Algorithm

In this section, I will explain my algorithm for each key feature.

Circular arc of edge

In the original sweep algorithm, we monitor the events, “Insert”, “Remove”, and “Swap” to maintain the order of line segments. The first challenge to apply this algorithm to circle-circle intersections is that the sweepline intersects with a circle at two points, which is the case we ignored in the line-line intersection algorithm. For this reason, I split each circle into two semicircles as shown below. Each semicircle has the center and the radius of the circle as the edge information in addition to the other already implanted ones. Also, the *leftside* flag is stored that indicates whether the semicircle is on the left side of the circle or not. As a result, we can define the “Insert” and “Remove” events similar to the line segments.

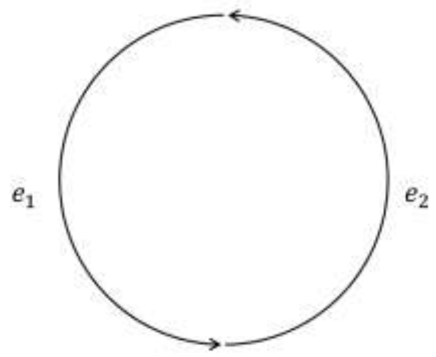

Fig 1. Each circle is split into four circular arcs to compute the “Insert” and “Remove” events correctly.

When we use semicircles instead of the line segments, we have to be careful in some predicates, such as Edge::leftOf(),Edge::clockwise(), and Edge::outer(). I will discuss more details in the following.

X coordinate ordering of edges on the sweepline

When an “Insert” event occurs, the sweepline algorithm adds the new edge into the binary search tree so that we can keep tracking the adjacent edges. To deal with the circles, we have to modify the algorithm for judging which edge is left or right. For the case of Fig. 2 (a), when the new edge e is added, the edge e is right of the edge f. On the other hand, for the case of Fig. 2 (b), the edge e is left of the edge f.

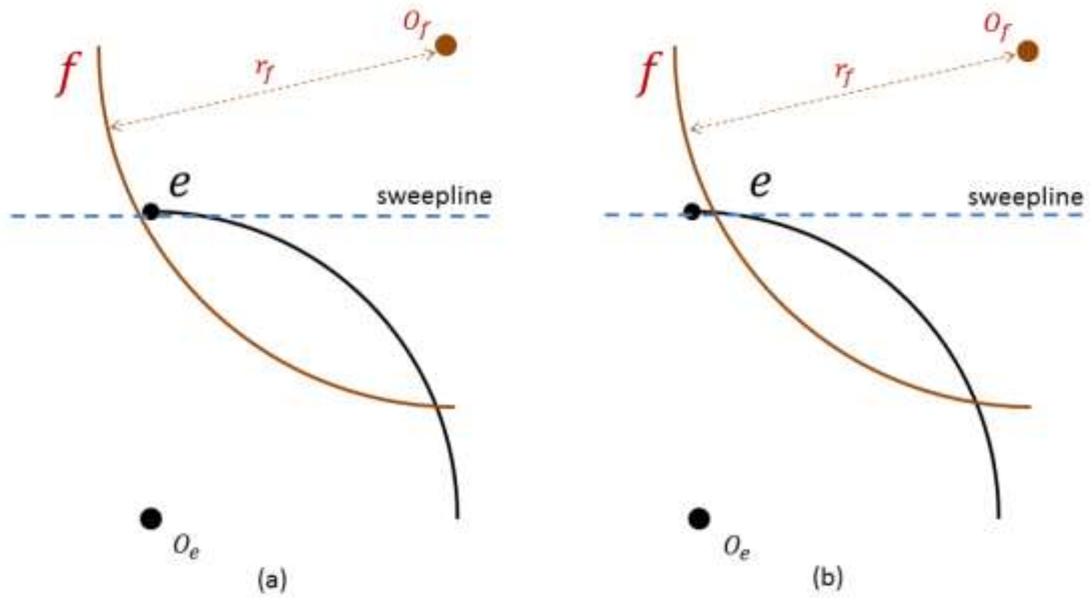

Fig 2. How can we judge that the edge e is right of the edge f in the case (a), while the edge e is left of the edge f in the case (b)?

Let e be the newly added circular arc, f be one of the existing circular arc, O_f be the center of its circle, r_f be its radius, and $head(e)$ be the vertex of e which has the higher Y coordinate than the other side vertex of e. When an edge e is added, and is compared with the existing edge f, the edge e is left of the edge f if and only if one of the following conditions are satisfied:

1) The edge f is on the left side of its circle, the X coordinate of $head(e)$ is less than the X coordinate of O_f, and the distance between $head(e)$ and O_f is greater than r_f,
2) The edge f is on the right side of its circle, and the X coordinate of $head(e)$ is less than the X coordinate of O_f or the distance between $head(e)$ and O_f is less than r_f.

Circle-circle intersection

Even though there are four circular arcs for each circle, we do not want to compute the intersections of circles for four times. Instead, I compute the intersections of each pair of circles at most once, and store them in hash table so that the computed intersections can be retrieved later in the constant time. For the computation of the circle-circle intersections, first we have to consider three cases:

1) The distance between two circles is greater than the sum of their radii.
   In this case, these two circles are too far away from each other, and there is no intersection between them.
2) The distance between two circles is less than the difference of their radii.
   In this case, one circle is completely inside of the other, and there is no intersection between them.
3) Otherwise, there are one or two intersections.

For the third case, we can use the following equation to compute the location of the intersections:

$$a = \frac{r_1^2 - r_2^2 + d^2}{2d}$$

where r_1 and r_2 are the radii of the two circles, respectively and d is the distance between two centers of the circles. Once a is computed, h can be easily computed by Pythagorean theorem. Thus, given the coordinate of the centers of two circles and their radii, we can compute the two intersections between them.

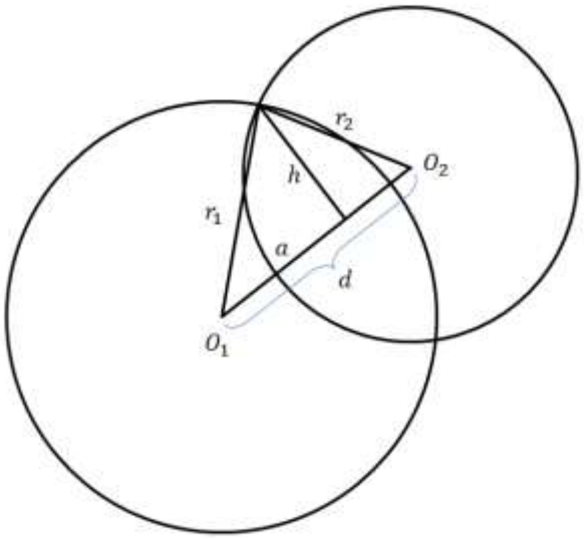

Fig 3. The two intersections of two circles are computed by first calculating the length of a, then calculating the length of h.

Swap of edges in case of two intersections

If there is only one intersection between two edges, we just need to swap two edges e and f in the binary search tree to update the order of the edges. However, if there are two intersections, we have to be a little more careful to take care of this. In the case of Fig. 4, when the edge e is added, it is left of the edge f. Then, when the sweepline reaches S_1, the first "Swap" event occurs, and the order of e and f is swapped. By the time when the sweepline reaches S_2, the order of e and f is reversed. Therefore, for the second "Swap" event, we have to put the edges in the reversed order.

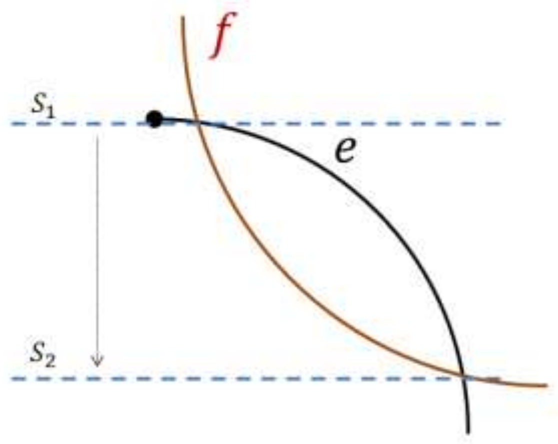

Fig 4. In the case that there are two intersections between two edges, the first "Swap" event swaps the order of these two edges, and by the time when the second "Swap" event occurs, the order of the edges *e* and *f* is reversed.

Computation of number of components

For the computation of the number of components, I keep tracking the set of circles for each component. Every time when there are intersections between two circles, I combine the two sets into one. In the end, I remove the components that are empty so that I have the set of components each of which contains one or more circles as its members.

Implementation

In this section, I will explain how I modified each function of ACP library to implement the aforementioned algorithm.

Point.h

I changed the constructor *InputPoint(const PV2 &ip)* to a public method so that I can create a *Point* object from *PV2* object.

Circle.h

I added *component* as a member of *Circle* class. This represents in which component this circle belongs to.

Component class

I added *Component* class that represents a connected component.

```
class Component {
 public:
  set<Circle*> members;
};

typedef vector<Component*> Components;
```

CirclePair class

This class represents a pair of two circles, and is used to check whether these two circles are already computed for their intersections. As I stated above, I compute the intersections of two circles at most one time even though each circle consists of two semicircles. Every time when I compute the intersections of two circles, I store the pair in the list. When I check the intersections of two circles for the second time, I first look up this list in order to find whether the two circles are already computed.

```
class CirclePair {
 public:
  CirclePair (Circle *c1, Circle *c2) : c1(c1), c2(c2) {}
  bool operator< (const CirclePair &p) const {
    if (c1 < p.c1) {
      return true;
       }
       if (c1 == p.c1 && c2 < p.c2) {
      return true;
    }

       return false;
  }

  Circle *c1, *c2;
};

typedef set<CirclePair> CirclePairSet;
```

Edge class

I added a pointer to a circle object and a Boolean flag to indicate whether the edge is on the left side of the circle or not.

```
class Edge {
 public:
  ...(snip)...
  Circle* circle;
  bool leftSide;
};
```

Edge::clockwise()

Since I split a circle into two semicircles, and both of them have the same vertex, the exising Edge::clockwise() function fails when it compares those two edges. Therefore, I modified it to check the *leftSide* flag if those two edges are from the same circle.

```
bool Edge::clockwise (Edge *e)
{
  if (circle == e->circle)
    return leftSide;
  ...(snip)...
}
```

Edge::leftOf()

As I described above, I modified *Edge::LeftOf()* function to deal with the circles. The main part of this function is as shown below.

```
bool Edge::leftOf (Edge *e)
{
  ...(snip)...

  if (e->leftSide) {
    if (tail->p->getP().getX() > e->circle->getO().getX()) return false;
       return !e->circle->contains(tail->p);
  } else {
    if (tail->p->getP().getX() < e->circle->getO().getX()) return true;
       return e->circle->contains(tail->p);
  }
}
```

Edge::intersects()

As I described above, I modified *Edge::intersects()* function to compute the intersections of two circles. The main part of this function is as shown below.

```
bool Edge::intersects (Edge *e, Points &points)
{
  ...(snip)...

  Parameter a = (d2 + circle->getRR() - e->circle->getRR()) / 2 / d;
  PV2 midPt = circle->getO() + dir * a / d;

  Normal normal(new InputPoint(circle->getO()), new InputPoint(e->circle->getO()));
  Parameter h = (circle->getRR() - a * a).sqrt();

  Point* intersection1 = new InputPoint(midPt + normal.getP() / d * h);
  Point* intersection2 = new InputPoint(midPt - normal.getP() / d * h);

  if (::YOrder(intersection1, intersection2)) {
    points.push_back(intersection2);
    points.push_back(intersection1);
  } else {
    points.push_back(intersection1);
    points.push_back(intersection2);
  }

  return true;
}
```

Edge::withinArc()

After the intersections are computed, this function checks whether those intersections lie on the edge. Since we already know that the intersections are on the circle, the only thing we have to check is whether the X coordinate and the Y coordinate of the intersection is within the bounding box of the circular arc. The actual code is as shown below.

```
bool Edge::withinArc (Point* point) const
{
  if (leftSide && point->getP().getX() > circle->getO().getX()) return false;
  if (!leftSide && point->getP().getX() < circle->getO().getX()) return false;

  return true;
}
```

Edge::outer()

The original function does not support a case that a resulting face consists of only two edges as shown below.

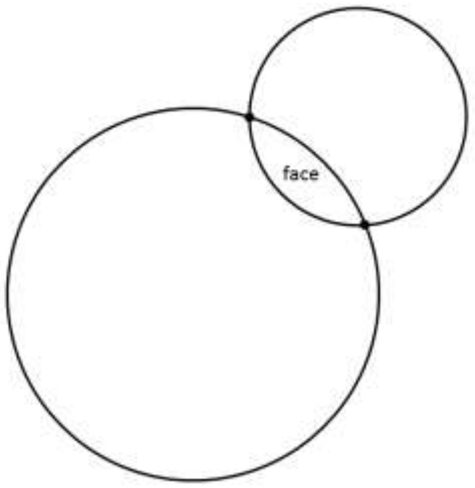

Fig 5. The face consists of only two edges. In this case, Edge::outer() function fails.

To deal with this situation, I modified the function such that if the face consists of two edges, the boundary is considered as an outer boundary. The modified function is as follows:

```
bool Edge::outer ()
{
  Edge *f = twin->next;

  if (f != twin && f->head()->p == tail->p) {
       return (leftSide && !increasingY()) || (!leftSide && increasingY());
  }

  ...(snip)...
}
```

Arrangement::swap()

Because of the perturbation of the coordinate of vertices, it may happen that a “Remove” event occurs before a “Swap” event. In the event handler of “Remove” function, the corresponding node is removed from the binary search tree. Therefore, I modified this function to check if the corresponding node exists before accessing it. The modified code is as shown below.

```
void Arrangement::swap (Edge *e, Edge *f, Point *p, Sweep &sweep,
                     Events &heap, map<CirclePair, Points> &eset)
{
  split(e, f, p);

  Edge *pred = 0;
  Edge *succ = 0;
  if (e->node) {
    pred = e->pred();
  }
  if (f->node) {
       succ = f->succ();
  }

  //Edge *pred = e->pred(), *succ = f->succ();
  if (e->node && f->node) {
    sweep.swap(e, f);
  }
  if (pred)
    e->head()->left = pred->twin;
  if (pred) {
    check(pred, f, heap, eset);
  }
  if (succ) {
    check(e, succ, heap, eset);
  }
}
```

Arrangement::check()

In the case that there are two intersections between the two edges, the two edges are swapped by the time when the second “Swap” event occurs. To deal with this, I used a flag “swapped” to check whether the intersection is the first one or the second one. Also, in this function, every time when the intersections of two circles are computed, the set of components are merged. The modified function is as follows:

```
void Arrangement::check (Edge *e, Edge *f, Events &heap, map<CirclePair, Points>
&intersectionsMap) const
{
  if (e && f && !(rbflag && e->aflag == f->aflag)) {
    if (e->circle == f->circle) return;

    Points intersections;

    CirclePair ef(e->circle < f->circle ? e->circle : f->circle, e->circle < f->circle ?
f->circle : e->circle);
    if (intersectionsMap.find(ef) != intersectionsMap.end()) {
      intersections = intersectionsMap[ef];
      if (intersections.size() == 0) return;
    } else {
      if (e->intersects(f, intersections)) {
        // update components
        if (e->circle->component != f->circle->component) {
          e->circle->component->members.insert(f->circle->component->members.begin(), f-
>circle->component->members.end());
          f->circle->component->members.clear();
          f->circle->component = e->circle->component;
        }
      }
      intersectionsMap[ef] = intersections;
    }

    Points remainedIntersections;

    bool swapped = false;
    for (int i = 0; i < intersections.size(); ++i) {
      if (e->withinArc(intersections[i]) && f->withinArc(intersections[i])) {
        if (swapped) {
          pushHeap(Event(Swap, intersections[i], f, e), heap);
        } else {
          pushHeap(Event(Swap, intersections[i], e, f), heap);
        }
        swapped = true;
      } else {
        remainedIntersections.push_back(intersections[i]);
      }
    }

    intersectionsMap[ef] = remainedIntersections;
  }
}
```

Arrangement::computeComponents()

In the end, I compute the components. Since the connected circles are in the same component, I just need to remove the components that are empty because of the merge. The remained components contain one or more circles.

```
void Arrangement::computeComponents ()
{
  for (vector<Component*>::iterator it = components.begin(); it != components.end(); ) {
    if ((*it)->members.size() == 0) {
         it = components.erase(it);
       } else {
         ++it;
       }
  }
}
```

Arangement::computePS2()

I added this function to compute the vertices, edges, faces, and components of the circles.

```
void Arrangement::computePS2 ()
{
  intersectEdges();
  formFaces();
  computeComponents();
}
```

How to compile?

Please type the following command to compile the files.

```
$ make
```

Then, please type the following command to run the program.

```
$ ./ps2-nishida

4

0 0 10 15 0 10 30 0 10 7.5 10 10
```