15-451: Algorithms October 17, 2019

Lecture Notes: Computational Geometry: Introduction and Sweep Line

Lecturer: Gary Miller *Scribe:*

[1]

1 Introduction

Computational geometry is a branch of computer science which concerns design and analysis of problems that arise in geometric settings; in particular, over 2D or 3D Euclidean spaces which are extremely powerful mathematical tools for modeling real-world problems. Computational geometry techniques are highly applicable in computer science related areas including but not limited to:

- Computer Graphics
  - Images creation
  - Hidden surface removal
  - Illumination
  - Physics-based simulations
- Robotics
  - Motion planning
- Geographic Information Systems
- Computer Aided Design/Computer Aided Manufacturing (CAD/CAM)
- Computer chip design and simulations
- Scientific Computation
  - Blood flow simulations
  - Molecular modeling and simulations

In this course, the following classic ideas that are frequently used for dealing with geometric problems will be covered:

- Sweep-Line
- Divide and Conquer
- Random Incremental

[1]Originally 15-750 notes by Amirbehshad Shahrasbi

2 Basic Definitions

2.1 Geometric Objects

The following simple geometric objects in the 2D plane can be represented as follows:

1. **Point**: A point can be simply represented by a pair of real numbers (x, y) which correspond to its coordinates.

2. **Line**: A line may be represented by two points on it, or, more efficiently, by a pair of real numbers, namely, the slope and the intercept.

3. **Line Segment**: A line segment can be represented by its two end points.

4. **Polygon**: A polygon can be represented by an ordered array of its vertices.

Further, the following definitions can be useful in representing more complex geometric objects:

1. **Linear Combination**: The *Linear Combination* of $P_1, \ldots, P_k \in \mathbb{R}^d$ is defined as:

$$\left\{\sum \alpha_i P_i : \alpha_i \in \mathbb{R}\right\}$$

   Taking all linear combinations of a set of points forms the minimal subspace that includes all these points.

2. **Affine Combination**: The *Affine Combination* of $P_1, \ldots, P_k \in \mathbb{R}^d$ is defined as:

$$\left\{\sum \alpha_i P_i : \alpha_i \in \mathbb{R}, \sum \alpha_i = 1\right\}$$

   Taking all affine combinations of a set of points forms the minimal affine subspace that includes all these points.

3. **Convex Combination**: The *Convex Combination* of $P_1, \ldots, P_k \in \mathbb{R}^d$ is defined as:

$$\left\{\sum \alpha_i P_i : \alpha_i \in \mathbb{R}, \sum \alpha_i = 1, \alpha_i \geq 0\right\}$$

   Taking all convex combinations of a set of points forms the minimal polygon that covers all these points.

2.2 Primitive Operations

We now depart to introduce primitive operations on geometric objects:

1. **Point Equality**: In order to check if two points are equal, one can simply check the equality of their coordinates.

2. **Line Segment Intersection**: This operation has to output the intersection of two given line segments (P_1, P_2) and (P_3, P_4) or report that they do not intersect. Note that, as previously mentioned, the line between P_i and P_j can be described as:

$$\alpha P_i + (1 - \alpha) P_j, \text{ for } 0 \leq \alpha \leq 1$$

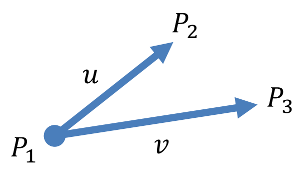

Figure 1: Line side test.

Therefore, if (P_1, P_2) and (P_3, P_4) intersect, their intersection is the solution of the following equation system:

$$\alpha P_1 + (1-\alpha)P_2 \tag{1}$$
$$\beta P_3 + (1-\beta)P_4 \tag{2}$$
$$0 \leq \alpha \leq 1 \tag{3}$$
$$0 \leq \beta \leq 1 \tag{4}$$

This can simply be done by a constant-sized matrix inversion.

3. **Line Side Test**: Note that in a 2D plane, any vector P_1P_2 splits the plane into two half-planes, i.e., its right-hand-side half-plane and its left-hand-side half-plane. Assume that P_1, P_2, and P_3 are given. Then, P_3 is on the left-hand-side half-plane made by P_1P_2 if and only if:

$$\det \begin{vmatrix} u_x & v_x \\ u_y & v_y \end{vmatrix} > 0$$

where $u = (u_x, u_y) = P_2 - P_1$ and $v = (v_x, v_y) = P_3 - P_1$ as depicted in Figure 1.

3 Line Segment Intersection Problem

We now present the *line segment intersection problem* and provide a solution using the sweep-line idea. Imagine that a set of n line segments as $S = \{S_1, \cdots, S_n\}$ are given and we want to find I, the set of all pairwise intersections of segments in S. For the sake of simplicity, we assume that the segments in set S do not form the following special cases:

1. A horizontal segment.

2. More than two segments intersecting at the same point.

3. A segment having an end point within another one.

4. Two segments sharing an endpoint.

A schematic representation of these forbidden cases can be found in Figure 2.

As we mentioned in Section 2, one can find the intersection of two line segments in $O(1)$ time. Hence, a trivial solution for our problem would be simply checking if any two line segments have an intersection or not. Clearly, this will take $O(n^2)$ time. Furthermore, note than one cannot really hope for a faster algorithm in terms of n as the input may contain as many as $\Theta(n^2)$ intersection points which will make the output size as large as $\Omega(n^2)$. However, we can find improved solutions considering an output-sensitive analysis of solutions.

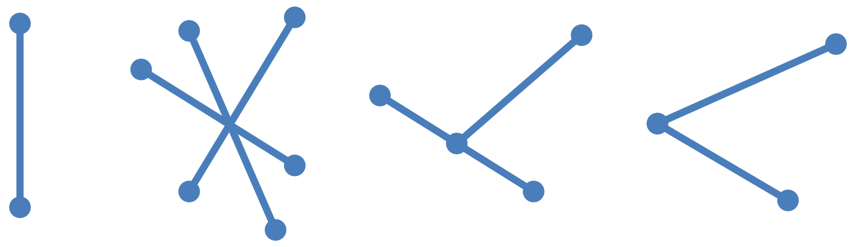

Figure 2: Forbidden cases

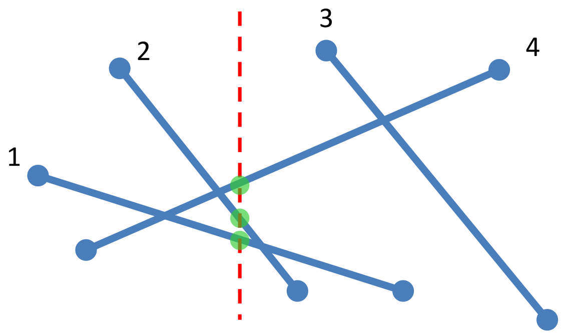

Figure 3: Status : `4, 2, 1`

An *output-sensitive algorithm* is an algorithm whose consumption of resources like running time depends on the size of its output. In our case, we will seek output sensitive algorithms that can potentially outperform the trivial solution for smaller output sizes. Note that the trivial algorithm is not output-sensitive. In fact, regardless of $|I|$, the trivial solution will take $\Theta(n^2)$ to run. However, considering output-sensitive analysis, one can find algorithms with following running times:

- Sweep-Line: $O\left((n + |I|)\log n\right)$
- Random Incremental: $O\left(|I| + n\log n\right)$

We now introduce and analyze a solution using the sweep-line idea. The high-level idea for this solution is having a horizontal line far left on the plane and scanning –or "sweeping"– the plane by moving that line gradually to the right and capturing intersections as we pass through them.

In order to formalize the algorithm, we define the *status* for our sweeping line at any point of the sweeping procedure when it does not lay over any endpoints or intersections. The status of the sweeping line is defined as an list of the IDs of all segments intersecting it sorted from the top to the bottom. As an example, in Figure 3, the status of the sweeping line is `4, 2, 1`.

Let P be the set of endpoints of segments in S. The sweep line idea is simply trying to keep track of how the status changes as it scans the whole plane. Note that the status only changes whenever the sweeping line passes through an endpoint or an intersection, i.e., members of $P \cup I$.

For the moment, assume that the set I is given and we only seek to track status changes. Clearly, if I is given, we can simply insert $P \cup I$ into priority queue Q which prioritizes points with smaller x coordinate and then repeatedly extract the minimum element of Q and update the status according to whether that element is an endpoint or an intersection. Let us refer to members of

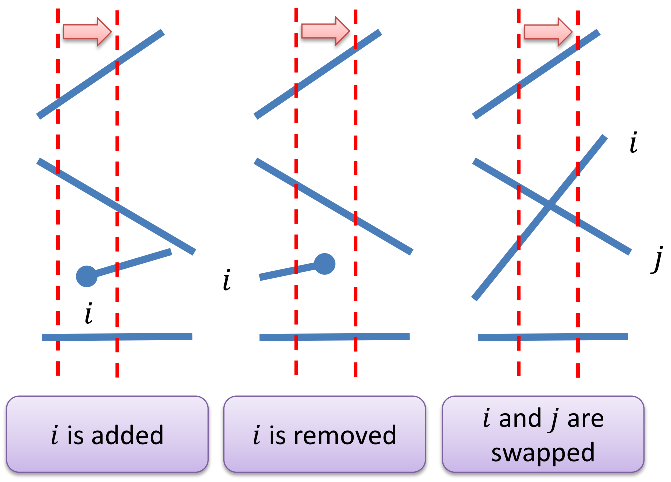

Figure 4: How status should be updated based on the type of the event.

$P \cup I$ which are points where status changes as *events*. Then, at the occurrence of an event, the status changes as follows (see Figure 4):

- If the next event is a starting point: the ID of the corresponding line has to be inserted into the status list.
- If the next event is a finishing point: the ID of the corresponding line has to be removed from the status list.
- If the next event is an intersection point: IDs of corresponding lines have to be swapped in the status list.

The idea that turns the "status tracking" algorithm into a solution for line segment intersection problem is simply starting with $Q \leftarrow P$ and adding intersection events, i.e., members of I, as the sweeping line passes through the segments. In order to do so, we make use of the following essential lemma:

Lemma 3.1. *If next event of the sweeping line is the intersection of lines l_1 and l_2, then l_1 and l_2 have to be two consecutive elements in the status.*

Proof. Clearly, l_1 and l_2 have to be present in the status since their start points must occur before intersection and endpoints must occur after intersection. Further, if by contradiction, there is a line l_3 between l_1 and l_2 in the status, that line has to either finish or cross one of l_1 and l_2 lines before l_1 and l_2's intersection. This contradicts the fact the l_1 and l_2's intersection is the closest event and finishes the proof. $\square$

Note that in order to have the "status tracking" algorithm to work without prior knowledge of I, we only need to have to make sure that the potential intersection of all consecutive pairs of lines in the status are available in Q at any point of the algorithm. Hence, we start with $P \leftarrow Q$ and insert the intersection of *new* consecutive pairs in the status whenever status changes. This way, we can generate segment intersections as we scan the plane. For a formal description of this algorithm please see Algorithm 1.

Algorithm 1 Sweep-Line Segment Intersection Algorithm

```
Input: $S$
Output: $I$
  $P \leftarrow$ Endpoints of $S$
  $I \leftarrow \emptyset$
  status $\leftarrow \emptyset$
  $Q \leftarrow P$
  while $Q \neq \emptyset$ do
      $E \leftarrow$ ExtractMin($Q$)
      HandleEvent($E$)
  end while
```

Algorithm 2 `HandleEvent`

```
Input: Event $E$
  if $E$ is the starting point of $l$ then
      insert($l$, status)
      addIntersection(left($l$), $l$, $Q$)
      addIntersection($l$, right($l$), $Q$)
  else if $E$ is the finishing point of $l$ then
      addIntersection(left($l$), right($l$), $Q$)
      delete($l$, status)
  else if $E$ is the intersection of $l < l'$ then
      swap($l$, $l'$, status)
      addIntersection(left($l'$), $l'$, $Q$)
      addIntersection($l$, right($l$), $Q$)
      insert($E$, $I$)
  end if
```

Note that there are $2n + |I|$ many events and handling each of them may take up to $O(\log n)$ due to the priority queue operations. Therefore, the overall time complexity is $O\left((n + |I|) \log n\right)$.