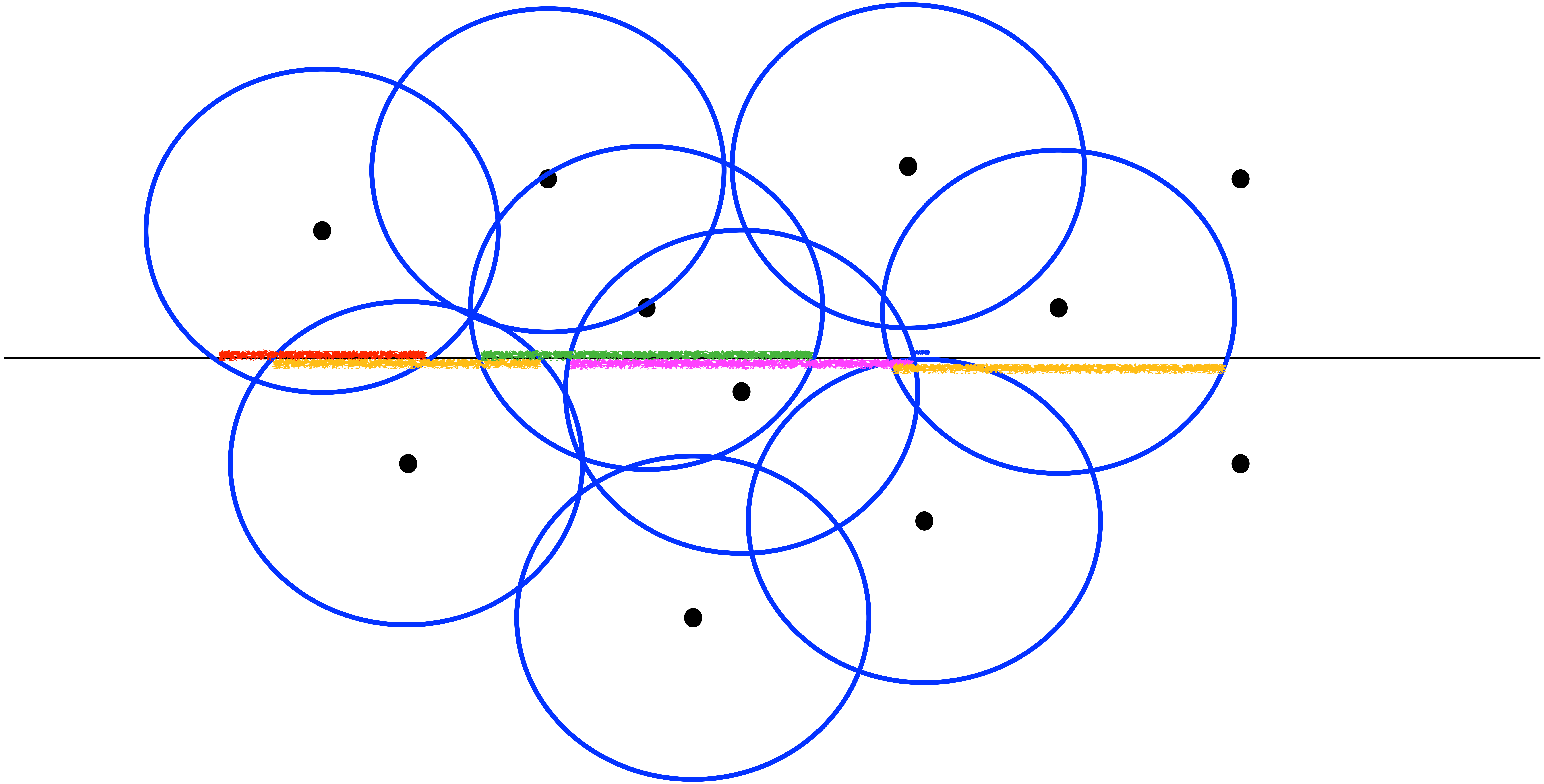
Computing the Line-Constrained r-Cycle Enclosing Most Points



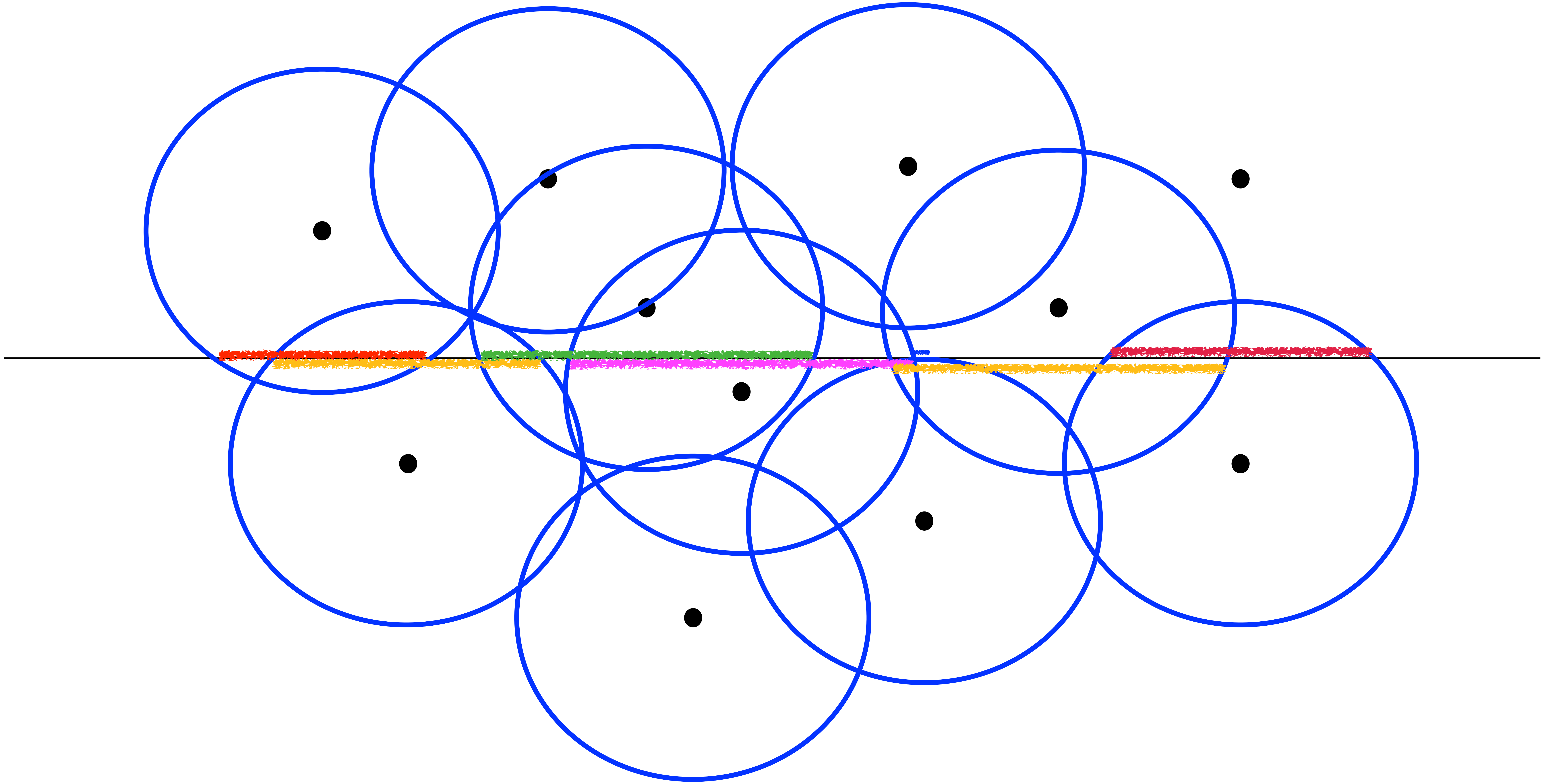
Computing the Line-Constrained r-Cycle Enclosing Most Points



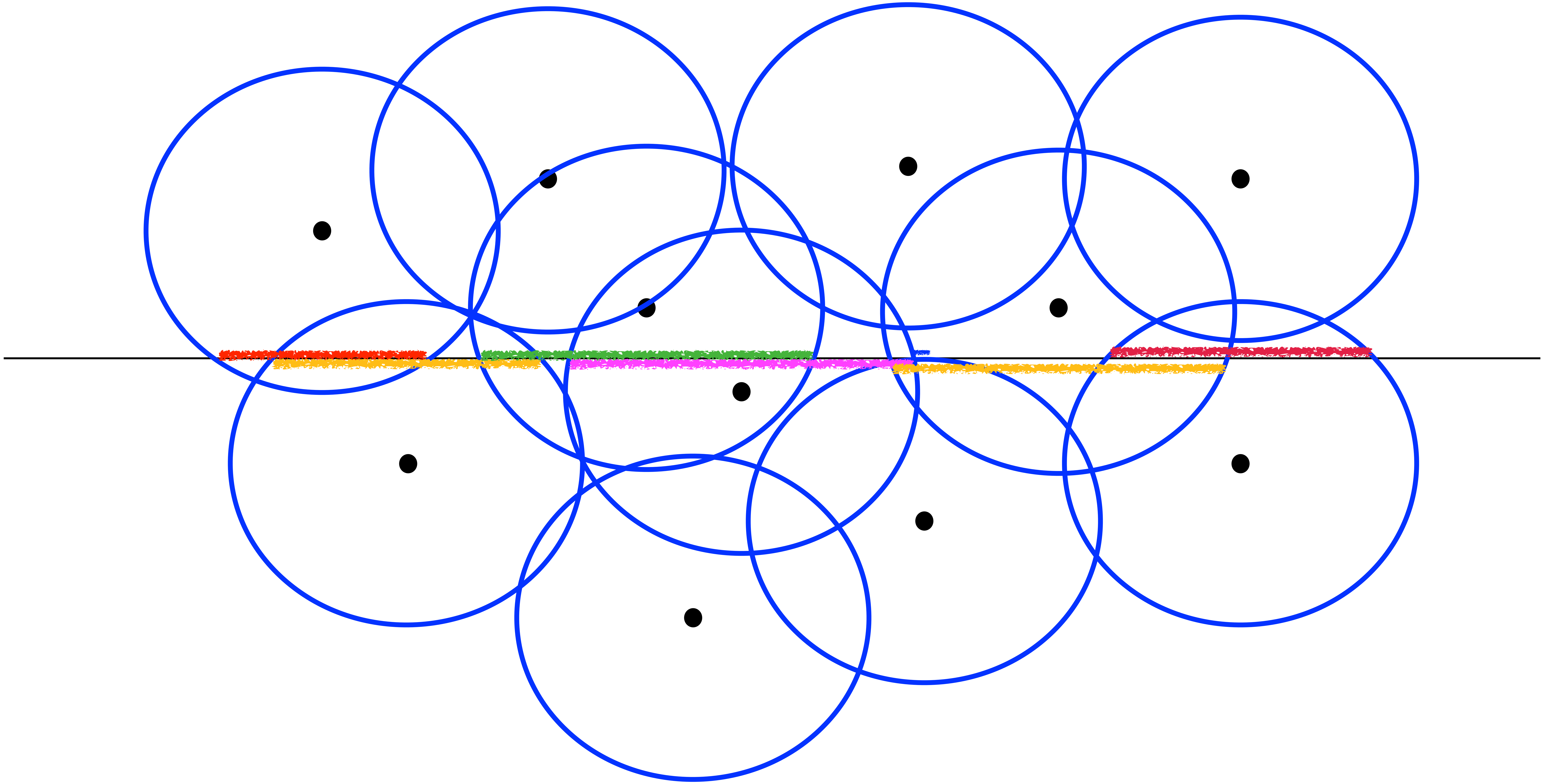
Computing the Line-Constrained r-Cycle Enclosing Most Points



Computing the Line-Constrained r-Cycle Enclosing Most Points



Computing the Line-Constrained r-Cycle Enclosing Most Points



Interval Piercing Problem

Input: n intervals on x-axis

Output: The point piercing most intervals

—- This point is the center c

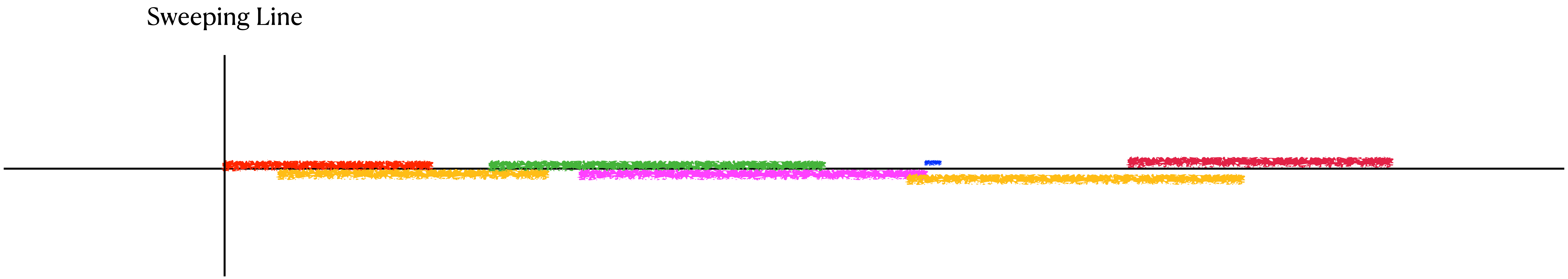
Our line sweeping algorithm computes c in O(n) time after O(nlog n) sorting.

Interval Piercing Problem

Input: n intervals on x-axis

Output: The point piercing most intervals

—- This point is the center c



Our line sweeping algorithm computes c in O(n) time after O(nlog n) sorting.

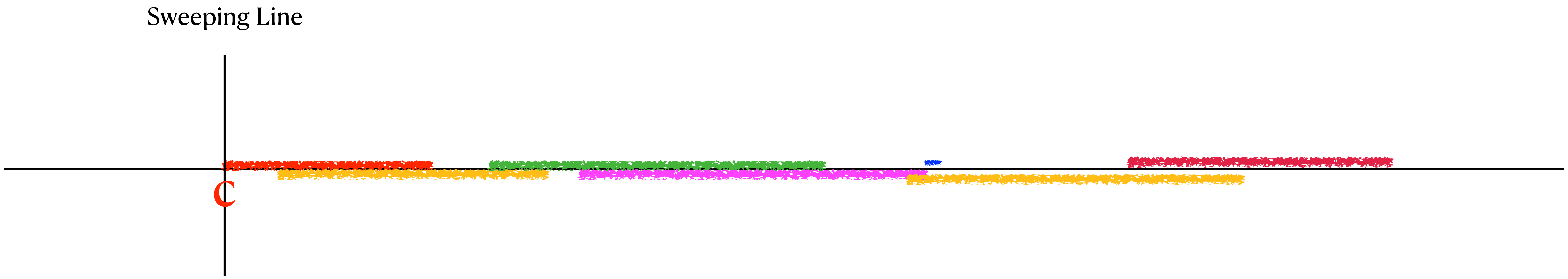
Interval Piercing Problem

Input: n intervals on x-axis

Output: The point piercing most intervals

—- This point is the center c

MaxCount = 1



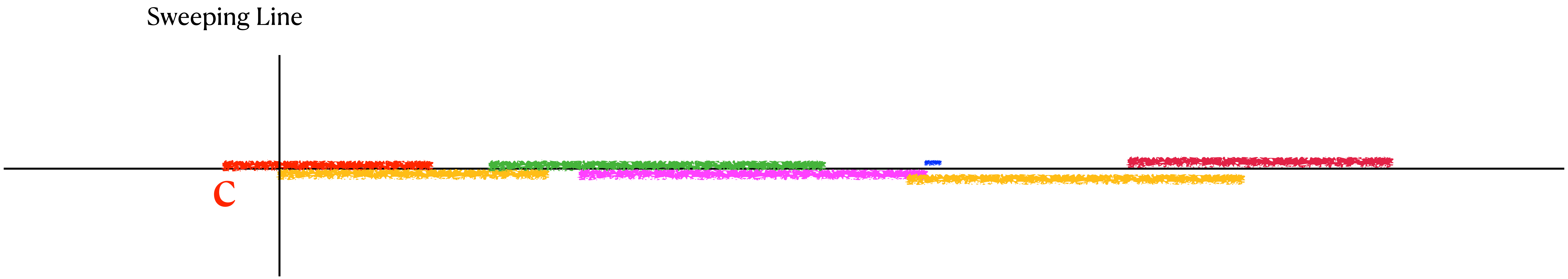
Our line sweeping algorithm computes c in O(n) time after O(nlog n) sorting.

Interval Piercing Problem

Input: n intervals on x-axis

Output: The point piercing most intervals

—- This point is the center c



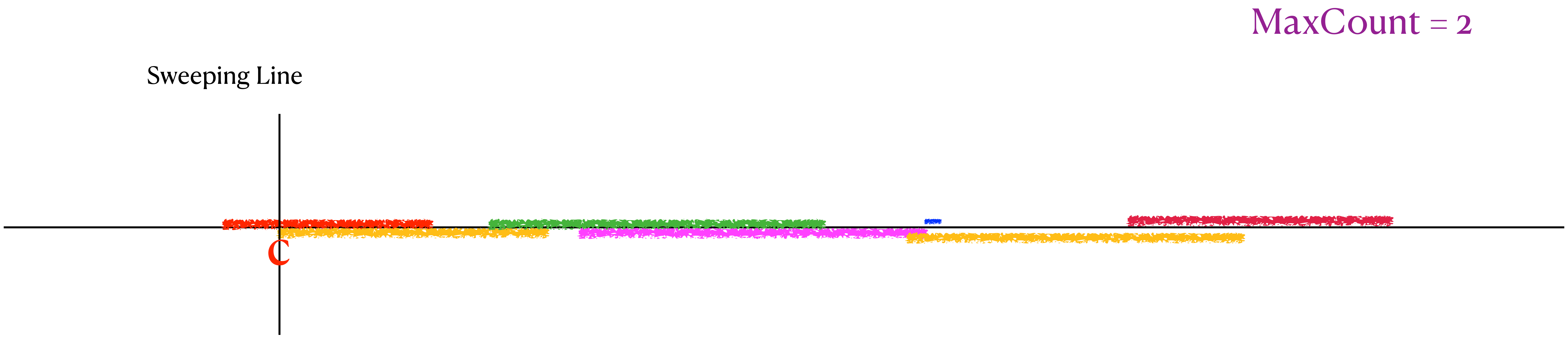
Our line sweeping algorithm computes c in O(n) time after O(nlog n) sorting.

Interval Piercing Problem

Input: n intervals on x-axis

Output: The point piercing most intervals

—- This point is the center c



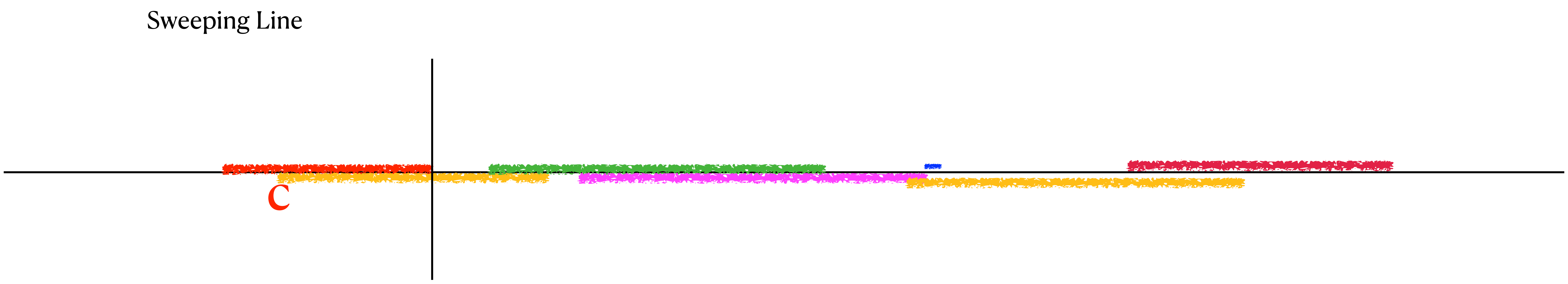
Our line sweeping algorithm computes c in O(n) time after O(nlog n) sorting.

Interval Piercing Problem

Input: n intervals on x-axis

Output: The point piercing most intervals

—- This point is the center c



Our line sweeping algorithm computes c in O(n) time after O(nlog n) sorting.

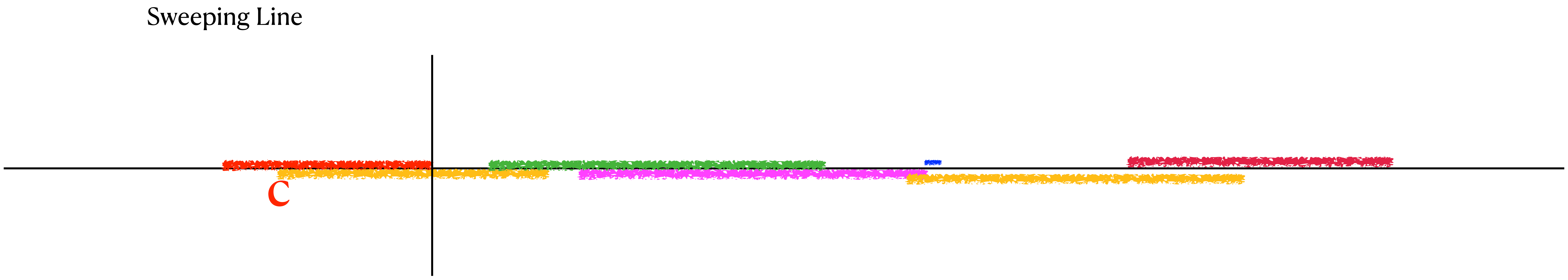
Interval Piercing Problem

Input: n intervals on x-axis

Output: The point piercing most intervals

—- This point is the center c

MaxCount = 2



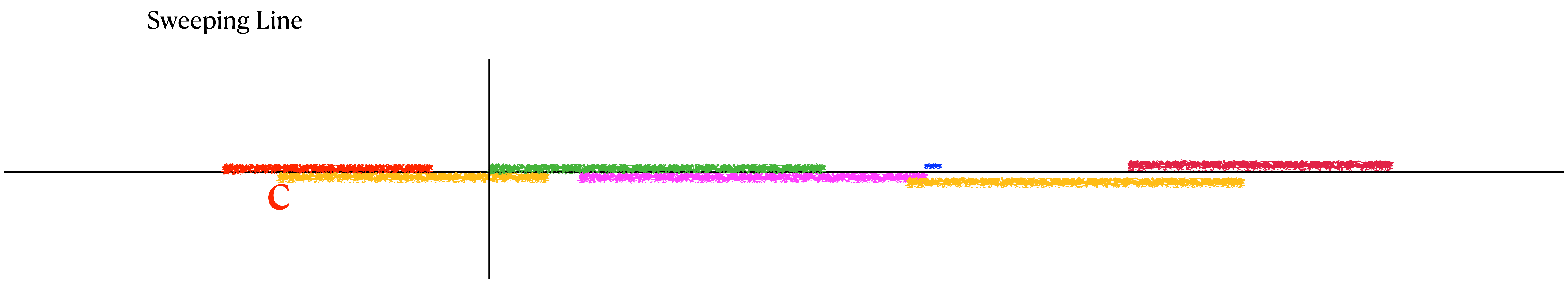
Our line sweeping algorithm computes c in O(n) time after O(nlog n) sorting.

Interval Piercing Problem

Input: n intervals on x-axis

Output: The point piercing most intervals

—- This point is the center c



Our line sweeping algorithm computes c in O(n) time after O(nlog n) sorting.

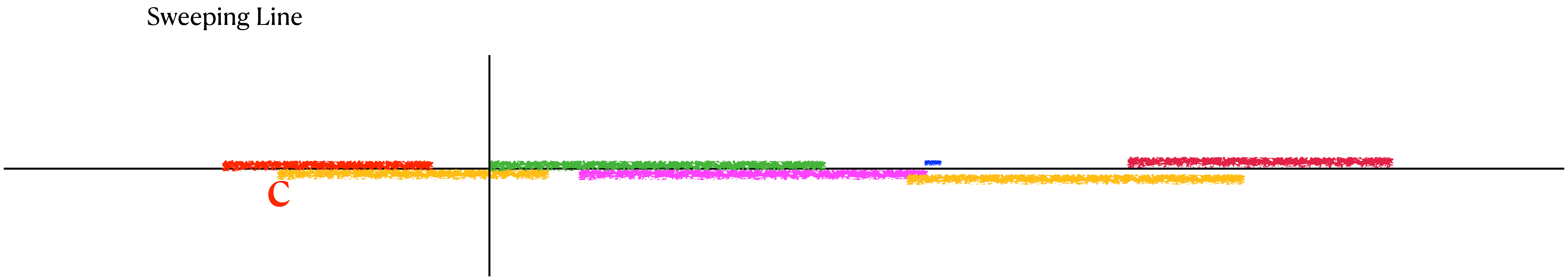
Interval Piercing Problem

Input: n intervals on x-axis

Output: The point piercing most intervals

—- This point is the center c

MaxCount = 2



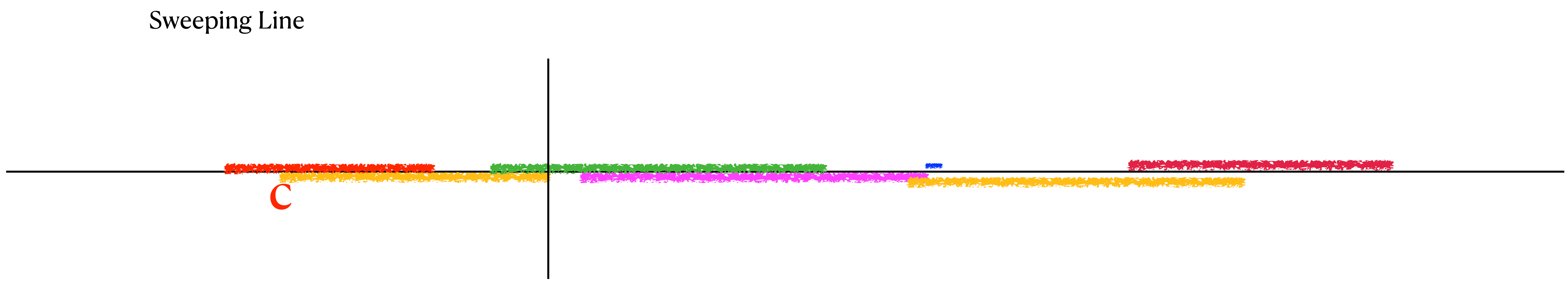
Our line sweeping algorithm computes c in O(n) time after O(nlog n) sorting.

Interval Piercing Problem

Input: n intervals on x-axis

Output: The point piercing most intervals

—- This point is the center c



Our line sweeping algorithm computes c in O(n) time after O(nlog n) sorting.

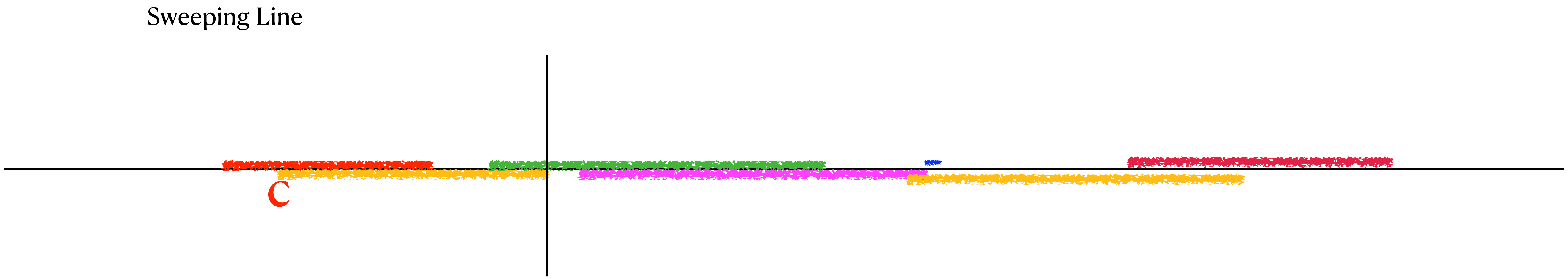
Interval Piercing Problem

Input: n intervals on x-axis

Output: The point piercing most intervals

—- This point is the center c

MaxCount = 2



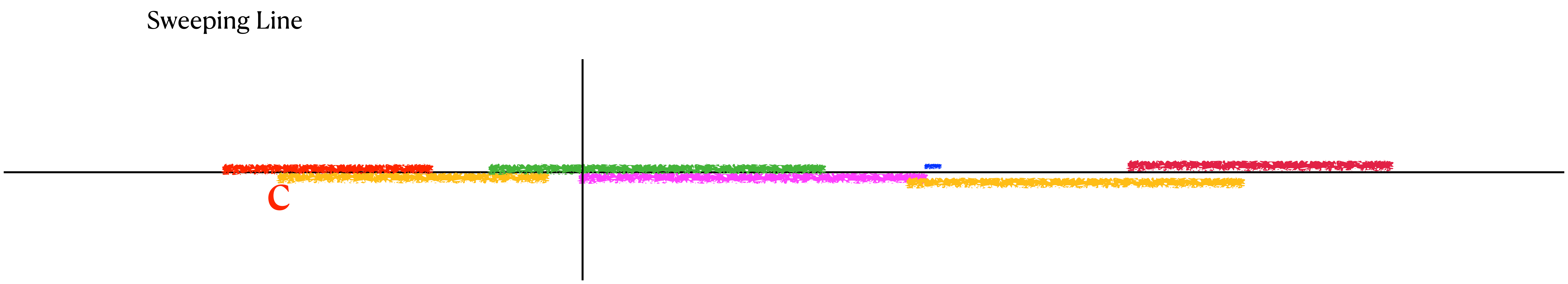
Our line sweeping algorithm computes c in O(n) time after O(nlog n) sorting.

Interval Piercing Problem

Input: n intervals on x-axis

Output: The point piercing most intervals

—- This point is the center c



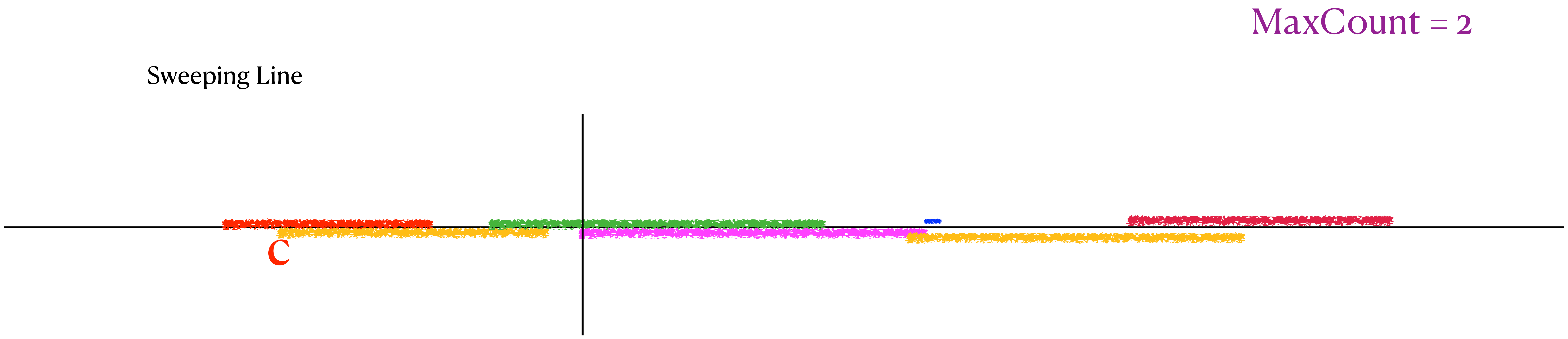
Our line sweeping algorithm computes c in O(n) time after O(nlog n) sorting.

Interval Piercing Problem

Input: n intervals on x-axis

Output: The point piercing most intervals

—- This point is the center c



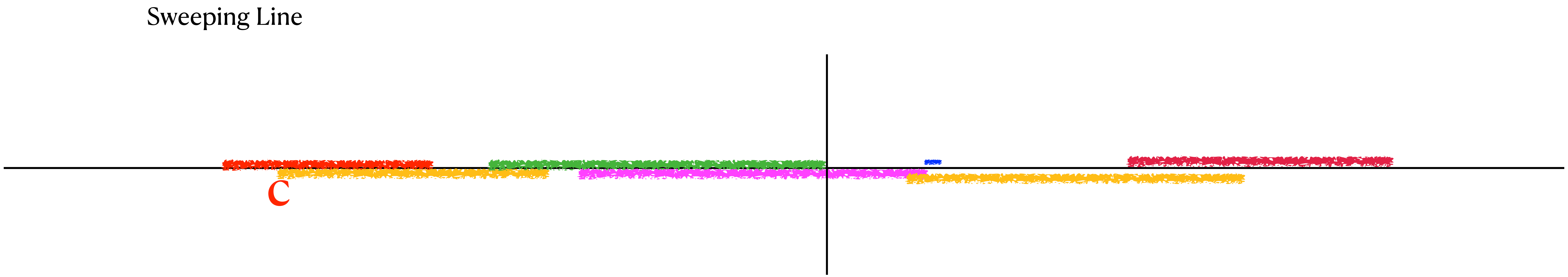
Our line sweeping algorithm computes c in O(n) time after O(nlog n) sorting.

Interval Piercing Problem

Input: n intervals on x-axis

Output: The point piercing most intervals

—- This point is the center c



Our line sweeping algorithm computes c in O(n) time after O(nlog n) sorting.

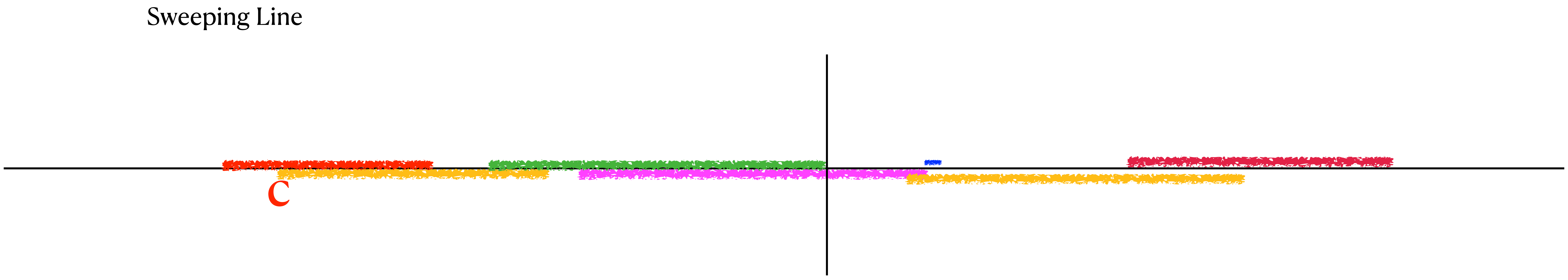
Interval Piercing Problem

Input: n intervals on x-axis

Output: The point piercing most intervals

—- This point is the center c

MaxCount = 2



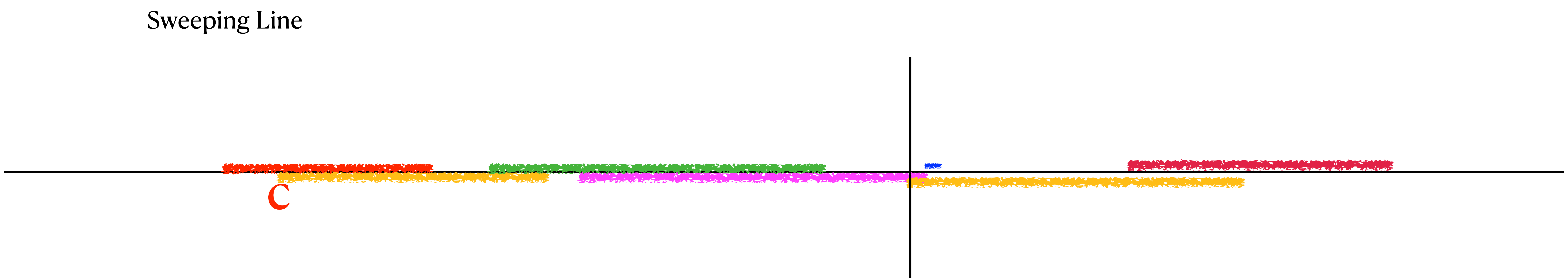
Our line sweeping algorithm computes c in O(n) time after O(nlog n) sorting.

Interval Piercing Problem

Input: n intervals on x-axis

Output: The point piercing most intervals

—- This point is the center c



Our line sweeping algorithm computes c in O(n) time after O(nlog n) sorting.

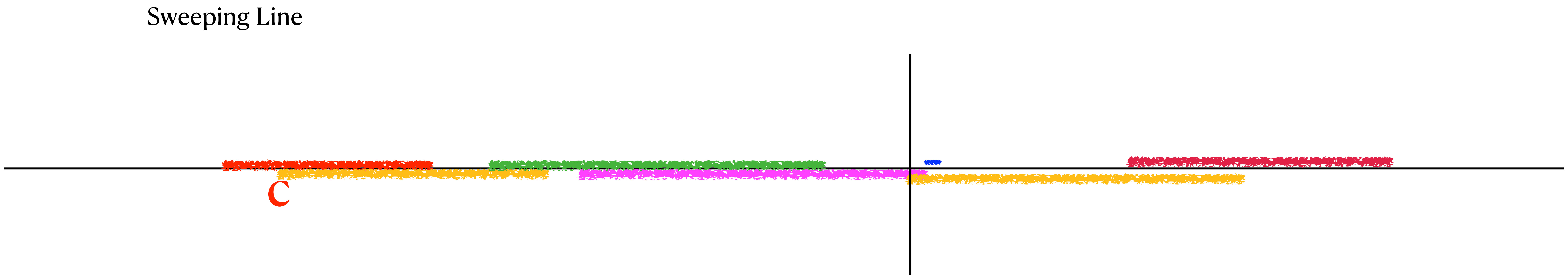
Interval Piercing Problem

Input: n intervals on x-axis

Output: The point piercing most intervals

—- This point is the center c

MaxCount = 2



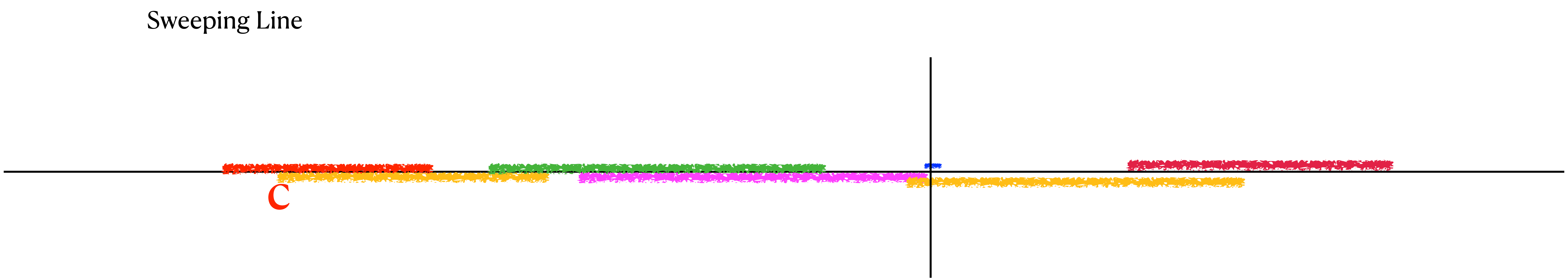
Our line sweeping algorithm computes c in O(n) time after O(nlog n) sorting.

Interval Piercing Problem

Input: n intervals on x-axis

Output: The point piercing most intervals

—- This point is the center c



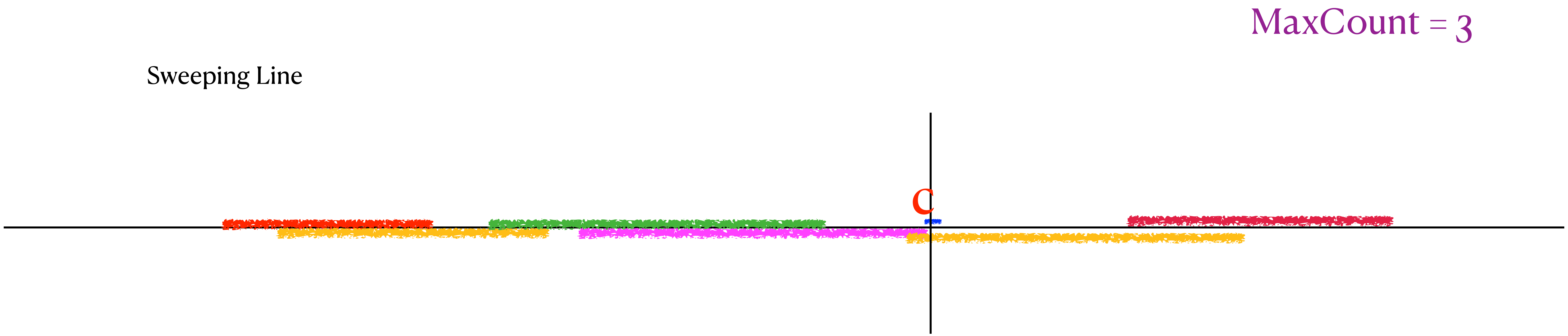
Our line sweeping algorithm computes c in O(n) time after O(nlog n) sorting.

Interval Piercing Problem

Input: n intervals on x-axis

Output: The point piercing most intervals

—- This point is the center c



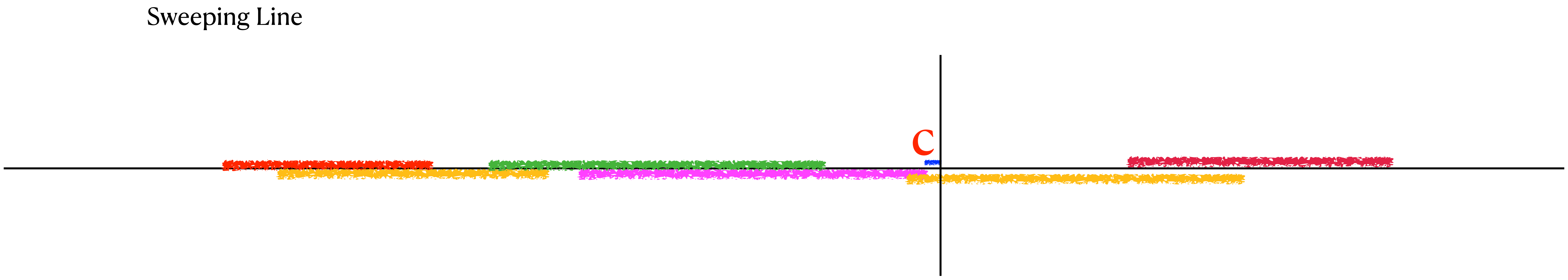
Our line sweeping algorithm computes c in O(n) time after O(nlog n) sorting.

Interval Piercing Problem

Input: n intervals on x-axis

Output: The point piercing most intervals

—- This point is the center c



Our line sweeping algorithm computes c in O(n) time after O(nlog n) sorting.

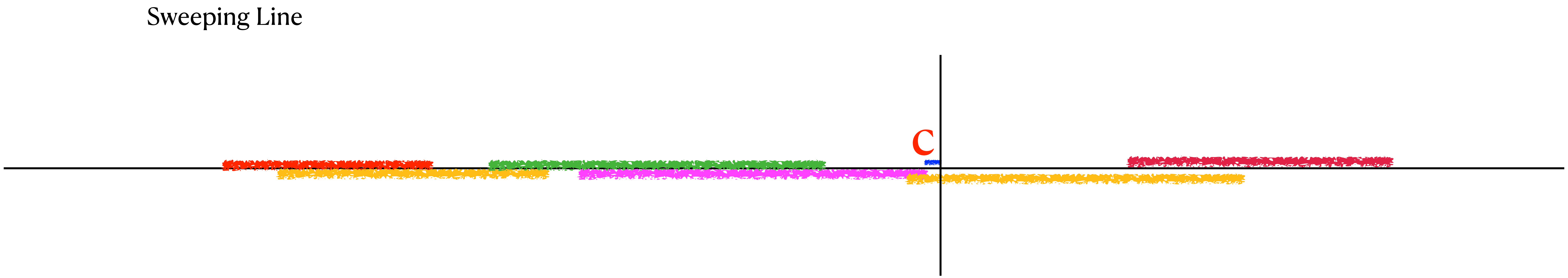
Interval Piercing Problem

Input: n intervals on x-axis

Output: The point piercing most intervals

—- This point is the center c

MaxCount = 3



Our line sweeping algorithm computes c in O(n) time after O(nlog n) sorting.

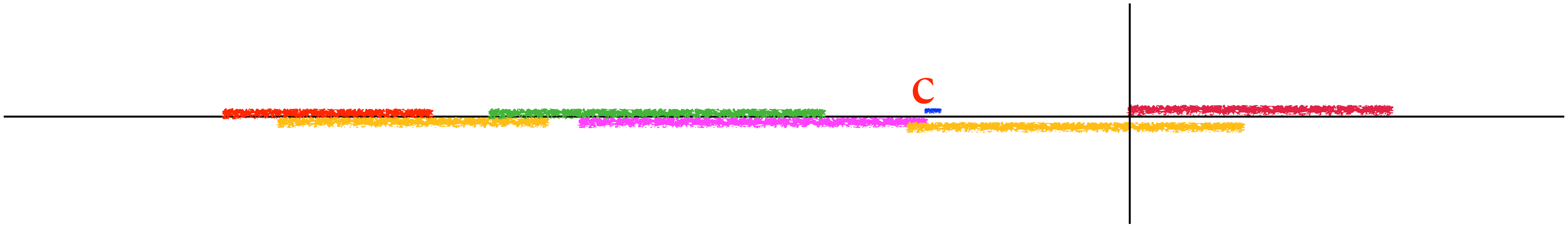
Interval Piercing Problem

Input: n intervals on x-axis

Output: The point piercing most intervals

—- This point is the center c

Sweeping Line



Our line sweeping algorithm computes c in O(n) time after O(nlog n) sorting.

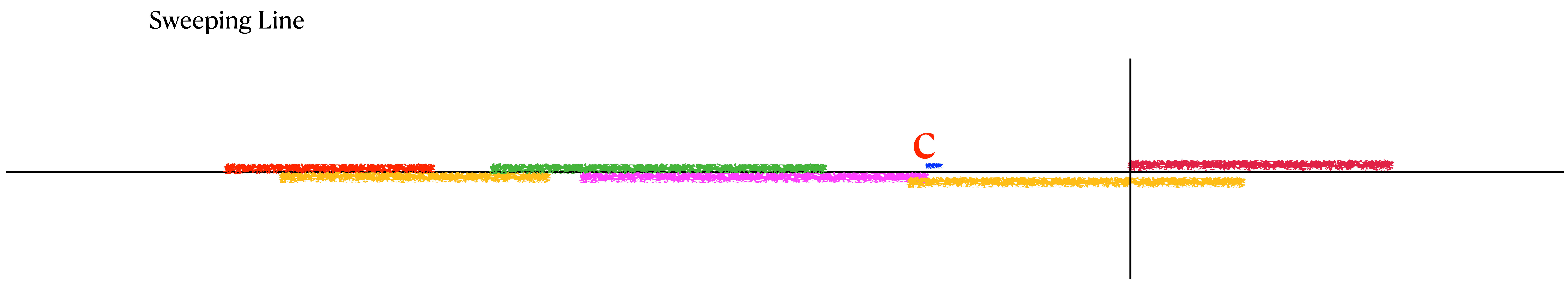
Interval Piercing Problem

Input: n intervals on x-axis

Output: The point piercing most intervals

—- This point is the center c

MaxCount = 3



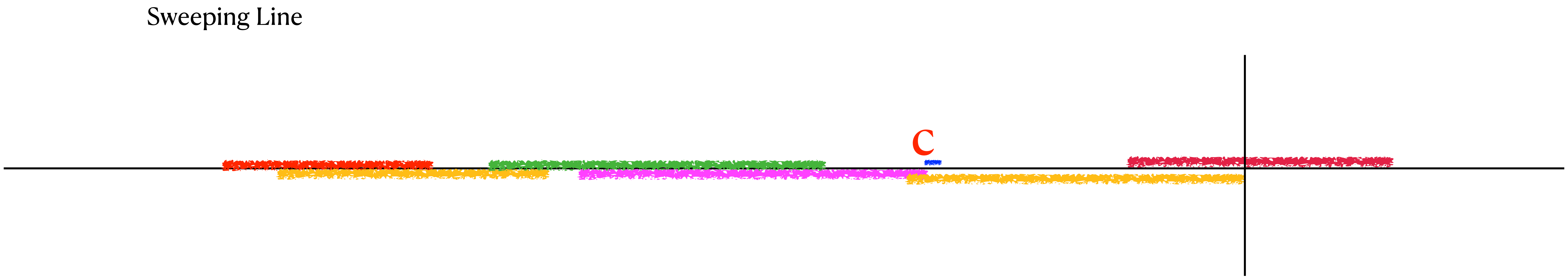
Our line sweeping algorithm computes c in O(n) time after O(nlog n) sorting.

Interval Piercing Problem

Input: n intervals on x-axis

Output: The point piercing most intervals

—- This point is the center c



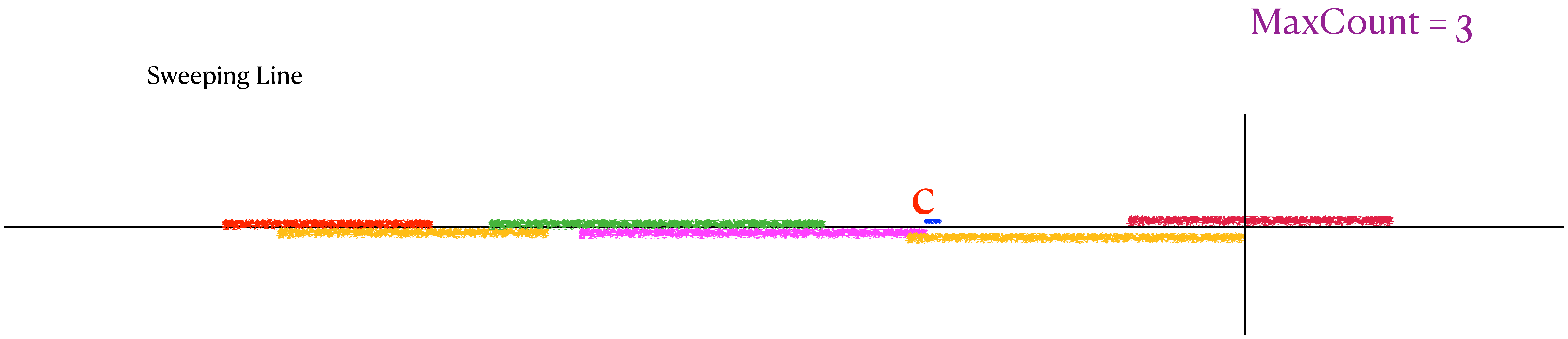
Our line sweeping algorithm computes c in O(n) time after O(nlog n) sorting.

Interval Piercing Problem

Input: n intervals on x-axis

Output: The point piercing most intervals

—- This point is the center c



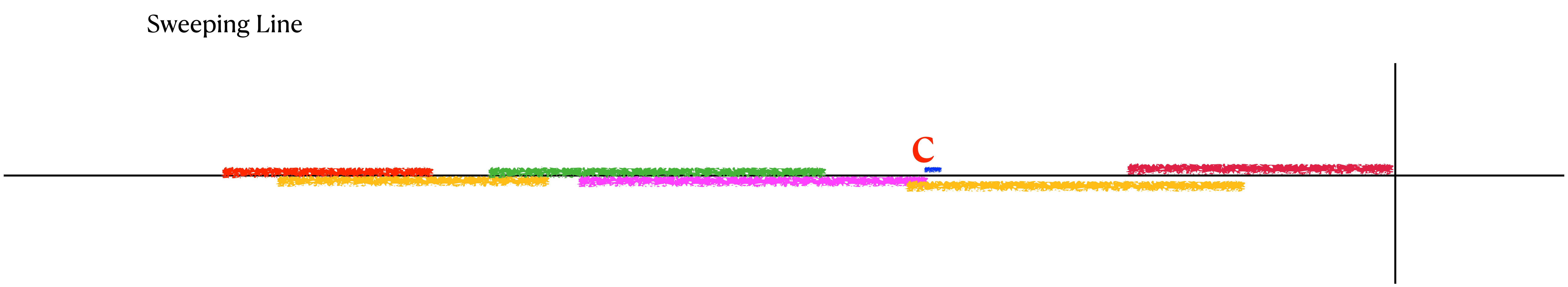
Our line sweeping algorithm computes c in O(n) time after O(nlog n) sorting.

Interval Piercing Problem

Input: n intervals on x-axis

Output: The point piercing most intervals

—- This point is the center c



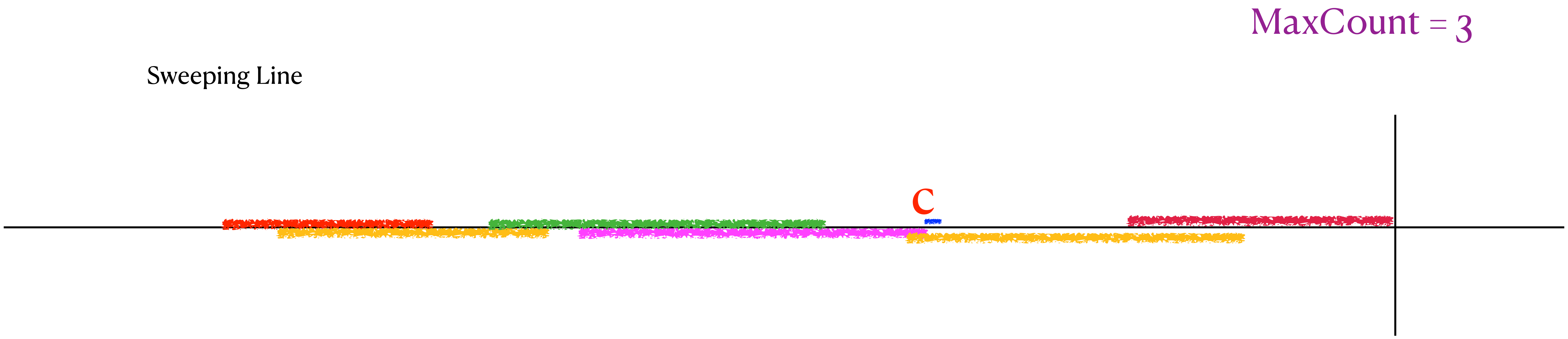
Our line sweeping algorithm computes c in O(n) time after O(nlog n) sorting.

Interval Piercing Problem

Input: n intervals on x-axis

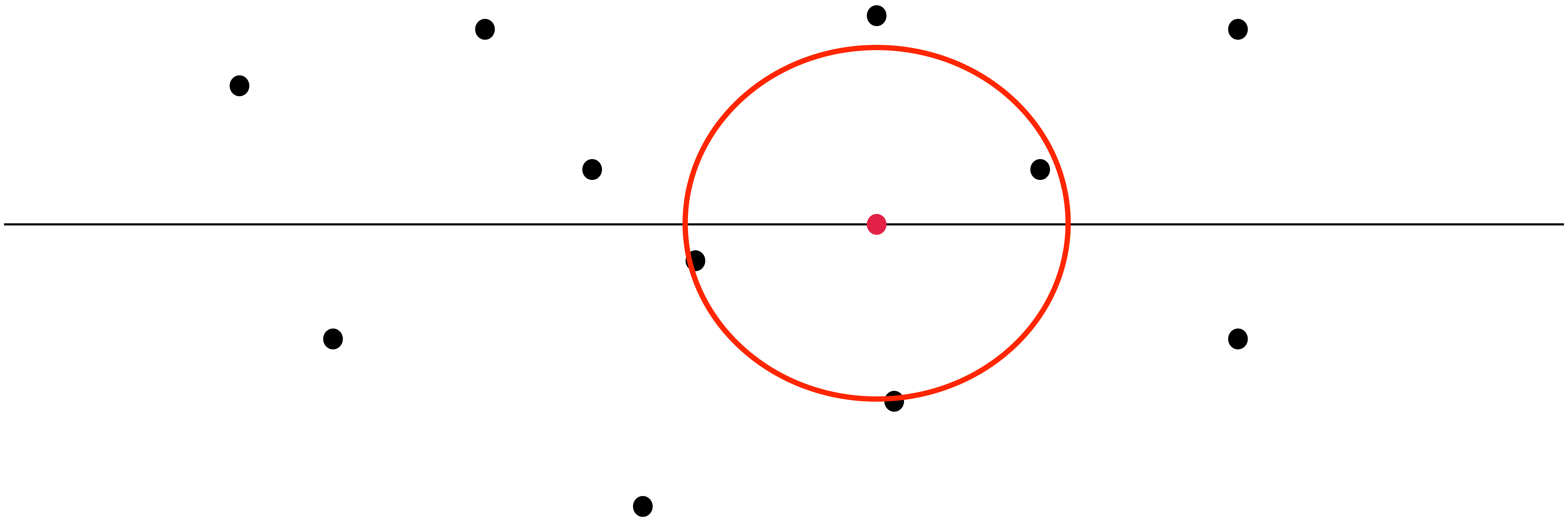
Output: The point piercing most intervals

—- This point is the center c



Our line sweeping algorithm computes c in O(n) time after O(nlog n) sorting.

Computing the Line-Constrained r-Cycle Enclosing Most Points



Our Algorithm computes c in O(n) time with O(nlog n) preprocessing work.

The One-Dimension Version

Computing the One-Dimensional r-Cycle Enclosing Most Points

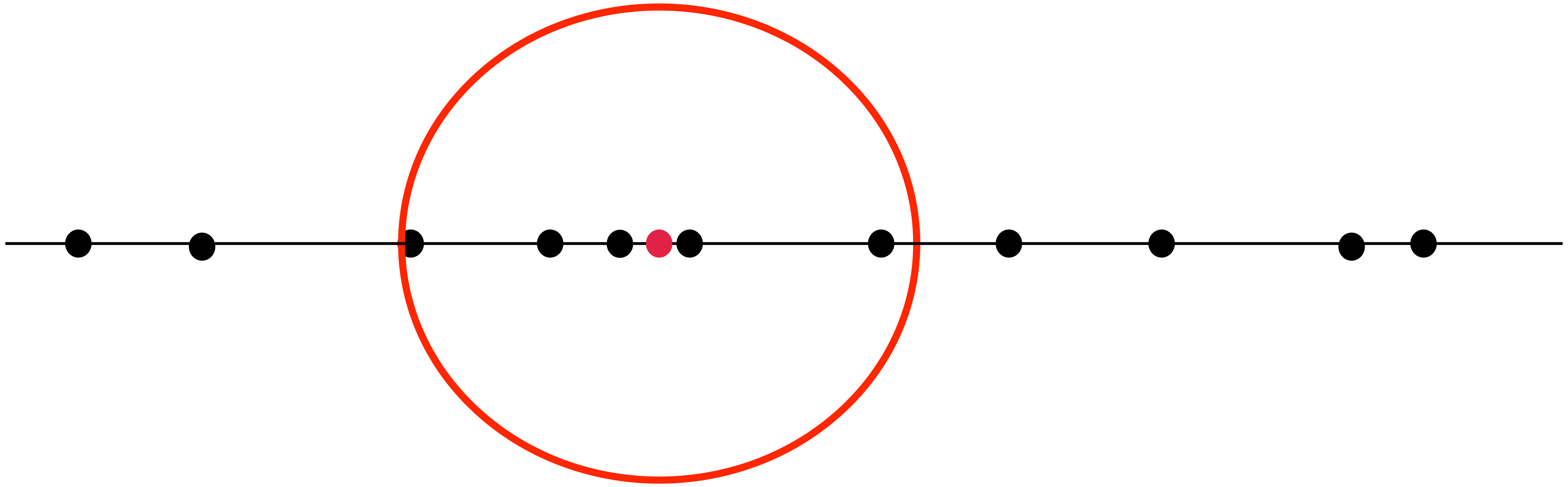
Input: n points on Line L

Radius r > 0

Output: The center c on L of the r-cycle enclosing most points.

Our line sweeping algorithm computes c in O(n) time.

Computing the One-Dimensional r-Cycle Enclosing Most Points



Input: n points on Line L

Radius r > 0

Output: The center c on L of the r-cycle enclosing most points.

Our line sweeping algorithm computes c in O(n) time.

Summary

- Two-dimension Version: Computing the center of the cycle of radius r to enclose most points in the plane

——— $O(n^2 \log n)$ time

- Line-constrained Version: Computing the center of the cycle of radius r to enclose most points in the plane with the constraint where the center must be on a given line

——— $O(n \log n)$ time

- One-dimension Version: Computing the center of the cycle of radius r to enclose most points on a given line

——— $O(n)$ time

Conclusion

- Efficiently designed algorithms that compute the optimal location of the facility to serve/communicate with most objects.
  - We propose an $O(n^2 \log n)$-time algorithm to solve the two-dimension Version: Computing the center of the cycle of radius r to enclose most points in the plane.
  - We propose an $O(n \log n)$-time algorithm to solve the line-constrained Version: Computing the center of the cycle of radius r to enclose most points in the plane with the constraint where the center must be on a given line.
  - We propose an $O(n)$-time algorithm to solve the one-dimension Version: Computing the center of the cycle of radius r to enclose most points on a given line.
- Our techniques can be applied to the high-dimension clustering.

C code for the Planar Version:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#define MAX_INPUT_POINTS 12000
#define PI 3.141592654

// Define Input Point Structure
typedef struct
{
   double X;
         double Y;
   char flag; // Flag to determine if sweep has encountered enter point ("I") before exit point
("O")
} input_point;

// Define Intersection Point Structure
typedef struct
{
   int pointNum;
   input_point *point;
   double X;
   double Y;
   double angle;
   char dir;
   int total_intersections;
} intersection;

// "Merge Sort Program in C"
// By: Aman Goel
// Source: https://hackr.io/blog/merge-sort-in-c
void merge_sort(int i, int j, intersection a[], intersection aux[]) {
   if (j <= i) {
      return;    // the subsection is empty or a single element
   }
   int mid = (i + j) / 2;

   // left sub-array is a[i .. mid]
   // right sub-array is a[mid + 1 .. j]

   merge_sort(i, mid, a, aux);    // sort the left sub-array recursively
   merge_sort(mid + 1, j, a, aux);    // sort the right sub-array recursively
```

```
   int pointer_left = i;      // pointer_left points to the beginning of the left sub-array
   int pointer_right = mid + 1;       // pointer_right points to the beginning of the right sub-array
   int k;     // k is the loop counter

   // we loop from i to j to fill each element of the final merged array
   for (k = i; k <= j; k++) {
      if (pointer_left == mid + 1) {      // left pointer has reached the limit
         aux[k] = a[pointer_right];
         pointer_right++;
      } else if (pointer_right == j + 1) {       // right pointer has reached the limit
         aux[k] = a[pointer_left];
         pointer_left++;
      } else if (a[pointer_left].X < a[pointer_right].X) {       // pointer left points to smaller
element
         aux[k] = a[pointer_left];
         pointer_left++;
      } else {       // pointer right points to smaller element
         aux[k] = a[pointer_right];
         pointer_right++;
      }
   }

   for (k = i; k <= j; k++) {      // copy the elements from aux[] to a[]
      a[k] = aux[k];
   }
}

/***************************************************************************
*
* C source code example
* Author: Tim Voght
* Date: 3/26/2005
* Web Address: http://paulbourke.net/geometry/circlesphere/
* Availability: http://paulbourke.net/geometry/circlesphere/tvoght.c
***************************************************************************
/
int circle_circle_intersection(double x0, double y0, double r0,
                               double x1, double y1, double r1,
                               double *xi, double *yi,
                               double *xi_prime, double *yi_prime)
{
  double a, dx, dy, d, h, rx, ry;
  double x2, y2;
```

```
/* dx and dy are the vertical and horizontal distances between
 * the circle centers.
 */
dx = x1 - x0;
dy = y1 - y0;

/* Determine the straight-line distance between the centers. */
//d = sqrt((dy*dy) + (dx*dx));
d = hypot(dx,dy); // Suggested by Keith Briggs

/* Check for solvability. */
if (d > (r0 + r1))
{
  /* no solution. circles do not intersect. */
  return 0;
}
if (d < fabs(r0 - r1))
{
  /* no solution. one circle is contained in the other */
  return 0;
}

/* 'point 2' is the point where the line through the circle
 * intersection points crosses the line between the circle
 * centers.
 */

/* Determine the distance from point 0 to point 2. */
a = ((r0*r0) - (r1*r1) + (d*d)) / (2.0 * d) ;

/* Determine the coordinates of point 2. */
x2 = x0 + (dx * a/d);
y2 = y0 + (dy * a/d);

/* Determine the distance from point 2 to either of the
 * intersection points.
 */
h = sqrt((r0*r0) - (a*a));

/* Now determine the offsets of the intersection points from
 * point 2.
 */
rx = -dy * (h/d);
```

```
  ry = dx * (h/d);

  /* Determine the absolute intersection points. */
  *xi = x2 + rx;
  *xi_prime = x2 - rx;
  *yi = y2 + ry;
  *yi_prime = y2 - ry;

  return 1;
}

int main() {
   // TEST: PRINT INPUT SIZE
   printf("Input Size: %d\n", MAX_INPUT_POINTS);

   // Initialize Timer Variables
   clock_t start_t, end_t;
   double total_t;

   // Radius r
   double r = 2;

   // Use current time as seed for random generator
          srand(time(0));

   // Generate random input points
   input_point Point[MAX_INPUT_POINTS];
   for (int i = 0; i < MAX_INPUT_POINTS; i++)
   {
      Point[i].X = (rand() % 10);
      Point[i].Y = (rand() % 10);

                    // TEST: PRINT INPUT POINTS
                    // printf("Point %d: (%f, %f)\n", i, Point[i].X, Point[i].Y);
   }

   // Start timer
   start_t = clock();

   // Initialize counter values to track current and max overlap
   int total = 1;
   int max = 0;

   // Initialize max pointer to track max intersection
```

```
  intersection *max_point;

  // Initialize array for max points of each loop
  intersection MaxPoints[MAX_INPUT_POINTS];
  int maxIndex = 0;

  for (int i = 0; i < (MAX_INPUT_POINTS-1); i++)
  {
    // Generate intersection points
    intersection Intersection[MAX_INPUT_POINTS * 2];
    int index = 0;

    // Reset total value
    total = 1;

    // Compare input point to each other input point
    for (int j = i+1; j < MAX_INPUT_POINTS; j++)
    {
      // Point 1: (x-h)^2 + (y-k)^2 = r^2     (Input point 1)
      double h = Point[i].X;
      double k = Point[i].Y;
      // Point 2: (x-o)^2 + (y-q)^2 = r^2     (Input point 2)
      double o = Point[j].X;
      double q = Point[j].Y;

      // Calculate distance between two input points
      double d = fabs(sqrt(pow((h-o), 2) + pow((k-q), 2)));

      // If distance between points is greater than 2r, print "no intersection" and continue loop
rest of points
      if(d > (2*r)) {
        // printf("No Intersection\n");
        continue;
      // If two points are the exact same, print "infinite solutions" and continue loop rest of
points
      } else if ((h == o) && (k == q)) {
        // printf("Infinite Solutions\n");
        continue;
      }

      // Initialize intersection coordinates
      double x1, x2 = 0;
      double y1, y2 = 0;
```

```
// Calculate Intersection Coordinates
circle_circle_intersection(h, k, r, o, q, r, &x1, &y1, &x2, &y2);

// Initialize intersection angles
double ang1, ang2 = 0;
double temp1, temp2 = 0;

// Calculate intersection angles
// Calculate angle 1
if (x1 == h)
{
  if (y1 < k)
    ang1 = 90;
  else
    ang1 = 270;
} else {
  temp1 = (atan2((y1 - k), (x1 - h)) * (180/PI));
  if(temp1 > 0) {
    ang1 = 360 - temp1;
  } else {
    ang1 = abs(temp1);
  }
}
// Calculate angle 2
if (x2 == h)
{
  if (y2 < k)
    ang2 = 90;
  else
    ang2 = 270;
} else {
  temp2 = (atan2((y2 - k), (x2 - h)) * (180/PI));
  if(temp2 > 0) {
    ang2 = 360 - temp2;
  } else {
    ang2 = abs(temp2);
  }
}

// Calculate distance between point 1 right bound (angle 0) and point 2 center (o, q)
double zero = fabs(sqrt(pow(((h+r)-o), 2) + pow((k-q), 2)));

// Store intersection values in array
// Define intersection 1
```

```
        Intersection[index].point = &Point[j];
        Intersection[index].pointNum = j;
        Intersection[index].X = x1;
        Intersection[index].Y = y1;
        Intersection[index].angle = ang1;
        // Check if angle 0 is within circle of compared point to determine "In" or "Out" Type
        if (zero < r || ((zero == r) && (q > k))) {
          if (ang1 < ang2) {
            Intersection[index].dir = 'O'; // Using 'I' for "In"/"Entering" point
          } else if (ang1 > ang2) {
            Intersection[index].dir = 'I'; // Using 'O' for "Out"/"Exiting" point
          } else {
            Intersection[index].dir = 'I'; // If ang1 = ang2, make intersection 1 "I" and
intersection 2 "O"
          }
        } else {
          if (ang1 < ang2) {
            Intersection[index].dir = 'I'; // Using 'I' for "In"/"Entering" point
          } else if (ang1 > ang2) {
            Intersection[index].dir = 'O'; // Using 'O' for "Out"/"Exiting" point
          } else {
            Intersection[index].dir = 'I'; // If ang1 = ang2, make intersection 1 "I" and
intersection 2 "O"
          }
        }
        // Set Input Point flag to 'F' by default (Indicates the point has not been scanned yet)
        Intersection[index].point->flag = 'F';
        index++;

        // Define intersection 2
        Intersection[index].point = &Point[j];
        Intersection[index].pointNum = j;
        Intersection[index].X = x2;
        Intersection[index].Y = y2;
        Intersection[index].angle = ang2;
        // Check if angle 0 is within circle of compared point to determine "In" or "Out" Type
        if (zero < r || ((zero == r) && (q > k))) {
          if (ang2 < ang1) {
            Intersection[index].dir = 'O'; // Using 'I' for "In"/"Entering" point
          } else if (ang2 > ang1) {
            Intersection[index].dir = 'I'; // Using 'O' for "Out"/"Exiting" point
          } else {
            Intersection[index].dir = 'O'; // If ang1 = ang2, make intersection 1 "I" and
intersection 2 "O"
```

```
            }
        } else {
            if (ang2 < ang1) {
                Intersection[index].dir = 'I'; // Using 'I' for "In"/"Entering" point
            } else if (ang2 > ang1) {
                Intersection[index].dir = 'O'; // Using 'O' for "Out"/"Exiting" point
            } else {
                Intersection[index].dir = 'O'; // If ang1 = ang2, make intersection 1 "I" and
intersection 2 "O"
            }
        }
        // Set Input Point flag to 'F' by default (Indicates the point has not been scanned yet)
        Intersection[index].point->flag = 'F';
        index++;

        // Increment total variable to account for points within a circle that overlaps angle '0'
        if (zero < r) {
            total++;
        }

    }

    // Check if there are no intersections; if so, continue loop and check next circle
    if (index == 0) {
        // printf("No Intersections\n\n");
        continue;
    }

    // Sort input points based off 'X' values (Merge Sort)
    intersection aux[MAX_INPUT_POINTS];
    merge_sort(0, index-1, Intersection, aux);

    // Sweep intersection list
    for (int k=0; k<index; k++)
    {
        if (Intersection[k].dir == 'I')
        {
            total++;
            if (total >= max)
            {
                max = total;
                max_point = &Intersection[k];
            }
            Intersection[k].point->flag = 'T'; // Set flag to true ('T')
```

```
            Intersection[k].total_intersections = total;

        } else if (Intersection[k].dir == 'O') {
            if (Intersection[k].point->flag == 'F') {
                if (total >= max) {
                    max = total;
                    max_point = &Intersection[k];
                }
            } else {
                Intersection[k].point->flag = 'F'; // Set flag to false ('F')
            }
            Intersection[k].total_intersections = total;
            total--;
        }
    }

    /*
    // Print coordinates of optimal point
    printf("Optimal point at:\n");
    printf("Point %d.%c: (%f, %f) Intersections: %d\n\n", max_point->pointNum,
max_point->dir, max_point->X, max_point->Y, max_point->total_intersections);
    */

    // Store max intersection for this loop
    MaxPoints[maxIndex].pointNum = max_point->pointNum;
    MaxPoints[maxIndex].X = max_point->X;
    MaxPoints[maxIndex].Y = max_point->Y;
    MaxPoints[maxIndex].dir = max_point->dir;
    MaxPoints[maxIndex].total_intersections = max_point->total_intersections;
    maxIndex++;
  }

  // Scan max point of each loop to determine final overall max point
  for (int count = 0; count < maxIndex; count++) {
    if (max_point->total_intersections < MaxPoints[count].total_intersections) {
      max_point = &MaxPoints[count];
    }
  }

  // Print coordinates of optimal point
  printf("\nFinal Optimal point at:\n");
  printf("Point %d.%c: (%f, %f) Intersections: %d\n\n", max_point->pointNum, max_point->dir,
max_point->X, max_point->Y, max_point->total_intersections);
```

```
    // End timer
    end_t = clock();

    printf("\nInput Size: %d\n", MAX_INPUT_POINTS);

    // Calculate runtime
    total_t = ((double)(end_t - start_t) / CLOCKS_PER_SEC);
    printf("Runtime: %0.10f seconds\n", total_t);

}
```

C code for Line-Constrained Case:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#define MAX_INPUT_POINTS 15000
/*
sources: https://iq.opengenus.org/qsort-in-
c/#:~:text=qsort%20in%20C%20is%20an,h%20header%20file%20in%20C.
*/
typedef struct
{
   int pointNum;
   int pointX;
   int pointY;
   double intersectX;
   double intersectY;
   char dir;
   int total_intersections;
}Intersection;

// "Merge Sort Program in C"
// By: Aman Goel
// Source: https://hackr.io/blog/merge-sort-in-c
void merge_sort(int i, int j, Intersection a[], Intersection aux[]) {
   if (j <= i) {
      return;     // the subsection is empty or a single element
   }
   int mid = (i + j) / 2;

   // left sub-array is a[i .. mid]
   // right sub-array is a[mid + 1 .. j]

   merge_sort(i, mid, a, aux);     // sort the left sub-array recursively
   merge_sort(mid + 1, j, a, aux);     // sort the right sub-array recursively

   int pointer_left = i;       // pointer_left points to the beginning of the left sub-array
   int pointer_right = mid + 1;        // pointer_right points to the beginning of the right sub-array
   int k;      // k is the loop counter

   // we loop from i to j to fill each element of the final merged array
   for (k = i; k <= j; k++) {
      if (pointer_left == mid + 1) {      // left pointer has reached the limit
```

```
         aux[k] = a[pointer_right];
         pointer_right++;
      } else if (pointer_right == j + 1) {       // right pointer has reached the limit
         aux[k] = a[pointer_left];
         pointer_left++;
      } else if (a[pointer_left].intersectX < a[pointer_right].intersectX) {       // pointer left points
to smaller element
         aux[k] = a[pointer_left];
         pointer_left++;
      } else {       // pointer right points to smaller element
         aux[k] = a[pointer_right];
         pointer_right++;
      }
   }

   for (k = i; k <= j; k++) {      // copy the elements from aux[] to a[]
      a[k] = aux[k];
   }
}

int main() { //horizontal line only

   Intersection intersection[MAX_INPUT_POINTS * 2]; //create array of intersection struct
   int index = 0; //index of struct array

   // Initialize Timer Variables
   clock_t start_t, end_t;
   double total_t;

   // Seed random number generator
   srand(time(0));
   // Random input points
   int pointsX[MAX_INPUT_POINTS];
   int pointsY[MAX_INPUT_POINTS];
   for (int i = 0; i < MAX_INPUT_POINTS; i++)
   {
      pointsX[i] = (rand() % 10);
      pointsY[i] = (rand() % 10);
   }

   // line y = mx + b
   double b = 3;
   double m = 0.5;
```

```
    // Radius r
    double r = 2;

    // Start timer
    start_t = clock();

    for (int i = 0; i < MAX_INPUT_POINTS; i++)
    {
      //Center point of circle (x-h)^2 + (y-k)^2 = r^2
      double h = pointsX[i]; //x-value
      double k = pointsY[i]; //y-value

      printf("\npoint %d: (%f, %f)\n",i,h,k);


      // Calculate intersection points x = +/- sqrt(r^2-b^2+2bk-k^2) + h
      double x1 = ((h-m*b+m*k) + sqrt(-(m*m)*(h*h) + 2*m*k*h - 2*m*b*h + (m*m)*(r*r) +
2*b*k + (r*r) - (b*b) - (k*k))) / (1+(m*m));
      double x2 = ((h-m*b+m*k) - sqrt(-(m*m)*(h*h) + 2*m*k*h - 2*m*b*h + (m*m)*(r*r) +
2*b*k + (r*r) - (b*b) - (k*k))) / (1+(m*m));

      // Calculate intersection points y = mx+b
      double y1 = m * x1 + b;
      double y2 = m * x2 + b;

      //if x != real number it is not intersecting
      if (isnan(x1) || isnan(x2))
      {
        printf("no intersection\n");
        continue;
      }

      //if x1=x2 it is tangent
      //just make two entries of the same point
      //only difference is L & R
      if (x1 == x2)
        printf("Tangent intersection\n");

      printf("Left intersection point  (%f,%f)\n", x2,y2);
      printf("Right intersection point (%f,%f)\n", x1,y1);

      //add left value to struct
      intersection[index].pointNum = i;
```

```
    intersection[index].pointX = h;
    intersection[index].pointY = k;
    intersection[index].intersectX = x2;
    intersection[index].intersectY = y2;
    intersection[index].dir = 'L';
    ++index;

    //add Right value to struct
    intersection[index].pointNum = i;
    intersection[index].pointX = h;
    intersection[index].pointY = k;
    intersection[index].intersectX = x1;
    intersection[index].intersectY = y1;
    intersection[index].dir = 'R';
    ++index;
}

// TEST: display point number and intersection x value
/*
printf("Before sorting\n");
printf("point num | intersect x-value\n");
for (int i=0; i< index; i++)
    printf("%d | %f\n", intersection[i].pointNum, intersection[i].intersectX);
*/

// Sort input points based off 'X' values (Merge Sort)
Intersection aux[MAX_INPUT_POINTS];
merge_sort(0, index-1, intersection, aux);

// TEST: prove it sorted based off the x-intersections
/*
printf("After sorting\n");
for (int i=0; i< index; i++)
    printf("%d | %f\n", intersection[i].pointNum, intersection[i].intersectX);

//end result
for (int i=0; i< index; i++)
    printf("| %d.%c ", intersection[i].pointNum, intersection[i].dir);
printf("|\n");
*/

//idk
int total = 0;
int max = 0;
```

```
    //sweep left to right to determine optimal point/s
    for (int i=0; i<index; i++)
    {
        if (intersection[i].dir == 'L')
        {
            total++;
            if (total >= max)
            {
                max = total;
                intersection[i].total_intersections = max;
            }
        }
        if (intersection[i].dir == 'R')
        {
            total--;
        }
    }

    printf("Optimal point/s at: ");
    for (int i=0; i<index; i++)
    {
        if (intersection[i].total_intersections == max)
            //printf("%d ", intersection[i].pointNum);
            printf("(%f, %f) ", intersection[i].intersectX, intersection[i].intersectY);
    }

    // End timer
    end_t = clock();

    printf("\nInput Size: %d\n", MAX_INPUT_POINTS);

    // Calculate runtime
    total_t = ((double)(end_t - start_t) / CLOCKS_PER_SEC);
    printf("Runtime: %0.10f seconds\n", total_t);

}
```

C code for One-Dimensional Case:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#define MAX_INPUT_POINTS 15000

// Define Intersection Point Structure
typedef struct
{
   int pointNum;
   double X;
   double Y;
   char dir;
   int total_intersections;
} intersection;

// Define Input Point Structure
typedef struct
{
   int pointNum;
          double X;
          double Y;
   intersection *Left;
   intersection *Right;
} input_point;

// "Merge Sort Program in C"
// By: Aman Goel
// Source: https://hackr.io/blog/merge-sort-in-c
void merge_sort(int i, int j, input_point a[], input_point aux[]) {
   if (j <= i) {
      return;     // the subsection is empty or a single element
   }
   int mid = (i + j) / 2;

   // left sub-array is a[i .. mid]
   // right sub-array is a[mid + 1 .. j]

   merge_sort(i, mid, a, aux);     // sort the left sub-array recursively
   merge_sort(mid + 1, j, a, aux);     // sort the right sub-array recursively

   int pointer_left = i;       // pointer_left points to the beginning of the left sub-array
```

```
   int pointer_right = mid + 1;        // pointer_right points to the beginning of the right sub-array
   int k;      // k is the loop counter

   // we loop from i to j to fill each element of the final merged array
   for (k = i; k <= j; k++) {
      if (pointer_left == mid + 1) {      // left pointer has reached the limit
         aux[k] = a[pointer_right];
         pointer_right++;
      } else if (pointer_right == j + 1) {        // right pointer has reached the limit
         aux[k] = a[pointer_left];
         pointer_left++;
      } else if (a[pointer_left].X < a[pointer_right].X) {        // pointer left points to smaller
element
         aux[k] = a[pointer_left];
         pointer_left++;
      } else {        // pointer right points to smaller element
         aux[k] = a[pointer_right];
         pointer_right++;
      }
   }

   for (k = i; k <= j; k++) {      // copy the elements from aux[] to a[]
      a[k] = aux[k];
   }
}

// Driver code
int main()
{
   // Use current time as seed for random generator
         // srand(time(0));

   // Initialize Timer
   clock_t start_t, end_t;
   double total_t;

   /*
         // Define User Input Values for Radius and Line
         double r = (rand() % 10) + 1;
         double m = (rand() % 10);
         double b = (rand() % 10);

   // TEST: PRINT LINE PARAMETERS
   printf("Line: y = %fx + %f \n", m, b);
```

```
    printf("Radius: %f \n", r);
    */


    // TEMPORARY: FOR TESTING
    double r = 2;
    double m = 1;
    double b = 1;


        // Generate random input points
    input_point Point[MAX_INPUT_POINTS];
    for (int i = 0; i < MAX_INPUT_POINTS; i++)
    {
      Point[i].X = (rand() % 10);
      Point[i].Y = (m * Point[i].X) + b;
      Point[i].pointNum = i;

            // TEST: PRINT INPUT POINTS
            // printf("Point %d: (%f, %f) \n", i, Point[i].X, Point[i].Y);
    }

        // Generate intersection points
        intersection Intersection[MAX_INPUT_POINTS * 2];
        int intersect = 0;
        for (int i = 0; i < MAX_INPUT_POINTS; i++)
    {
            // Define Left Bound Point
            Intersection[intersect].pointNum = i;
            Intersection[intersect].X = Point[i].X - (r * cos(atan(m)));
            Intersection[intersect].Y = (m * Intersection[intersect].X) + b;
            Intersection[intersect].dir = 'L';
      Point[i].Left = &Intersection[intersect];

            // TEST: PRINT LEFT-BOUND INTERSECTION
            // printf("Point %d Left-Bound: (%f, %f)\n", i, Intersection[intersect].X,
  Intersection[intersect].Y);
            intersect++;

            // Define Right Bound Point
            Intersection[intersect].pointNum = i;
            Intersection[intersect].X = Point[i].X + (r * cos(atan(m)));
            Intersection[intersect].Y = (m * Intersection[intersect].X) + b;
            Intersection[intersect].dir = 'R';
```

```
    Point[i].Right = &Intersection[intersect];

                    // TEST: PRINT RIGHT-BOUND INTERSECTION
                    // printf("Point %d Right-Bound: (%f, %f)\n", i, Intersection[intersect].X,
Intersection[intersect].Y);
                    intersect++;
          }

   // Sort input points based off 'X' values (Merge Sort)
   input_point aux[MAX_INPUT_POINTS];
   merge_sort(0, MAX_INPUT_POINTS-1, Point, aux);

   // TEST: SORTED INPUT POINTS
   /*
   printf("\n\nInput Points After Sorting:\n");
   for (int i=0; i < MAX_INPUT_POINTS; i++) {
      printf("Point %d: (%f, %f)\n", Point[i].pointNum, Point[i].X, Point[i].Y);
   }
   */

  // Start timer
  start_t = clock();

  int total = 0;
  intersection *maxPoint = Point[0].Right;
  maxPoint->total_intersections = 0;
  for(int i=0, j=1; (j<MAX_INPUT_POINTS);) {
     // TEST: VIEW COMPARED POINTS AND VALUES
     printf("%d vs %d | Compare: Point %d.R (%f) vs Point %d.L (%f) | ", i, j, Point[i].pointNum,
Point[i].Right->X, Point[j].pointNum, Point[j].Left->X);
     if (Point[i].Right->X >= Point[j].Left->X) {
        j++;
      } else {
        i++;
      }
      total = j - i;
      Point[i].Right->total_intersections = total;
      // TEST: CHECK OVERLAP VALUES
      printf("Overlap: %d\n", total);
      if (maxPoint->total_intersections <= Point[i].Right->total_intersections) {
         maxPoint = Point[i].Right;
         // TEST: UPDATE MAX POINT
         // printf("       Max Point: Point %d | Overlap %d\n", maxPoint->pointNum,
maxPoint->total_intersections);
```

```
        }
    }

    printf("\nOptimal Point: Point %d.R (%f, %f) | Overlap %d\n", maxPoint->pointNum,
maxPoint->X, maxPoint->Y, maxPoint->total_intersections);

    // End timer
    end_t = clock();

    printf("\nInput Size: %d\n", MAX_INPUT_POINTS);

    // Calculate runtime
    total_t = ((double)(end_t - start_t) / CLOCKS_PER_SEC);
    printf("Runtime: %0.15f seconds\n", total_t);
}
```