Implementation of simple arrangement of lines in 2D plane with the application of finding cells intersected by a line

Abhiram Chilukuri

Abstract—Given a set of n lines $L \in \mathbb{R}^2$, We need to compute the arrangement A(L) using incremental construction. In this report, we'll understand the C++ implementation and the use of DCEL data structure to store the cells, vertices, and faces. Zone's theorem puts an upperbound to the toal number of edges of all cells intersected by a query line. The upperbound is $6n$. We can therefore find all the cells intersected by a query line in $O(h) + O(\log(n))$, where h is the number of cells intersected and the $O(\log n)$ overhead is due to finding the starting point of the line in the cells. **We assume that no two lines are parallel and no 3 lines are concurrent**

Keywords—

1. Introduction

We start by discussing the DCEL structure. We then briefly discuss about the incremental algorithm and it's implementation. The main challenge is to carefully update the pointers in the edge-list to create new cells, point to the next edges, updating the edge-twins, etc.

2. DCEL Structure and the Arrangement class

The DCEL structure stores the records for three different entities — vertices, edges, and faces. The file "halfedge_structs.h" defines the DCEL. The file "arrangements.h" defines the "Arrangement" class. The member variables it contains are:

- list of DCEL edges
- list of DCEL vertices
- list of DCEL faces
- array of 4 elements describing the boundary positions
- and 4 balanced binary search trees, in the form of "sets" provided by C++ STL, for storing boundary edges

In this project, we set the boundary box as x=-1000, x=+1000, y=-1000, and y=+1000. Therefore, all the lines should intersect within this box. We can change the boundary box by providing the boundary positions as arguments to the constructor.

The use for the 4 sets come from the need to locate the first cell from which we can start our traversal of the line in the incremental algorithm. During the algorithm we need to partition the DCEL edges which includes the boundary DCEL edges as well. In the first step of the algorithm, we compute any one intersection point of the line with the boundary box. We then start our traversal from the cell containing (or touching) this point.

By using the tree (in the form of sets provided by C++ STL), we can find this starting cell in $O(\log n)$ time.

Member functions and their descriptions:

- getBoundaryIntersection — given a line, it computes the intersection point of the line with the bounding box
- findIntersection — given a line and a DCEL edge, it computes their intersection point
- getBoundaryEdge — given the line, it returns the boundary edge attached to the cell, from which we start our algorithm. This step takes $O(\log n)$ time for each line and a total contribution of $O(n \log n)$ to the total time
- findExit — returns the edge through which line l exits the cell attached to the edge e. This function should be only used if l enters the cell through e
- updateDCEL — this function does all the nasty pointer creations and updations in the DCEL records when a line cuts through a cell, creating extra cell and edges
- insertLine — Call this function whenever we need to add extra line to the 2D space. call this function n times and we are done creating the line arrangements!
- zoneQuery — returns all the cells incident to the line l. Uses the same logic as insertLine, but instead of creating new DCEL elements, it stores all the cells incident to the l

3. updateDCEL

This is the most challenging part of the algorithm to implement. Though it looks easy if we try to do the computation using hand, if not implemented carefully can lead to segfaults which are exhausting to trace back. Below are the rules I followed to implement this function.

Let v_1 and v_2 denote the vertices through which the line l enters and exits the current cell `curr_face` respectively. e_1 and e_2 be the edges bounding `curr_face` and passing through v_1 and v_2 respectively. Let `pass1` and `pass2` represent e_2 and its partition from the previous iteration respectively.

1. Updating DCEL edge records
    - Create two new DCEL edges e_1' and e_2' that represents the new edges created after partitioning e_1 and e_2. Set e_1'`->origin` $= v_1$, e_1'`->next` $= e_1$`->next`
    - Create a new DCEL face `new_face` and set e_1'`->face` = `new_face`.
    - Create new twin DCEL edges `new_edge` and `new_twin_edge`. Set `new_edge->origin` $= v_2$, `new_twin_edge->origin` $= v_1$, `new_edge->next` $= e_1'$, `new_edge->prev` $= e_2$, `new_twin_edge->prev` $= e_1$, `new_twin_edge->next` $= e_2'$, `new_edge->face` = `new_face`
    - Set e_1`->next` = `new_twin_edge`, e_1'`->prev` = `new_edge`
    - Set e_1`->twin` = `pass1`, e_1'`->twin` = `pass2`, `pass1->twin` $= e_1$ and `pass2->twin` $= e_1'$
    - Set e_2'`->origin` $= v_2$, e_2'`->next` $= e_2$`->next`, e_2'`->prev` = `new_twin_edge`, `e2->next` = `new_face`, e_2`->face` = `new_face`
    - Set e_1'`->next->prev` $= e_1'$, e_2'`->next->prev` $= e_2'$
    - Set the faces of all edges between e_1' and e_2 bounding `curr_face` to point to `new_face`
    - Set `pass1` $= e_2'$ and `pass2` $= e_2$
2. Updating DCEL Face records — easy
3. Updating DCEL Vertex records — easy

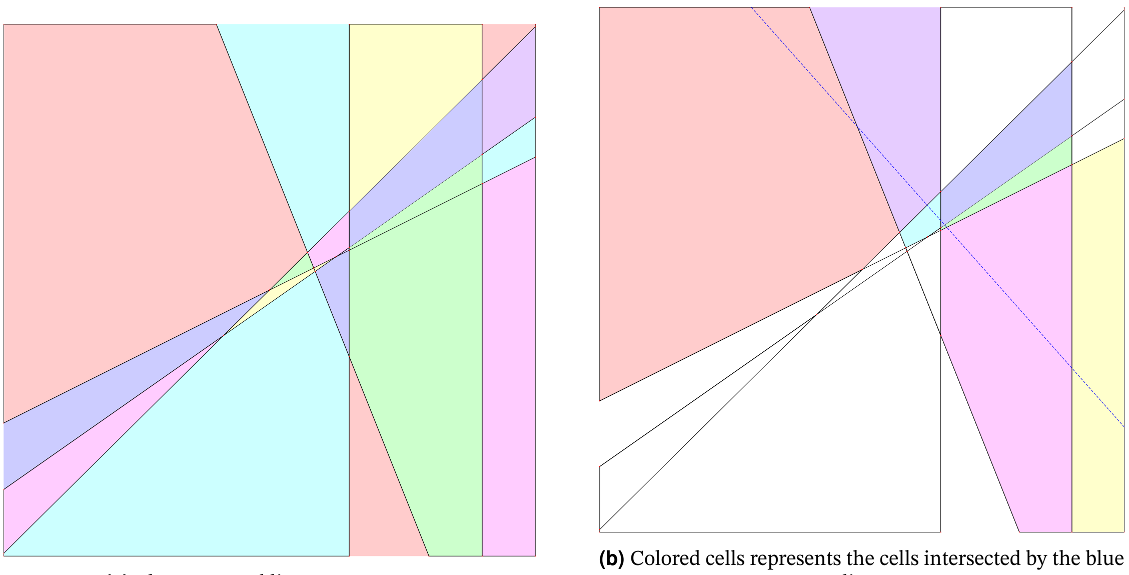
Besides, we also need to add e_1' to one of the sets maintained for boundaries if e_1' is the partition of the boundary edge e_1

The updates to the records are summarized in a table form in section 5

4. Example

Here's an example of the line arrangement computed by the code for the following lines.

- $p_0 = (0, 0)$, $p_1 = (1, 0.99)$, $p_2 = (2, 1)$
- $p_3 = (0, -50)$, $p_4 = (150, 55)$, $p_5 = (300, 0)$


(a) The computed line arrangements

(b) Colored cells represents the cells intersected by the blue line.

Figure 1. Line arrangements computed for the example in section 4. Note that some edges are missing from the boundary. That's because of a defect in SVG on the borders.

- $p_6 = (300, 300)$, $p_7 = (800, 0)$, $p_8 = (800, 200)$
- $p_9 = (600, -1000)$, $p_{10} = (200, 0)$
- l_0 =Line(p_0, p_1), l_1 =Line(p_0, p_2)
- l_2 =Line(p_3, p_4), l_3 =Line(p_5, p_6)
- l_4 =Line(p_7, p_8), l_5 =Line(p_9, p_{10})

To find all the cells intersected by a line, consider a query line l' =Line(z_1, z_2) where $z_1 = (1000, -600)$ and $z_2 = (200, 300)$

Figure 1 shows the computed A(L) and intersected cells.

5. DCEL Edges updates

5.1. Tables

Table **??** shows some of the major updates to DCEL Edge records. The entries under the column **name** are the edges that either need to be created or modified. The entries under all the other columns need to be found first, and then assign it to their respective entries under 1^{st} column

Table 1. Some Major updates in DCEL Edge records

Name	Origin	Twin	Next	Prev	Face
e_1	(old)	pass1	new_twin_edge	(old)	(old)
e'_1	v_1	pass2	e_1->next	new_edge	new_face
e_2	(old)	-	new_edge	(old)	new_face
e'_2	v_2	-	e_2->next	new_twin_edge	(old)
new_edge	v_2	new_twin_edge	e'_1	e_2	new_face
new_twin_edge	v_1	new_edge	e'_2	e_1	(old)

Note: The table doesn't show the updates to `pass1`, `pass2`.

6. Algorithm

Below is the pseudocode for the insertLine algorithm. Call the function n times, for each line $l \in L$ to complete the incremental algorithm

```
function insertLine(l)
    % Initialize the first two terms of the sequence
    e1 = getBoundaryEdge(l);
    v1 = getBoundaryIntersection(l);
    vertices.push_back(v1);

    pass1 = pass2 = e1->twin
    % for boundary edge, pass1 and pass2 values don't
     ↪ matter

    curr_face = e1->face;
    while curr_face:
        e2 = findExit(l, e1);
        v2 = findInterscetion(l, e2);
        vertices.push_back(v2);

        updateDCEL(curr_face, v1, v2, e1, e2, pass1,
     ↪ pass2);
        pass1 = e2->next->twin->next;
        pass2 = e2;
        v1 = v2;
        e1 = e2->twin;
    end

end
```

Code 1. insertLine algorithm.

7. Contact

✉ 210050042@iitb.ac.in