CP8201/CPS815 Advanced Algorithms

Assignment 2

Rohaan Ahmed
PhD Student
Department of Computer Science
rohaan.ahmed@ryerson.ca

October 31, 2020

Instructor: Dr. Javad (Jake) Doliskani
October 31, 2020

1 and 3: Algorithm to find the largest subset of disjoint lines

In order to solve this problem, we will first reduce it to a problem for which the solution is already known. We will do this by representing the line segments as intevals spanning 0 to 2π radians.

Section I. Reduction: Representing Line Segments as Intervals

1. Given a line segment $L = (p_a, pb)$, where p_a and p_b are the start and end points of L, respectively, we will draw an abstract line from the center of the circle to points p_a and p_b, and calculate the angles θ_a and θ_b to these points. Illustrated in Figure 1

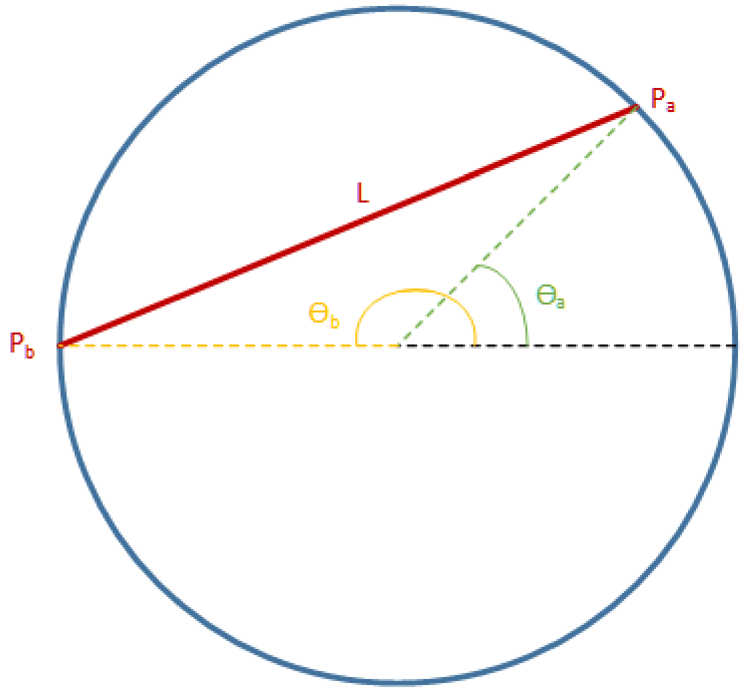

Figure 1:

2. We will then draw an interval, I_x between points θ_a and θ_b on the line spanning 0 to 2π radians. Illustrated in Figure 2

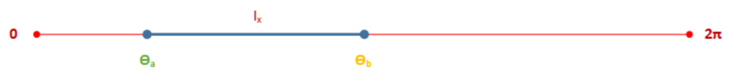

Figure 2:

We will perform steps (1) and (2) for each line segment in S, resulting in a list of intervals spanning 0 to 2π. Illustrated in Figure 3

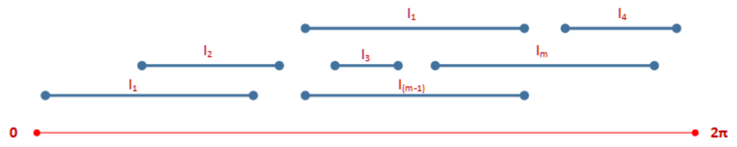

Figure 3:

We have now successfully reduced our problem to a well known problem known as *Interval Scheduling*. By employing this strategy of dividing the problem up into two simpler problems, we have used the *Divide And Conquer* technique.

In order to find the maximum disjoint set, we will solve the Interview Scheduling problem with the objective of *Maximizing the number of Non-Intersecting Intervals*. The Algorithm is shown below:

```
Function: reduce_to_interval_scheduling_problem($S, P$)
  for each $L$ in $S$ do
    Calculate angle $\theta_a$ between start point, $p_a$, and $0rad$ line
    Calculate angle $\theta_b$ between end point, $p_b$, and $0rad$ line
    {// $\theta_a$ to $\theta_b$ is a new Interval}
    $I \leftarrow (\theta_a, \theta_b)$
    Delete $L$ from $S$
  end for
  return $I$
```

I$= \{I_1, I_2, ..., I_m\}$ will contain the set of intervals obtained after reduction.

Section II. Interval Scheduling for Maximizing the number of Non-Intersecting Intervals

Interval Scheduling is a well known problem with several known solutions depending on the objectives, such as minimum finish time. Here, we will pursue the objective of *Maximizing the number of Non-Intersecting Intervals*. The Algorithm for this is shown below:

```
Function: interval_scheduling($I = \{I_1, I_2, ..., I_m\}$)
  Sort the intervals in $I$ in ascending order of their end points, still denoted as $I$
  while $I$ is not empty do
    Select first interval $I_x$ in $I$
    $R \leftarrow I_x$
    {// This is the 'Greedy' part of this Greedy Algorithm}
    Delete all intervals from $I$ which intersect with $I_x$
  end while
  return $R$, $r \in \mathbb{Z}^+$
```

R will contain the set of intervals representing the *Maximum Number of Non-Intersecting Intervals* (Answer to Q3), and ***r*** will be the number of intervals contained within R (Answer to Q1).

Reference: The above algorithm was adapted from the textbook *Algorithm Design. By Jon Kleinberg and Éva Tardos*

Complexity Analysis

The algorithm presented above has two subsections which are executed one after another. Therefore, the worst case time complexity of this algorithm is the maximum of the worst case time complexity of each of the two algorithms

1. The Reduction step loops through each line segment in the circle once. Thus, it is $O(m)$.
2. The Interval Scheduling algorithm performs in $O(mlogm)$ time, driven by the sorting step performed at the beginning.

Thus, the algorithm presented above has a time complexity of $O(mlogm)$. Below we show that this implies that the algorithm is also $O(mn)$.

$$\begin{align}
m &\leq \frac{n(n-1)}{2} \tag{0.1}\\
m &\leq n^2 \tag{0.2}\\
\implies log(m) &\leq 2log(n) \tag{0.3}\\
\implies log(m) &\leq 2n \tag{0.4}\\
\implies mlog(m) &\leq 2mn \tag{0.5}\\
\implies mlog(m) &= O(mn) \tag{0.6}\\
&\tag{0.7}
\end{align}$$

where:
m: Number of Line Segments in $S = \{L_1, ..., L_m\}$
n: Number of Points in $P = \{p_1, ..., p_n\}$

Equation 0.1 follows from the definition of a simple connected graph, and Equation 0.6 follows from the definition of Big-O notation. Therefore, the algorithm presented above has a time complexity of $O(mn)$.

2: Algorithm to find the largest mutually crossing subset

We once again use the Reduction algorithm in the previous question to reduce the problem to an Interval Scheduling problem. An example of this is shown in Figure 4.

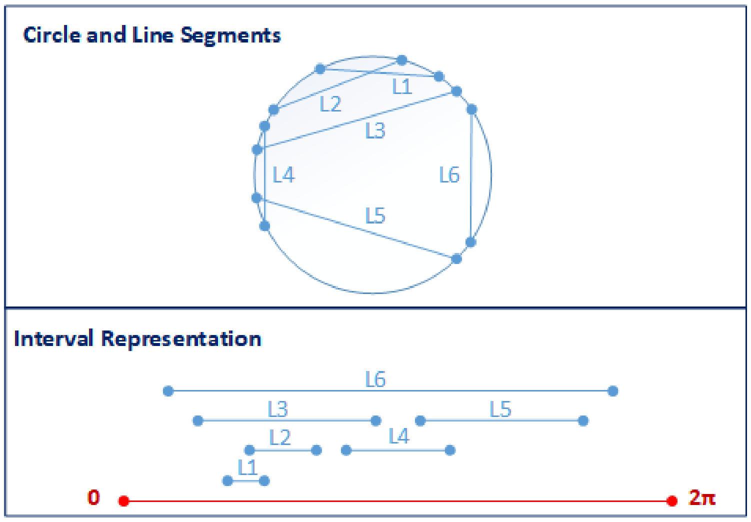

Figure 4:

This time, we will pursue the objective of *Maximizing the Number of Mutually Intersecting Intervals*, which, in turn, will give us the maximum number of mutually intersecting line segments. We will do this by considering three different cases illustrated in Figure 5. By employing this strategy of dividing the problem up into simpler problems, we have used the *Divide And Conquer* technique of problem solving. The Algorithm is shown below:

```
Function: interval_scheduling($parent\_interval\_end = 2\pi, L = \{L_1, L_2, ..., L_m\}$)
  // $L$ is pre-sorted in ascending order by start points, and sub-sorted by
  duration
  $R \leftarrow L[first]$
  $maxR = empty$
  for i from current $L$ to last $L$ and $L[i].end < parent\_interval\_end$ do
    if $R$ contains $L[i]$ then
      {// If the next L is completele contained within current R}
      $R \leftarrow max(R, interval\_scheduling(parent\_interval\_end = R.end, L[i, ..., m]))$
    else if $R$ crosses $L[i]$ then
      {// Else If the next L crosses current R}
      Find the furthest prior interval, $R[j]$ in $R$, which crosses $L[i]$
      Delete $R[1, ..., (j-1)]$ from $R$
      $R \leftarrow L[i]$
    else if $R$ does not cross with $L[i]$ then
      {// Else If L and R do not cross}
      $R = L[i]$
    end if
    $maxR = max(maxR, R)$
  end for
  return $max(R_k)$
```

The algorithm above returns the set, R, containing the largest mutually intersecting lines.

The algorithm below can be used to determine whether two intervals, $L1$ and $L2$, cross, contain each other, or don't overlap. The three cases are illustrated in Figure 5. The function executes in constant time.

```
Function: relation($L1, L2$)
  if $L1.start < L2.start$ and $L1.end < L2.end$ and L2.start ¡ L1.start then
    return L1 crosses L2
  else if $L1.start \leq L2.start$ and $L1.end \geq L2.end$ then
    return L1 contains L2
  else
    return L1 and L2 don't overlap
  end if
```

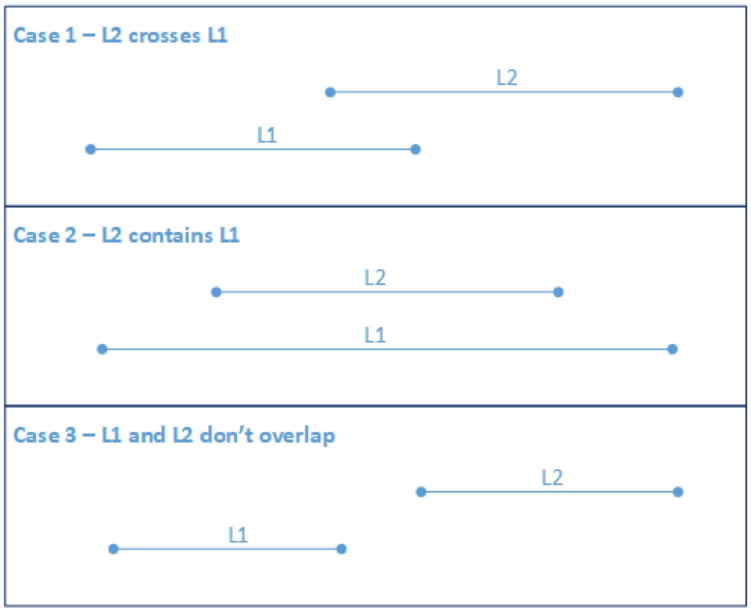

Figure 5:

Complexity Analysis

The algorithm presented above has three subsections which are executed one after another. Therefore, the worst case time complexity of this algorithm is the maximum of the worst case time complexity of each of the two algorithms

1. The Reduction step loops through each line segment in the circle once. Thus, it is $O(m)$
2. The sorting operation prior to Interval Scheduling, which performs in $O(mlogm)$ time
3. The Interval Scheduling problem, which traverses the list of intervals once, occasionally looking at previously collected $k < m$ intervals to find the furthest prior intersection (which can be done using hashing, iteratively or using Binary Search). Therefore, in the worst case, this code executes in $O(m)$ time.

Thus, the algorithm presented above has a time complexity of $O(mlogm)$. As shown previously, this implies that the algorithm is also $O(mn)$.

Extra: Algorithm to return the subset of all intersecting lines

This section was written when the author misunderstood the intent of Q2. It is kept here for In this section, we return the set of all intersecting lines, regardless of how many other lines they intersect. In other words, we delete all non-intersecting lines from the circle.
We once again use the Reduction algorithm in the previous question to reduce the problem to an interval scheduling problem. This time, we will focus on *Maximizing the Span of Overlapping Intervals*, which, in turn, will give us the maximum number of intersecting line segments.

Algorithm 1 Function: ***interval_scheduling***$(I = \{I_1, I_2, ..., I_m\})$

```
Sort the intervals in $I$ in ascending order of start times
For intervals with similar start times, subsort in ascending order of duration
while $I$ is not empty do
  Select first interval $I_x = (a_x, b_x)$ in $I$
  Set $Current\_Cluster\_Start = a_x$
  Set $Current\_Cluster\_Finish = b_x$
  for $k$ from 0 to Length of $I$ do
    Select next shortest interval, $I_y = (a_y, b_y)$ with $min(b_y - a_y)$, which satisfies criteria (i) and
    (ii):
    (i) $a_y > Current\_Cluster\_Start$
    (ii) $b_y > Current\_Cluster\_Finish$
    {// This is the 'Greedy' part of this Greedy Algorithm}
    $R_k \leftarrow I_x$
    Set $Current\_Cluster\_Finish = b$
    If criteria (i) and (ii) are not met: break
  end for
  Delete $R_k$ from $I$
end while
return $max(R_k)$
```

$max(R_k)$ will returns the list of intervals, the union of which, provides the *Maximum Span of Overlapping Intervals.*

Complexity Analysis
Similar to the previous question, this solution has two steps.

1. The Reduction step loops through each line segment in the circle once. Thus, it is $O(m)$.

2. The Interval Scheduling problem performs in $O(mlogm)$ time, driven by the sorting step performed at the beginning.

Therefore, the algorithm presented above has a time complexity of $O(mn)$.

Author's Note:

I would like to share credit for the above work with Katrina Hooper, a fellow student in CP8201. Discussing Q1 with her enabled me to realize that circle-and-lines can be reduced to intervals without losing semantic information. This served as a basis for my algorithms above.