CS 6150: Greedy Algorithm

Pre-submission deadline: Nov 15, 2014

This assignment has 5 questions, for a total of 100 points. Unless otherwise specified, complete and reasoned arguments will be expected for all answers.

Question	Points	Score
Spanning Tree and *s*-*t* Path	20	
Unique Minimum Spanning Tree	20	
Class Scheduling Alternatives	20	
Getting Stabby	20	
Zapping Balloons	20	
Total:	100	

Question 1: Spanning Tree and s-t Path .... **[20]**

This is from Kleinberg/Tardos, Exercise 2.

(a) Suppose we are given an instance of the Minimum Spanning Tree Problem on a graph G, with edge costs that are all positive and distinct. Let T be a minimum spanning tree for this instance. Now suppose we replace each edge cost c_e by its square, c_e^2, thereby creating a new instance of the problem with the same graph but different costs.

True or False? T must still be a minimum spanning tree for this new instance.

Solution: It's still true.

Proof. Assume the edges of T is $\{c_1, c_2, c_3, .., c_{k-1}, c_k, c_{k+1}, ..., c_{n-1}\}$ ranked increasingly according to their edge costs.
After each has been replaced by their squares, the edges of T are still the original ones and their cost have been squared. If the conclusion is wrong, then we could make a hypothesis that the minimum spanning tree is $T' = \{d_1, d_2, d_3, ..., d_{k-1}, d_k, d_{k+1}, ...d_{n-1}\}$. Since T and T' are different, and we could always find a different edge from the beginning of the edge set. Let's suppose that c_k is the first edge different from d_k, since each edge have distinct edge cost, so $c_k \neq d_k$ and $c_i = d_i, i \leq k-1$.
If $c_k < d_k$, then add c_k to T'. It also implies that $d_j > c_k, j > k$ and c_k is not in T'. Since T' is a spanning tree, adding c_k will form a circle in T'. Let's suppose edge c_k connects vertices u and v. It's obvious that there is a path from u to v before, so c_k is now in this circle. We could always find an edge in this circle whose cost is larger than c_k. If we couldn't find such one, then it implies the edges in this circle in T' are among the set $\{d_1, d_2, d_3, ..., d_{k-1}\}$. It means the edges in this set and c_k form a circle. However, edges $c_1, c_2, ..., c_{k-1}, c_k$ are all in a spanning tree and it's impossible to form a circle. Hence there is an edge d_m larger than c_k in this circle.If we delete this edge d_m from T', then $T' \bigcup c_k \backslash d_m$ is still a spanning tree. And the total cost now is smaller which is contradict to the hypothesis that T' is MST.
If $c_k > d_k$, then add edge d_k to T. Similarly, a circle containing d_k has formed and there is an edge c_m larger than d_k. It's obvious that $T \bigcup d_k \backslash c_m$ is still a spanning tree. Now replace edges in T by their square root. And we could get a spanning tree whose total sum is smaller than the MST of the original instance. That's impossible.
Hence we could make a conclusion that the edges of T and T' are always the same and it implies T is still the MST. □

(b) Suppose we are given an instance of the Shortest s-t Path Problem on a directed graph G. We assume that all edge costs that are all positive and distinct. Let P be a minimum-cost s-t path for this instance. Now suppose we replace each edge cost c_e by its square, c_e^2, thereby creating a new instance of the problem with the same graph but different costs. True or False? P must still be a minimum-cost s-t path for this new instance.

Solution: It's false. See the counter-example in Figure1.
At first, the Shortest $A - D$ Path is $A \rightarrow B \rightarrow D$. After each edge cost be replaced by its square. $|AB| = 1, |BC| = 4, |CD| = 9, |BD| = 16$. Hence the Shortest $A - D$ path should be $A \rightarrow B \rightarrow C \rightarrow D$ now.The conclusion is wrong.

Question 2: Unique Minimum Spanning Tree .... **[20]**

This is from Kleinberg/Tardos, Exercise 8.

Suppose you are given a connected graph G, with edge costs that are all distinct. Prove that G has a unique minimum spanning tree.

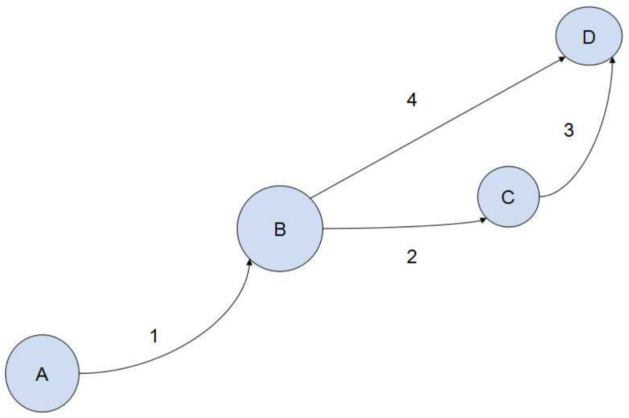

Figure 1

Solution: Define the MST of G is T and rank the edges of T from the smallest to largest. $T = \{t_1, t_2, ..., t_{k-1}, t_k, ..., t_{n-1}\}$. If there is another MST $T' = \{g_1, g_2, ..., g_{k-1}, g_k, ..., g_{n-1}\}$. Since edge cost is distinct, t_i and g_i could represent either the edge cost or the edge itself. Since they are different, we could always find two different edges, let's suppose edge t_k is different from g_k. Since the edge costs are distinct, $t_k \neq g_k$ and $t_i = g_i (i < k)$. Without losing generality, let's suppose $t_k < g_k$. Since $g_j > g_k (j > k)$, we could get $g_j > t_k$. And $g_i = t_i < t_k (i < k)$. So edge t_i is not in T' at first. Now add edge t_k to T' and suppose t_k connects vertices u and v. T' is a spanning tree, so there is a path from u to v at first. After adding edge t_k, a circle containing t_k has formed. Since the edges $g_i = t_i (i < k)$, we could find $g_i (i < k)$ and t_k are edges in T and they couldn't form a circle. So there is an edge g_m larger than t_k in this circle. If we delete edge g_m, then $T' \bigcup t_k \backslash g_m$ is still ap spanning tree and the total cost of edges is smaller than the original T'. It implies that T' is not MST which is contradict to the hypothesis.
In conclusion, if the edge cost is distinct, then we could find a unique MST.

Questions 3-5 are from Erickson (`http://web.engr.illinois.edu/~jeffe/teaching/algorithms/notes/07-greedy.pdf`)

Question 3: Class Scheduling Alternatives . . . . **[20]**
Problem #1, choose any three of the subproblems.

Solution: (a)It's wrong. See the counter-example in Figure 2.
There are three courses in total. Course 1 conflicts with course 2 and course 3 and course 2 doesn't conflict with course 3. If we choose the course ends last that is, course 1, and discard the conflicting ones. Then we could only choose one course. However, we could choose two courses at most. So this algorithm is wrong.
(b)It's wrong. We could still use the counter-example in Figure 2. According to the algorithm, we should choose course1 since it starts first and discard course2 and course3. This is not the optimal solution. So it's wrong.

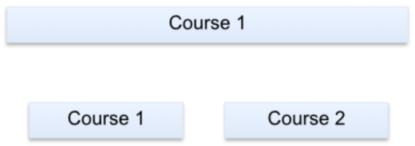

Figure 2

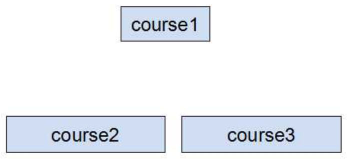

Figure 3

(c)It's wrong. See the counter-example in Figure 3.
According to the algorithm, we will choose course1 since it has the shortest duration and we should discard the conflicting course2 and course3. It's obviously not the optimal solution.

Question 4: Getting Stabby . . . . **[20]**
Problem #4

Solution:

Algorithm 1: getting stabby

Require: Two arrays $X_L[1..n]$ and $X_R[1..n]$, representing the left and right endpoints of the interval in X.
Ensure: The smallest set of points that stabs X.

```
1: $points[1..n]$;
2: sort $X_L[1..n]$ from the most left to most right and permute $X_R[1..n]$ correspondingly;
```

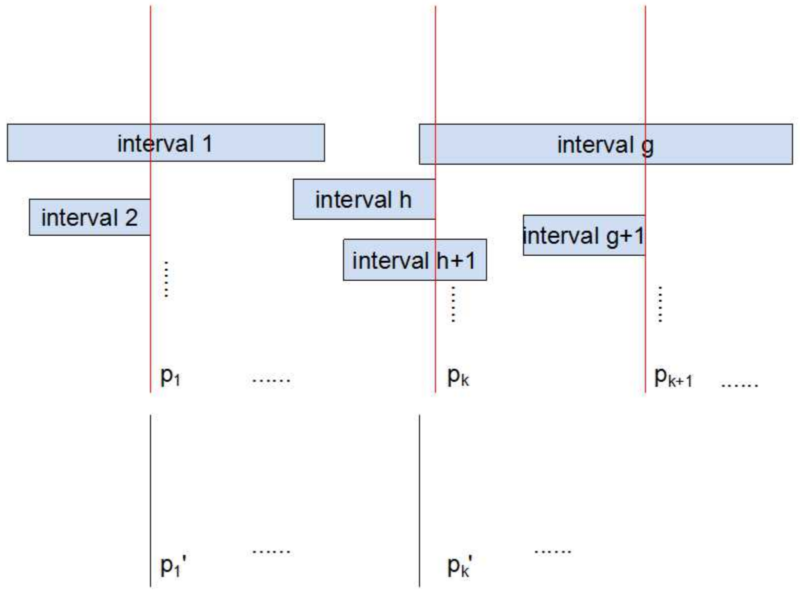

Figure 4

```
3: $left = X_L[1]$,$right = X_R[1]$;%use two variants $left$ and $right$ to represent the *common interval*
   of the first several intervals.
4: $point\_count = 1$;
5: **for** $i = 1$ to $n$ **do**
6:   **if** $X_L[i] > right$ **then**
7:     $points[point\_count++] = right$; %If the next interval's left endpoint is on the right of the
       *common interval*, add the common interval's right endpoint to the points set
8:     $left = X_L[i]$, $right = X_R[i]$;
9:   **else**
10:    $left = max\{X_L[i], left\}$; %continuously narrow the *common interval*
11:    $right = min\{X_R[i], right\}$;
12:  **end if**
13: **end for**
```

Claim: Use variant *point_count* to record the number of points and array *points*$[1..n]$ to record the points set. For line2, use QUICKSORT algorithm to sort $X_L[1..n]$ and permute the corresponding right endpoints of the arrays. I use *Common interval* to represent an interval within which the points belong to several intervals.

Proof. As Figure 4 shows,the algorithm will produce a points set like $p = \{p_1, p_2, ..., p_{k-1}, p_k, p_{k+1}...\}$. At first, let's prove the points set could stab X.
The algorithm continuously update the *common interval*. Once the next interval has intersection with the *common interval*, algorithm use the intersection between *common interval* and the next interval as the new *common interval*. It runs this process until the next interval has no intersection with the *common interval*. So the points including the right endpoint within this *common interval* are also belong to the intervals that form the *common interval*. So the right endpoint could stab

these intervals. For example, as the figure shows, p_k could stab interval h, $h+1$ and $h+2$. For next intervals, new right endpoints will stab them similarly. So the points set could stab X.
Next, prove the points set is the smallest.
Suppose there is a smaller points set $p' = \{p'_1, p'_2, ..., p'_k, ...\}$. Suppose set p has m points and p' has n points($n < m$).
Let's first suppose the n points of p' and the first n points of p are the same. There are still intervals whose left endpoints are on the right of endpoint p_n, otherwise, the algorithm won't produce other $m-n$ points. So point p_n couldn't stab next intervals and the first n points of p and the n points of p' can't be same.
Suppose the $k-th$ pair of points of the two sets are the first pair that are different with each other,say,$p_k \neq p'_k$ and the first $k-1$ pairs of points are the same. We could refer the figure above. Let's suppose the first w intervals have been stabbed and consider them as disappeared. Point p'_k can't be on the right of p_k. That's because p_k is the right endpoint of that *common interval*. It implies that it's the right endpoint of an interval, like interval h as the figure shows. So if p'_k couldn't stab interval h. In addition, $p'_i = p_i (i < k)$ can't stab interval h since interval h's left endpoint is on the right and it hasn't disappeared. So point p'_k could only be on the left of p_k. If it is on the left of *common interval*, then it couldn't stab some intervals whose left endpoints are the left endpoint of *common interval* , however, point p_k could stab these intervals. If it's within the *common interval*, it could the same number intervals as point p_k. So point p'_k could stab no more intervals than p_k, we should change p'_k to p_k in order to stab more intervals. So the first n points of p and the n points of p' should be the same which is impossible as proved above.
In conclusion, there is no smaller points set than the one p produced by this algorithm. □

*Analysis.*This algorithm runs a loop with n iterations. And the operation number in each loop is constant. However, it should sort array $X_L[1..n]$ first. Most sort algorithms have a time complexity $O(nlogn)$. So the time complexity of this algorithm is $O(nlogn)$. We should maintain the points set, so the space complexity is $O(n)$.

Question 5: Zapping Balloons .... **[20]**
Problem #9, part (d) is optional.
(a)

Solution:

Algorithm 2: minimum zap with a known starting position

```
Require: $circle[1..n]$, each element in the array is a structure storing the $(x, y)$ coordinates and
    radius of n circles in the plane.
Ensure: The minimum number of rays to intersect every circle in C.
1: Find an origin that the ray will not intersect with any circle(balloons) and consider it as positive
   Y axis and adjust the $(x, y)$ coordinates of circles;
2: Start from the coordinate axis's origin$(0, 0)$ and produce *right tangents* for every circle, that is,
   $tangent[1..n]$ ;
3: Use QUICKSORT to sort $tangent[1..n]$ according to the *corresponding angle* and permute the
   order of $circle[1..n]$ correspondingly;
4: $rayNum = 0$;
5: $current\_position = positive\ Y\ axis$;
6: for $i = 1$ to $n$ do
7:     Produce a ray m with the starting point$(0, 0)$ and rotate it in the clockwise direction from
       $current\_position$;
8:     Rotate it continuously until the ray m becomes the *right tangent* of a circle, say, $tangent[i]$;
```

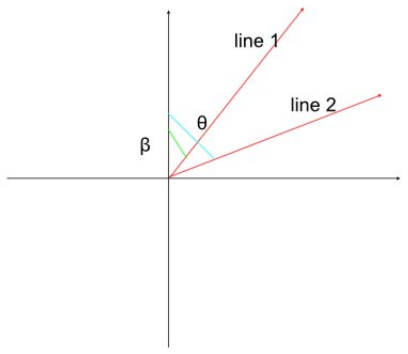

Figure 5: The right tangents of circles

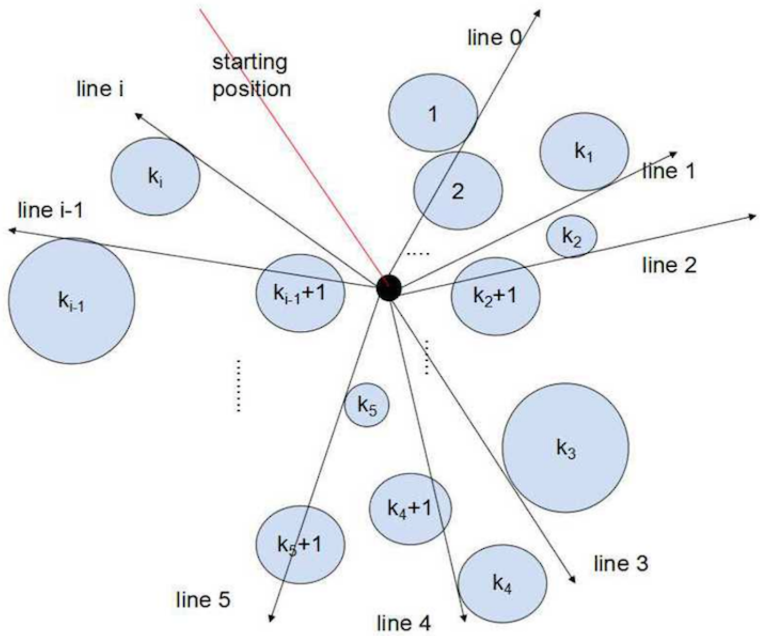

Figure 6

```
 9:     $rayNum++;$
10:     **while** INTERSECTS(RAY M,CIRCLE[I]) $== true$ **do**
11:         $i++;$
12:     **end while**
13:     $current\_position = ray\ m;$
14: **end for**
```

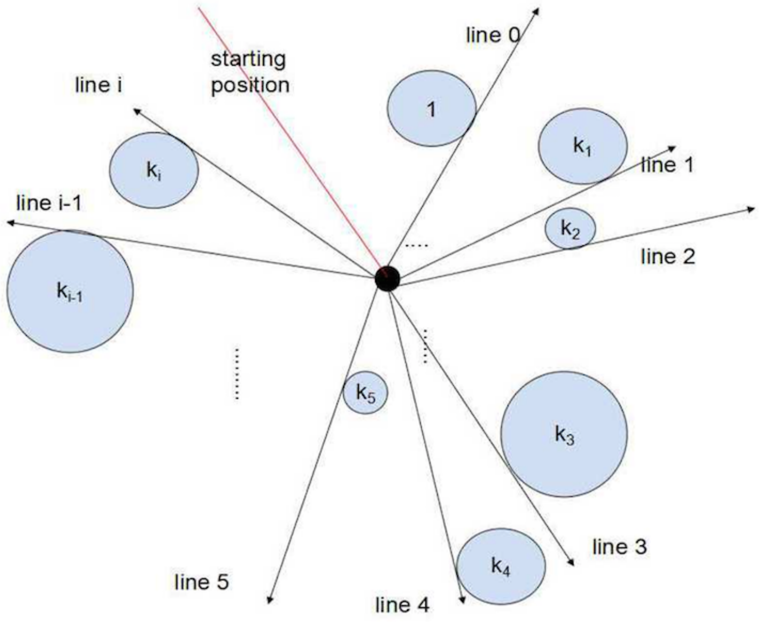

Figure 7

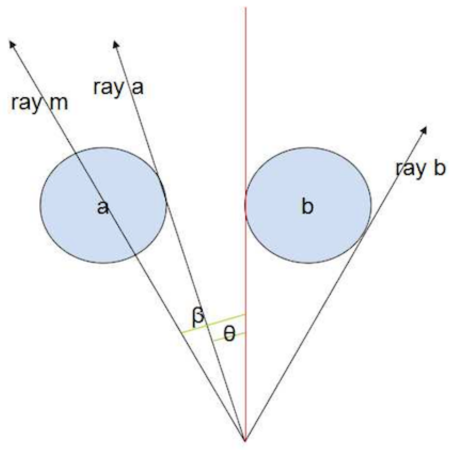

Figure 8

15: return $rayNum$;

Claim:1.*right tangent* represents the tangent of a circle that will leave the circle soon and $tangent[1..n]$ store these tangents. 2.The *corresponding angle* of a circle is the angle between the positive Y axis and the tangent(the tangent is the destination of the angle) and the algorithm sorts the angle from smallest to largest. As Figure 5 shows, line 1's *corresponding angle* β is smaller than line 2's *corresponding angle* θ and line 1 should be put in front of line 2 in $tangent[1..n]$. 3.In this case, we could consider the tangent of a circle as intersecting with the circle.4.At step 4, the algorithm permutes the order of $circle[1..n]$ according to the order of the circles in $tangent[1..n]$. 5. The subroutine INTERSECTS(r,c) determines whether an arbitrary ray r intersects an arbitrary circle c in $O(1)$ time.

Proof. Figure 6 could help prove. The red line is the starting position(positive Y axis).Firstly, let's prove that rays produced by this algorithm could intersect all the balloons.
For any ray j, if there is a circle(let's define it circle t) between the starting position and ray j(ray j is destination) and $ray[1..j]$ don't intersect it. It's obviously impossible. Suppose circle t is between ray m and ray $m+1$. Since ray m couldn't intersect it, the algorithm will produce ray $m+1$. Ray $m+1$ will stop its rotation once it becomes the *right tangent* of the first circle(let's define it circle k_{m+1}). So circle t is between ray m and $tangent[k_{m+1}]$, it implies that ray $m+1$ leaves circle t before circle k_{m+1} which is a contradiction. Hence, the circles before ray j have all been intersected.
Let's suppose the algorithm produces i rays in total as the figure shows. If there is still circle hasn't been intersected, its *right tangent* could only exist after ray i in the clockwise direction, and the algorithm will continue to produce a new ray until all the circles have been intersected. So the algorithm could guarantee all the circles intersected.
Now prove that the algorithm will produce the fewest rays i. Let's reserve these circles tangent with the i rays(if there are several circles tangent with ray m, then reserve the one that doestn't intersect with ray $m-1$) and delete the others. Then there will be at least i circles reserved. As Figure 7 shows.
We need at least i rays to intersect them. If we use fewer rays, it means at least two circles(let's define them circle a and circle b, and the corresponding tangent ray are ray a and ray b) among the i circles could be intersected by one ray(define it ray m). It's impossible since there is a gap between ray a and circle b. If ray m intersects circle a, then ray m's *corresponding angle* should be smaller than ray a's and it implies there is a gap between ray m and circle b as Figure 8 shows. If ray m's *corresponding angle* is larger than ray a's, then there is a gap between circle a and ray m. That's impossible. So we need at least i rays to intersect the i circles. And i rays could intersect all the circles as shown above, so the algorithm could solve the minimum zap problem correctly.

□

*Analysis.*Step one and step two only costs $O(n)$ time to find the start position, adjust the coordinates of circles and produce each circle's tangents. Step 3 uses QUICKSORT to sort $tangent[1..n]$, its time complexity is $O(nlogn)$. Step 6 to step 14 is the major part of this algorithm and it just goes through every circle and there is no backtracking. The time complexity of operations including the subroutine INTERSECTS(R,C) are all $O(1)$, so the time complexity of going through all the circles is $O(n)$. Hence the time complexity of the whole algorithm is $O(nlogn)$. We use $tangent[1..n]$ to store the *right tangent* of circles and a variant $rayNum$ to record the rays needed, $current\_position$ to record the starting positions for each ray. So the space complexity is $O(n)$.

(b)

Solution: For this case, we just make a small adjustment of the algorithm of problem of part(a). For the step one of algorithm 2, it will find a starting position where the ray will not hit any balloons and its premise is such a position should exist. Now there is no such a premise, and we just delete the first step of algorithm 2.

```
Require: circle[1..n], each element in the array is a structure storing the (x, y) coordinates and
    radius of n circles in the plane.
Ensure: The minimum number of ray to intersect every circle in C.
1: Start from the coordinate axis's origin(0, 0) and produce right tangents for every circle, that is,
   tangent[1..n];
2: ...
3: ...
4: ...
5: return rayNum;
```

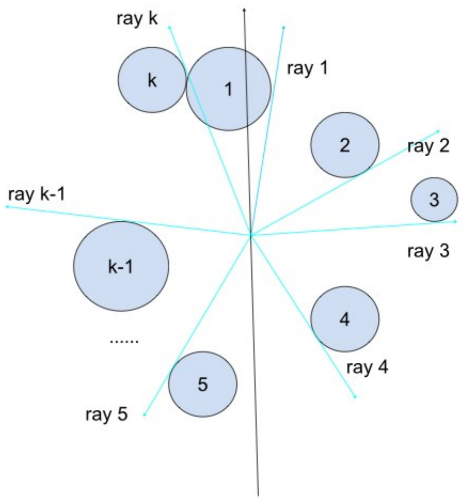

Figure 9

This algorithm's output is within 1 of the optimal.

Proof. Suppose the algorithm produces k rays. Firstly, reserve the circles tangent with the m rays and delete others. If there are several circles tangent with a specific ray h and then reserve only one that doesn't intersect ray $h-1$. It's bound to exist such a circle, otherwise ray h won't exist. Then there are at least $k-2$ circles separately. As Figure 9 shows, circle 2, circle 3,... and circle k can't be hit by a ray which has been shown in part(a). Circle k and circle 2 can't be hit a same ray as well. Since ray k is on the left of positive Y axis, otherwise, the algorithm will produce ray k first rather than the last one. And circle 2 is on the right of positive Y axis since ray 1 is on the right of positive Y axis and circle 2 is bound to be on the right of ray1, which implies that circle 2 is on the right of positive Y axis. So there is gap between ray k and circle 2 and there is also a gap between ray 2 and circle k. So we need at least $k-1$ rays to hit the $k-1$ balloons. For circle 1, if it can't be hit by ray k, it implies that we need at least k rays to hit the k balloons and the k rays could hit all the balloons which has been proved in part(a), so k is the optimal solution m. If the balloons(circle) hit by ray 1 including circle 1 could be hit by other rays, e.g. ray 2, ray k, it implies that ray 1 is no need and $k-1$ is the optimal solution. It's obviously possible for the other rays hit these circles. For circle 1, since we choose positive Y axis as the starting position and it's a random position actually and there may be no gap between ray k and circle 1, then circle 1 will be hit by ray k. Hence, $k = m + 1$. So the algorithm could produce a solution within 1 of the optimal. □

*Analysis.*The time complexity and space complexity of this algorithm is the same as that in part(a) obviously. The time complexity is $O(nlogn)$ and the space complexity is $O(n)$.

(c)

Solution:

Algorithm 3: minimum zap with no gap between balloons existing

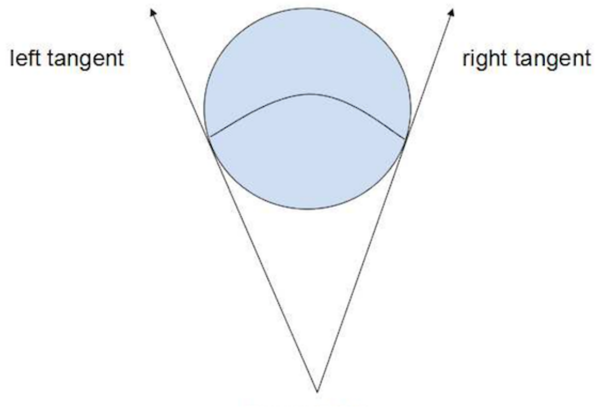

Figure 10

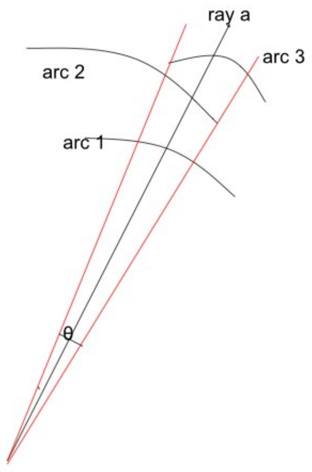

Figure 11

```
Require: $circle[1..n]$, each element in the array is a structure storing the $(x, y)$ coordinates and
    radius of n circles in the plane.
Ensure: The minimum number of rays to intersect every circle in C.
1: Start from the coordinate axis's origin$(0, 0)$ and produce right tangents,that is $tangent[1..n]$;
2: Use QUICKSORT to sort $tangent[1..n]$ according to the corresponding angle and permute the
   order of $circle[1..n]$;
3: Calculate the left tangents of $circle[1..n]$, that is, $left\_tangent[1..n]$;
4: $max = 0$;
5: for $i = 1$ to $n$ do
6:     $rayNum = 0$;
7:     $current\_position = left\_tangent[i]$;
8:     Produce a ray m with the starting point$(0, 0)$ in the $current\_position$;
```

```
    $first\_ray = 1$;
    for $i = 1$ to $n$ do
      if $tangent[i]$ is on the right of $current\_position$ in the clockwise direction then
        $first\_ray = i$;
        break;
      end if
    end for
    $i = 0$;
    while $(first\_ray + i)$ mod n! $= first\_ray - 1$ do
      $rayNum++$;
      while INTERSECTS($ray\ m, circle[first\_ray + i]$) $== true$ do
        $i++$;
      end while
    end while
    $max = max\{max, rayNum\}$;
  end for
  return max;
```

Claim:1.*right tangent* represents the tangent of a circle that will leave the circle soon and $tangent[1..n]$ stores these tangents. 2. The *left tangent* represents the tangent of a circle that will intersect the circle soon in the clockwise direction and $left\_tangent[1..n]$ stores these tangents. 3.The *corresponding angle* of a circle is the angle between the positive Y axis and the tangent(the tangent is the destination of the angle) and the algorithm sorts the angle from smallest to largest. As Figure 10 shows, line 1's *corresponding angle* β is smaller than line 2's *corresponding angle* θ and line 1 should be put in front of line 2 in $tangent[1..n]$. 4.In this case, we could consider the tangent of a circle as intersecting with the circle. 5.At step 2, the algorithm permutes the order of $circle[1..n]$ according to the order of the circles in $tangent[1..n]$. 6. The subroutine INTERSECTS(r,c) determines whether an arbitrary ray r intersects an arbitrary circle c in $O(1)$ time.

Proof. Firstly, we could consider every circle as an arc starting from *left tangent* of this circle and ending at *right tangent* as Figure 10 shows. Then there will be many arcs centering $(0, 0)$. Suppose there is an optimal solution$\{ray\ a1, ray\ a2, ..., ray\ a_{k-1}, ray\ ak\}$. Suppose ray h could hit g arcs, including $c1, c2, ..., cg$, then there must be an angel interval within which ray a1's rotation won't affect the result. As Figure 11 shows.
We could rotate a within the angle θ without affecting the result. The angle interval θ is the smallest interval that ray could hit the circles that it hits in the optimal solution. The left bound of θ is the *left tangent* a circle and the right bound of θ is the *right tangent* of a circle. If we rotate the ray from the left bound of θ, then the first *right tangent* it meets is just the right bound of θ.
If ray a1's corresponding rotation scope θ 's right bound is *right tangent w* and the left bound is *left tangent v*. Since the algorithm tries all *left tangent* as the starting position, it's bound to try *left tangent v* as a starting position. When starting at *left tangent v*, it produces ray b1 and the first *right tangent* b1 meets is *right tangent w*. So the algorithm will stop the rotation of ray b1 at the position of *right tangent w*. Hence, ray b1 could hit same circles as ray a1 since *right tangent w* is within the rotation scope of ray a1. Now the algorithm gets a ray of the optimal solution and suppose the algorithm starting at *left tangent v* will produce solution $\{ray\ b1, ray\ b2, .., ray\ b_{m-1}, ray\ b_m\}$. Since the rays in the optimal solution all have a rotation scope within which the result is identical, let's rotate all the rays to the right bounds of their rotation scopes. Let's make a hypothesis that ray a_i is the same as ray b_i. For the optimal solution, ray a_{i+1} is obviously the first *right tangent* on the right of ray a_i. The algorithm will produce a ray after ray b_i stops and rotate the new ray until it meets the first *right tangent*. So b_{i+1} is the same as a_{i+1}. If $m > k$, it implies that there are some still circles after ray b_k, the algorithm continues to produce a new ray. Since ray b_k is the same with ray a_k, it implies the optimal solution couldn't hit all the circles, since there are circles

left after a_k. It's a contradiction. So $m \leq k$. If $m < k$, since there are still rays after ray a_m, then it implies there are still circles after b_m, the algorithm will continue to produce new rays until $m = k$. So the algorithm could produce the optimal solution. □

*Analysis.*The algorithm tries all n circles' *left tangents* as starting position and for each try, it evaluates all the other circles' *right tangents*. So the time complexity is $O(n^2)$. The algorithm add the array $left\_tangent[1.n]$ comparing with the former algorithm 2, so the space complexity is $O(2n + c) = O(n)$.