University of Trento
Spatial Databases

Project Report

Corte Pause Manuela
manuela.cortepause@studenti.unitn.it
240183

Contents

1 Introduction

The project aims at implementing a sweep line algorithm for computing the overlay of two planar subdivisions. To achieve this goal, the project is divided into three main parts: the implementation of the sweep line algorithm for computing the intersection of line segments, the implementation of the doubly connected edge list data structure, and the implementation of the overlay algorithm.

The project is implemented in Python and the code, along with some examples, is available on GitHub at the following link: `https://github.com/ManuelaCorte/Map-Overlay.git`. The project can be run from a simple command line interface that allows the user to either execute exclusively the sweep line algorithm for line segment intersection or the overlay algorithm.

2 Algorithm

2.1 Line Segment Intersection

A naive approach to line segment intersection is to compare each pair of segments and test them for intersection. This approach has a time complexity of $O(n^2)$ where n is the number of line segments.

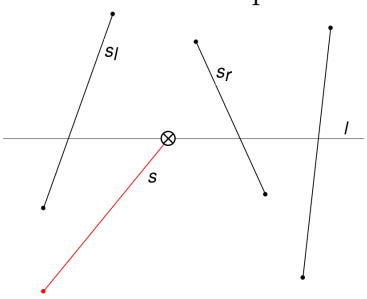
Generally, in the worse case scenario where all line segments intersect with each other, the lower bound for the time complexity is $\Omega(n^2)$ for any algorithm because all intersections must be reported but in real cases scenarios we usually have that a line segment intersects only a small subset of other segments. This type of algorithms where the complexity depends not only on the input size but also on the input itself are called **output-sensitive algorithms**.

We want an algorithm that performs better than the naive approach in these more common scenarios. The one we decided to implement is based on the **sweep line** paradigm. The idea is to move a horizontal line from top to bottom while keeping track of the segments currently intersecting it and only testing for intersection these segments.

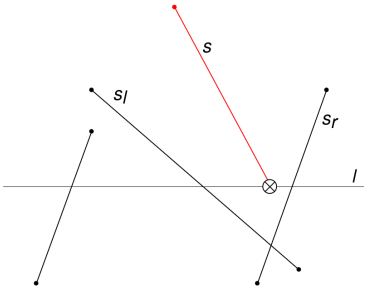
The algorithm uses two main data structures: the **event queue** and the **status structure**. The event queue is a priority queue that contains all the event points sorted by their y coordinate. The status structure is a balanced binary search tree that contains all the segments intersecting the sweep line sorted by their x coordinate.

The algorithm works as follows:

1. Initialize the event queue with all the endpoints of the segments.
2. While the event queue is not empty:
   (a) Pop the event point with the largest y coordinate from the event queue.
   (b) If the event point is the upper endpoint of a new segment s:
      i. Insert s in the status structure.
      ii. Check if s intersects the segment to its left s_l and the one to its right s_r in the status structure.



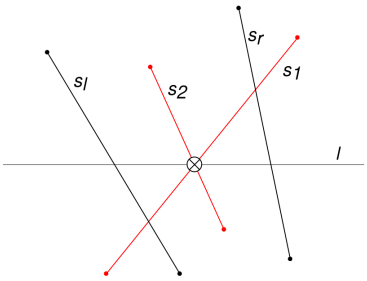
(c) If the event point is the lower endpoint of a segment s:



   i. Remove s from the status structure.

   ii. Check if the segments to the left and to the right of s (s_l and s_r respectively) which are now adjacent in the status structure intersect.

(d) If the event point is an intersection point between two segments s_1 and s_2:



   i. Report the intersection.

   ii. Swap the positions of s_1 and s_2 in the status structure.

   iii. Assuming s_1 is now to the left of s_2, test them for intersection with their new neighbors. For s_1 this means testing for intersection with the segment to its left (s_l) and for s_2 testing for intersection with the segment to its right (s_r).

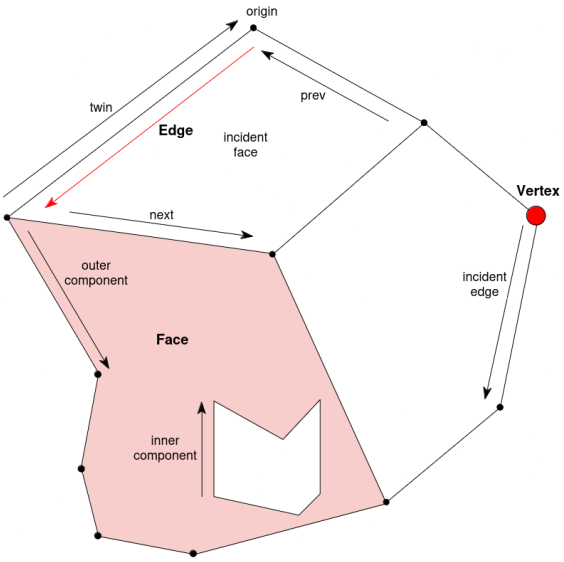
We can check that a segment is intersecting exclusively with the segments to its left and to its right because the segments in the status structure are sorted by their x coordinate. Since it can be proven that in order to intersect two segments must be adjacent at some point, they must be adjacent in the status structure as well.

The time complexity of this algorithm is $O(n \log n + m \log n)$ where n is the number of segments and m is the number of intersections. The first term comes from the construction of the event queue and the second term comes from the operations on the status structure which all have a time complexity of $O(\log n)$ and are performed m times.

2.2 Doubly Connected Edge List

The overlay task consists in computing the intersection of two planar subdivisions. The algorithm we implemented is derived from the sweep line algorithm for line segment intersection described above.

The first step is to define a data structure to represent these planar subdivisions. We chose to use the **Doubly Connected Edge List** (DCEL) data structure because it allows us to easily represent the planar subdivision as a set of vertices, edges and faces and to perform operations on it.



- **Vertex**: each vertex stores its coordinates and a pointer to one of the half-edges that starts at that vertex.
- **Edge**: each edge stores a pointer to its origin vertex, a pointer to its twin edge, a pointer to the previous and to the next edge in the boundary to and a pointer to the face it belongs to. We don't need to store the destination vertex because its equal to the origin of its twin edge.
- **Face**: each face stores a pointer to one of the half-edges that bounds it (this pointer is null for the unbounded face). This edge is chosen so that the face it bounds is always on its left side which implies we traverse the face in counter-clockwise order. It also stores a list of pointers to a half-edge for each hole in the face. Differently from the outer boundary, the holes are stored in clockwise order so that the face they bound is still on their left side.

2.3 Overlay of Two Subdivisions

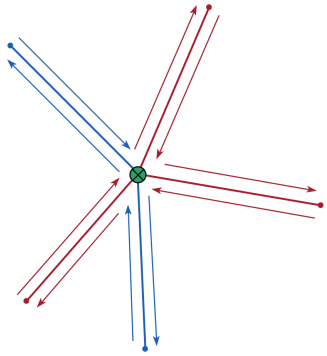
Definition (Overlay): the overlay of two planar subdivisions S_1 and S_2 is the planar subdivision S such that there is a face f in S if and only if there are faces $f_1 \in S_1$ and $f_2 \in S_2$ such that $f = f_1 \cap f_2$ is maximal. More simply, the overlay is the intersection of the two subdivisions.

We create the DCEL for the overlay by first copying the vertices and edges of the two input DCELs. The edges from the original subdivisions that don't intersect any other edge don't need to be modified while the edges that intersect other edges from a different subdivision need to be split at the intersection points.

Starting from the sweep line algorithm for line segment intersection we can easily extend it to compute the overlay of two subdivisions. The algorithm works as follows:

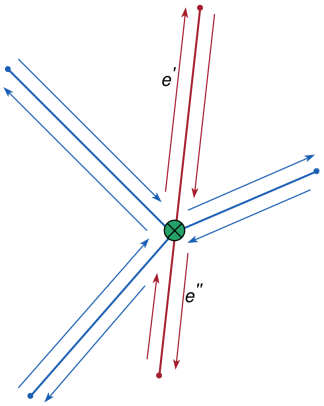
1. Initialize the event queue with all the endpoints of the edges of the two subdivisions.
2. While the event queue is not empty:
   (a) Pop the event point with the largest y coordinate from the event queue.
   (b) Update the status structure as in the line segment intersection algorithm.
   (c) If the event point is an intersection point between edges from different subdivisions we need to split the edges at the intersection point and update the DCEL. Three main cases can happen:
      i. The intersection point is an endpoint for all intersecting edges
      ii. The intersection point is an endpoint only for the edges belonging to one of the two subdivisions
      iii. The intersection point is an interior point for all intersecting edges

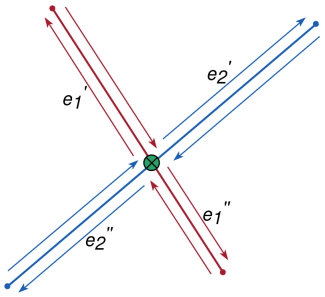
3. Reconstruct the faces of the overlay.



Intersection is an endpoint This first case is the simplest because no edge needs to be split and no vertex has to be added to the DCEL. The only thing we need to do is update the pointers of the edges around the intersection point v so that they point to the correct edges: the prev pointer for each of the incident edges of v must point to the twin of the first incident edge of v seen in counter-clockwise order. Respectively, the next pointer for each of the incident edges twins must point to the first incident edge of v seen in clockwise order.

Intersection is an endpoint for one subdivision We only need to split the edge e from the second subdivision into two edges e' and e'' both with the intersection point v as origin, together with their corresponding twins.
We need to update the pointers around the endpoints of the original edge e: the next pointers of the two new half-edges e' and e'' each copy the next pointer of the old half-edge that is not its twin, Respectively, the half-edges to which these pointers point must also update their prev pointer and set it to the new half-edges.
The situation around the intersection point v is the same as in the previous case where the prev pointers of the incident edges must point to the twin of the first incident edge in counter-clockwise order and viceversa.

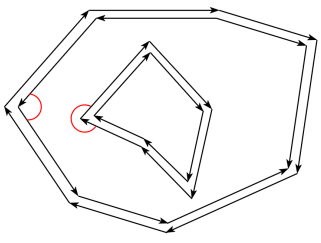




Intersection is an interior point We need to split both edges e_1 and e_2 into two half-edges and add a new vertex v to the DCEL which has as incident edges the new four split half-edges e_1', e_1'', e_2' and e_2''. The pointers around the endpoints of the original edges e_1 and e_2 must be updated as in the previous case.

After the intersection points have been processed, we need to reconstruct the faces by traversing the boundary cycles. We can do this by traversing the edges of the DCEL starting from an arbitrary edge and following the next pointers until we reach the starting edge again. This process can be repeated until all edges have been visited and all faces have been reconstructed.

We still need to decide if the faces found are outer faces or holes. We can determine this by checking the angle between the two incident edges of the leftmost wertex. If this angle is smaller than 180° then the cycle is an outer boundary, and otherwise it is the boundary of a hole.



In order to decide which boundary cycles bound the same face, we construct a graph where for each boundary, including the unbounded face, there is a node and an edge between is added between two nodes if and only if one of the cycles is the boundary of a hole and the other cycle has a half-edge immediately to the left of the leftmost vertex of that hole cycle. We can then find the connected components of this graph and assign the same face to all the cycles in the same component.

The last step is to label each face of the overlay with the attributes of the faces for the input subdivisions that intersect in that face.

The time complexity of this algorithm is $O((n)\log(n) + k\log(n))$ where $n = n_1 + n_1$ is the total number of edges in the two input subdivisions and k is the number of intersections between edges of the two subdivisions.

3 Implementation

3.1 Sweep Line Segment Intersection

The line segment intersection algorithm is implemented following the description in the theoretical section 2.1.

For the data structures, the implementation uses a Red-Black Tree as the event queue where the events are sorted based on descending y coordinate and ascending x coordinate. This is equivalent to having a sweep line moving from top to bottom and from left to right in case of events with the same y coordinate.

For the status structure, a simple list was used to store the segments that are currently intersecting the sweep line. The list is sorted based on the x coordinate of the intersection point with the sweep line. Initially, the status structure was implemented using a Red-Black Tree as well but updating the position of the segments every time the sweep line moved proved to be too complex. Moreover, the number of segments in the status structure is expected to be small, so the performance loss is only visible in worst case scenarios where a large number of segments all intersect the sweep line at the same time.

As a result, the overall complexity of the algorithm is worse than the theoretical complexity of $O(n \log n + k \log n)$, where n is the number of segments and k is the number of intersections. The complexity of the implementation is $O(n \log n + kn \log n) = O(kn \log n)$ because keeping the status structure sorted requires $O(n \log n)$ instead of the $O(\log n)$ of a balanced search tree.

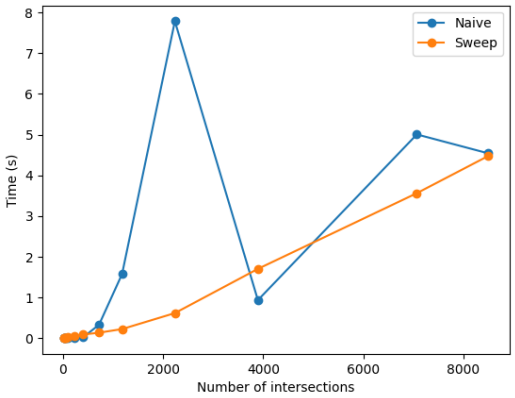
Another aspect that needed careful consideration was the handling of degenerate cases:

- **Horizontal Segments**: Horizontal segments are handled by sorting the events based on the x coordinate of the left endpoint of the segment. This ensures that the left endpoint is always processed before the right endpoint.

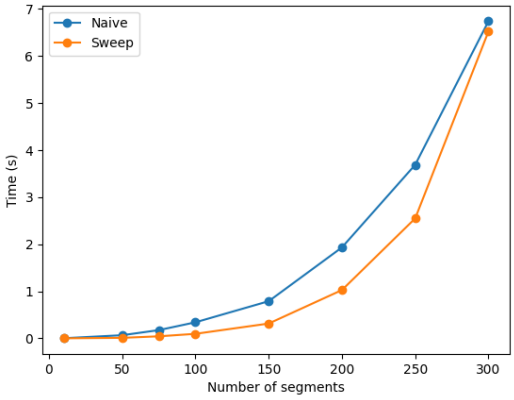
- **Colinear Segments**: Colinear segments are considered as intersecting at the endpoints. This requires special handling because differently from all other cases, the intersection point is not unique . The implementation manually adds the intersection points instead of relying on the algorithm to compute them.

Below we can compare the performance of the sweep line intersection algorithm and the naive algorithm. It is interesting to notice how in the comparison based on the number of intersections, the naive algorithm can sometimes outperform the sweep line

approach. This is due to the fact that the naive algorithm is not affected by the number of intersections but only by the number of input segments.



(a) Comparison based on the number of intersections



(b) Comparison based on the number of segments

3.2 Overlay of Two Subdivisions

The first step for the implementation of the overlay algorithm is the construction of the Doubly Connected Edge List for the input planar sibdivisions. Because the implementation was carried out in Python, keeping track of pointers between the different elements isn't as straightforward as in other lower-level languages. To overcome this, the implementation uses a series of dictionaries to store vertices, half-edges and faces while each element only contains the unique identifiers. This way, the pointers are replaced by the identifiers and the dictionaries can be used to access the elements on $O(1)$ time. Compared to storing the entire object, this approach is more memory efficient and less prone to errors because a single copy of the object is stored in memory.

```
class Vertex:
    id: VertexId
    coordinates: Point
    incident_edges: List[EdgeId]
```

```
class Edge:
    id: EdgeId
    origin: VertexId
    twin: EdgeId
    incident_face: FaceId
    next: EdgeId
    prev: EdgeId
```

```
class Face:
    id: FaceId
    outer_component: EdgeId
    inner_components: List[EdgeId]
    area: float
    info: str
```

Take for example the `Edge` class: the class contains its unique identifier, the identifier of the origin vertex, the edge ids of its twin, next and previous edges and the identifiers of the face it belongs to. If instead of storing the identifiers, the class stored the actual objects (e.g. `Vertex` instead of `VertexId`), the memory usage would be higher because

each object would contain a copy of the other objects it references. Moreover, the implementation would be more error-prone because changing an object would require updating all the references to it.

Some differences with respect to the theoretical data structure are present:

- **Vertex**: each vertex stores a list of incident edges instead of a single edge. This is not strictly necessary but it simplifies the implementation because it allows to easily access and update the edges incident to a vertex and this is done often during the overlay algorithm.

- **Face**: each face stores the area of the face and an info string in addition to the standard fields. The signed area is computed as follows

$$A = \frac{1}{2}\sum_{i=0}^{n-1}(x_i y_{i+1} - x_{i+1} y_i)$$

  and is used during the DCEL construction to determine the unbounded face: faces with a negative area are considered unbounded.

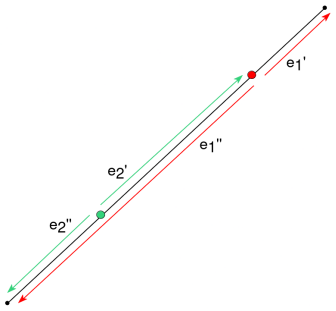
  The info string is used to optionally store information about the face. In the final overlay, the info field is used to store the input subdivisions that intersect in that face.

The overlay algorithm is implemented similarly to how its was described in the theoretical section with the major difference being that the overlay isn't computed during the sweep responsible to find the line segments intersections but after. This allows to keep the two step separate and more easily debug the algorithm without increasing the overall complexity.

Additionally, the line segment intersection algorithm is modified to keep track of each segment and in how many points it is intersected by other segments. This information is then used in the overlay computation for two purposes:

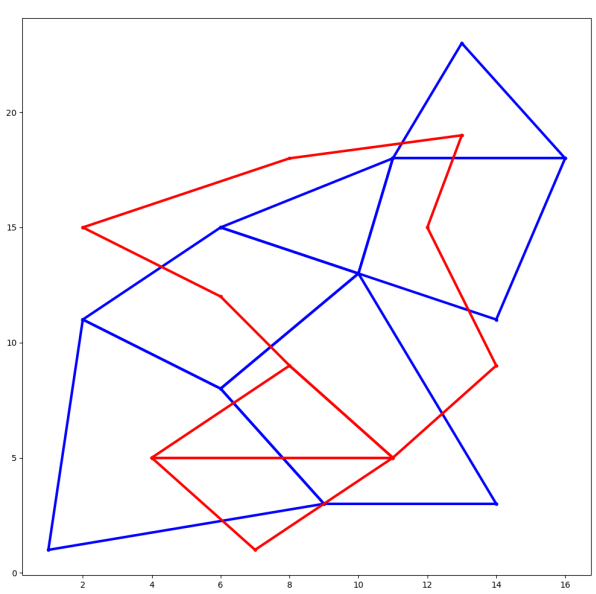
- **Split edges** at the intersection points. This is particularly useful when an edge is intersected multiple times by the other subdivision.

  For example, the edge e is intersected in two points. At first the edge is split at the first intersection point and divided in e'_1 and e''_1; e''_1 then has to be split again at the second intersection point to obtain e'_2 and e''_2. Instead, if we already know that e is intersected in two points, we can directly construct only e'_1, e'_2 and e''_2.

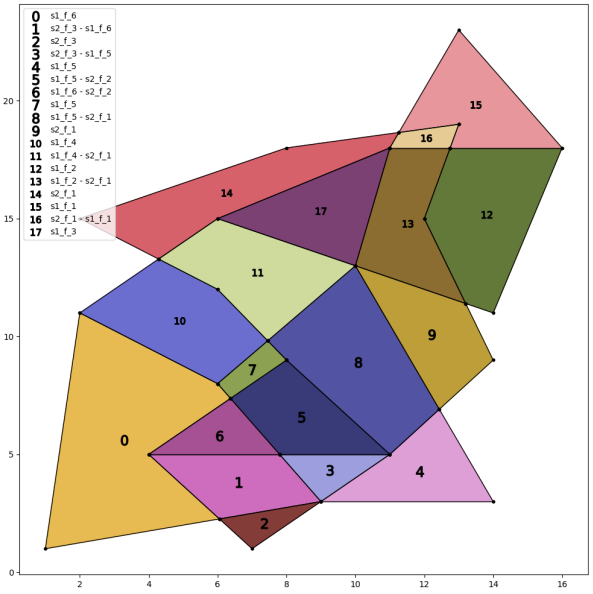


- **Determine the faces** from the original subdivisions that make up a face in the overlay: for each split edge that is now part of a specific face in the overlay, we know the original edge it was a part of and therefore the face it belonged to.

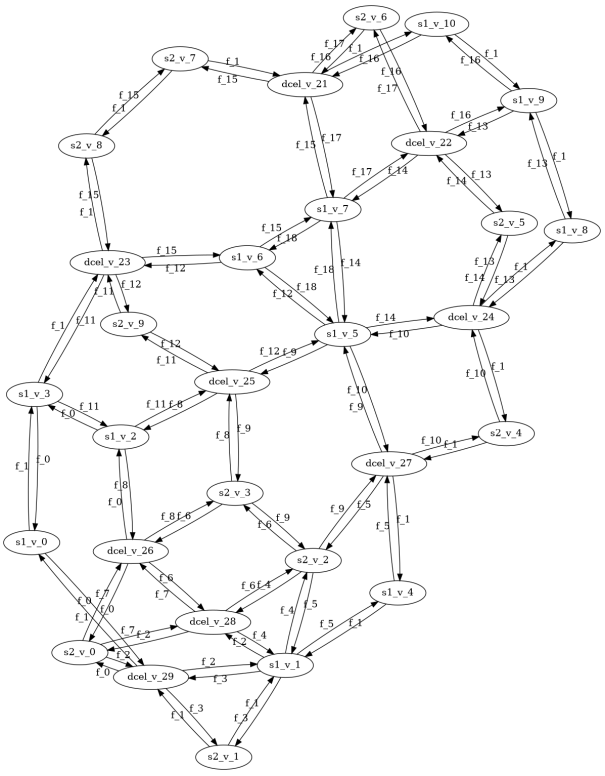
Below we can see an example of the overlay algorithm applied to two simple subdivisions



(a) Original subdivisions



(b) Overlay of the two subdivisions



(c) Different representation of the overlay as a graph

The overlay computation doesn't increase the complexity with respect to the line segment intersection algorithm but, given the suboptimal complexity of our intersection algorithm implementation, the overall complexity is $O(kn \log n)$ where $n = n_1 + n_2$ is the total number of segments in the two subdivisions and k is the number of intersections instead of the theoretical $O(n \log n + k \log n)$.

4 Conclusions

4.1 Possible Improvements

The implementation of the overlay algorithm is functional and produces the expected results. However, there are several improvements that could be made to the implementation to make it more efficient and robust. Some of the possible improvements are:

- **Status data structure**: the current implementation uses a list to store the status. This is inefficient because the list has to be sorted at each iteration. Initially, the implementation used a Red-Black Tree for the status but updating the position of the segments intersections line basically meant reconstructing the entire tree at each iteration. A different data structure that allows to update the value of a node (i.e. the intersection x coordinate of a segment with the sweep line) without removing and reinserting the node might work better.
- **Holes in faces**: the current implementation of DCELs doesn't support holes in faces which would make it more general and versatile. This wasn't done due to time constraints.
- **More tests for overlay**: while I was able to find a variety of tests to check the correctness of the intersection algorithm, the overlay algorithm could benefit from more tests. Moreover, the tests only check the correctness of the algorithm by comparing the number of faces in the output DCEL with the expected number but this doesn't necessarily cover all the possible errors that could occur.