Doubly-Connected Edge List

A Minimum Viable Product

Margarida Vieira
Nuno Costa
Tiago Rodrigues
Faculty of Engineering of the University of Porto
Porto, Portugal

Abstract

The doubly-connected edge list (DCEL) is a valuable data structure for solving computational geometry problems. This report delves into the exploration of the DCEL and provides insights into the implementation process to address the Map Overlay Problem. The report highlights the encountered details and challenges, including difficulties in showcasing its functionality. However, a small working example is presented to illustrate its application. Furthermore, the results of the implementation are discussed, and potential future enhancements to the project are considered.

Keywords: doubly-connected edge list, computational geometry, geometry, map overlay problem

1 Introduction

What should be included in a subdivision representation? There are several operations that can be highly valuable, such as identifying the face that contains a specific point, traversing the boundary of a given face, accessing an adjacent face through a shared edge, or examining all edges surrounding a vertex. These operations prove to be essential in many Computational Geometry tasks. The representation that can support all these operations is known as a doubly-connected edge list (DCEL), which will be the main focus of this report.

Initially, we will explore the structure itself and its definition. Additionally, we will provide an overview of some common applications of the DCEL. However, our primary emphasis will be on one specific application called the Map Overlay Problem. This problem serves as the motivation behind our project. We will describe our implementation of the DCEL and how it enabled us to efficiently solve the Map Overlay Problem. Furthermore, we will delve into the challenges posed by the algorithm and discuss the additional data structures and algorithms that played a role in our solution. Finally, we will present the results we achieved with this algorithm and provide guidelines for potential future enhancements to the project.

2 The Doubly-connected Edge List

This section provides an overview of the doubly-connected edge list (DCEL) data structure and its distinguishing features compared to other graph representations. Additionally, it discusses the relevance of the DCEL in various domains by highlighting common use cases. Furthermore, it outlines the specific problem that this report aims to address and provides the necessary contextual background.

2.1 Structure Overview

The doubly-connected edge list consists of three collections of records: one for the vertices, one for the faces, and one for the half-edges. These records store the following information:

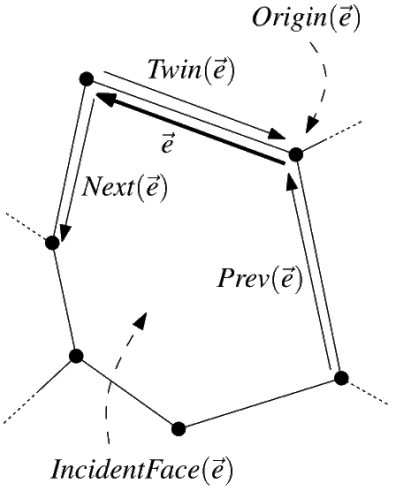

Figure 1. Information of a DCEL's half-edge.

- The vertex record of a vertex v stores the coordinates of v in a field called $Coordinates(v)$. It also stores a pointer $IncidentEdge(v)$ to an arbitrary half-edge with v as its origin.
- The face record of a face f stores a pointer *OuterComponent(f)* to some half-edge on its outer boundary. For the unbounded face, this pointer is *nil*. It also stores a list $InnerComponents(f)$, which contains a pointer to some half-edge on the boundary of the hole for each hole in the face.
- The half-edge record of a half-edge $\vec{e}$ stores a pointer $Origin(\vec{e})$ to its origin, a pointer $Twin(\vec{e})$ to its twin half-edge, and a pointer $IncidentFace(\vec{e})$ to the face that it bounds, as represented in Figure 1. There is no need to store the destination of an edge because

it is equal to $Origin(Twin(\vec{e}))$. The origin can be chosen such that $IncidentFace(\vec{e})$ lies either to the right or to the left of $\vec{e}$ when it is traversed from origin to destination. However, by convention, the latter is usually chosen. The half-edge record also stores pointers $Next(\vec{e})$ and $Prev(\vec{e})$ to the next and previous edge on the boundary of $IncidentFace(\vec{e})$. Thus $Next(\vec{e})$ is the unique half-edge on the boundary of $IncidentFace(\vec{e})$ that has the destination of $\vec{e}$ as its origin, and $Prev(\vec{e})$ is the unique half-edge on the boundary of $IncidentFace(\vec{e})$ that has $Origin(\vec{e})$ as its destination.

2.2 Motivation and Applications

The Doubly-Connected Edge List (DCEL) is a foundational and versatile data structure widely employed in diverse fields. It offers numerous benefits, including efficient representation, topological analysis, spatial queries, and navigation capabilities.

In the realm of Computational Geometry, the DCEL plays a vital role by enabling efficient representation and manipulation of geometric structures. It provides the necessary operations to address various geometric problems effectively. Moreover, in the realm of Computer Graphics, the DCEL's ability to navigate boundaries, access adjacent elements, and perform topological analysis makes it indispensable. Additionally, the DCEL finds extensive use in Geographical Information Systems (GIS), providing efficient point location, spatial analysis, map overlay operations, and topological consistency checking. Furthermore, the DCEL finds applications in fields such as finite element analysis and computational biology, among others.

2.3 Problem description

This project focuses on exploring the applicability of the Doubly-Connected Edge List (DCEL) data structure in the context of Geographic Information Systems (GIS). The primary objective is to develop a GeoJSON Reader capable of performing common set operations, such as intersection and union, on different regions represented within the GeoJSON format.

To accomplish this objective, the project initially addresses the Map Overlay Problem, which involves integrating multiple layers of spatial data. In this context, the Map Overlay Problem entails combining distinct maps or geospatial datasets, each with unique geographic features or attributes, to create a composite map that incorporates the shared spatial extent of the original maps. This overlay enables the identification of spatial correlations, patterns, and facilitates the desired set operations while preserving the individual data from each original map.

The methodology involves implementing the DCEL data structure to efficiently handle and manipulate the geographic data encoded in the GeoJSON format. By leveraging the DCEL's capabilities, the project aims to enable the execution of set operations, such as intersection and union, on the regions within the composite map.

Through this approach, the project seeks to provide a practical solution for performing set operations on spatial data within a GIS context, demonstrating the effectiveness and usefulness of the DCEL data structure in this domain.

3 Implementation

This section outlines the steps taken to achieve the project's objectives. The chosen technologies are briefly discussed, along with their rationale. The implementation of the Line Sweep Algorithm, a key building block of the Map Overlay Problem, is highlighted, emphasizing the importance of the utilized data structures. The challenges associated with solving the Map Overlay Problem are also addressed, showcasing the intricacies of developing the algorithm.

The implementation can be found in this Github Repository.

3.1 Technological stack

The following technologies were chosen for the implementation of the solution:

- **GeoJSON for data storage**, since it provides a standardized and efficient way to represent geographic features and their associated attributes, whilst allowing us to retain the compatibility and integrity of our data;
- **C++ for the implementation of the data structures and algorithms**, due to its performance-oriented nature;
- **Matplot++ for visualization**, which is a C++ graphics library that provides a convenient and intuitive interface for generating high-quality visualizations.

3.2 Line Sweep

In order to lay the groundwork for addressing the Map Overlay problem, the Line Sweep Algorithm is introduced.

The Line Sweep Algorithm is specifically designed to determine the intersection points among multiple line segments. In this context, a line segment is defined as a section of a straight line bounded by two endpoints. As a result, the representation of a line segment can be expressed solely through these endpoints.

By implementing the Line Sweep Algorithm, the objective is to efficiently calculate the intersection points between the given line segments. This algorithm provides a fundamental framework for subsequent advancements towards solving the more complex Map Overlay problem.

3.2.1 General Idea. The Line Sweep Algorithm adopts an event-driven approach, utilizing a virtual horizontal line, known as a "sweep line," that progressively sweeps across the

plane. At each significant position, referred to as an "event point," the algorithm performs specific operations.

To efficiently handle event points, the Line Sweep Algorithm relies on two key data structures: an event queue and a status structure. The event queue captures all event points and arranges them in the order they would intersect with the sweep line. This arrangement allows for systematic processing and analysis of each event point. On the other hand, the status structure optimizes the algorithm's performance by reducing the number of comparisons required and avoiding brute-force intersection calculations.

The status structure represents an ordered sequence of line segments intersecting the sweep line. By maintaining this ordered sequence, the algorithm can efficiently identify intersecting segments and perform the necessary computations.

The pseudocode for this algorithm that guided our implementation can be found in appendix 7. This pseudocode was obtained from [1].

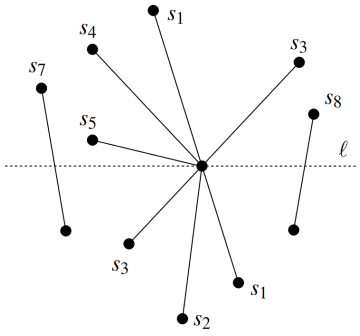

Figure 2. Representation of the sweep line on an event point

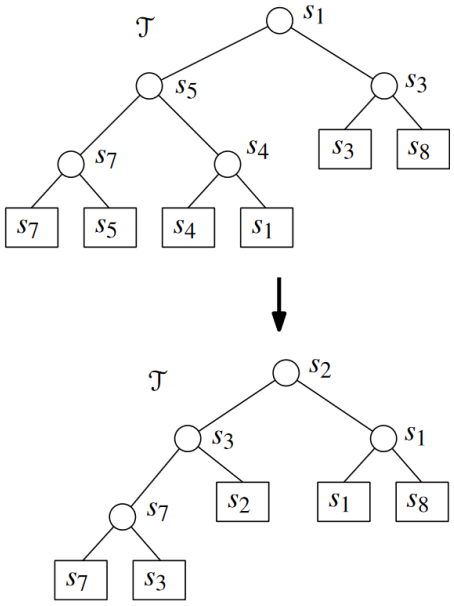

Figure 3. Corresponding changes in the status structure after crossing the event point.

3.2.2 Implementation Details. The implementation process begins by establishing the necessary data structures to support the Line Sweep Algorithm. Initially, a segment is defined as a pair of twin half-edges, as the underlying structure does not inherently provide a segment definition.

To facilitate the functioning of the algorithm, structures for the event queue and status structure are required. For the event queue, a priority queue is chosen as it aligns with the definition, sorting points based on their y-coordinate. In case of a tie, the leftmost point takes precedence. In C++, the `std::priority_queue` provided by the Standard Template Library (STL) is utilized, incorporating a custom comparator to order the events accordingly.

As for the status structure, segments that intersect the sweep line need to be inserted or removed dynamically. Thus, a dynamic structure is essential. Considering the well-defined ordering of segments within the status structure, a balanced binary search tree (BST) is deemed suitable. This choice, recommended in [1] and illustrated in Figure 3, offers optimal performance for the required operations. However, as the STL does not include a built-in implementation for a BST that allows mutable elements -`std::set` uses `const` extensively to maintain immutability -, a custom implementation is created, utilizing a `std::vector` as the foundational structure.

By defining these necessary data structures, the implementation lays a solid foundation for executing the Line Sweep Algorithm effectively.

To illustrate the concrete steps involved in the algorithm, a systematic approach is followed. The process commences with the initialization of the data structures - the status structure is initially empty, while the event queue is populated with all the endpoints. Notably, for each upper endpoint, a reference to the corresponding segment is stored. By ordering the endpoints based on the y-axis, the event queue effectively simulates the top-to-bottom traversal of a virtual sweep line.

With the data structures prepared, the algorithm proceeds to handle events by performing specific actions at each event point. The general approach for event handling is as follows: when encountering upper and lower endpoints, the corresponding segments are inserted and deleted from the status structure, respectively. Additionally, at intersection points, the order of the intersecting segments must be switched.

In both cases, intersection tests are crucial. When the event point represents an upper endpoint, signifying the start of a new segment intersecting the sweep line, the segment is added to the status structure. Subsequently, an examination

is conducted to determine whether it intersects any of the existing neighboring segments.

Likewise, when an intersection is detected, and the order of intersecting segments is switched, the left and right neighbors of each segment undergo a change. This modification is depicted in Figure 4, highlighting the consequential adjustments resulting from the intersection.

By diligently following these steps, the algorithm effectively handles events and intersections, progressively building the desired output.

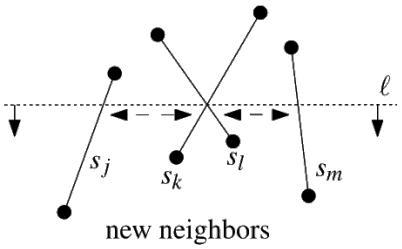

Figure 4. An event (intersection) point and the consequent change in the neighbors of the intersecting segments. Before the intersection point between s_k and s_l, s_k had s_l and s_m as its left and right neighbors, respectively. After the intersection, the order of the intersecting segment is switched and s_k now has s_j as its left neighbor and s_l as its right neighbor, so we need to test for intersections against the new neighbors. The same logic applies to s_l.

An additional crucial implementation aspect regarding the status structure pertains to the ordering of segments within it, which ensures that only adjacent segments in the horizontal ordering are tested. Consequently, when a new segment is introduced, it is solely evaluated against the two neighboring segments — the ones immediately positioned to the left and right. By adopting this approach, the algorithm circumvents the need to test a quadratic number of segment pairs, leading to significant efficiency gains. The correctness of this approach in computing all the intersections can be found in [1].

There were some special cases to take into account here. For example, when finding all the segments in the status tree that contained the endpoint, p, special attention needed to be taken to segments of infinite slope (vertical segments), and segments of slope 0 (horizontal segments). Both needed some extra conditions to be checked before considering them intersecting segments.

With this setup, we can be assured that each intersection will be properly computed, and the events will be correctly processed. The final time complexity is of $O(n \log n + l \log n)$, where l is the number of intersection points while taking $O(n)$ space complexity. The proof for this theorem is available in [1].

3.3 Map Overlay

Having established the foundational algorithm for computing intersections between edges, the scope can now be expanded to accommodate more intricate structures, specifically full plane subdivisions. With this expansion, the concept of faces is introduced, enriching the representation.

In this context, a face refers to a distinct region of the plane that possesses specific information. In the context of a map, for instance, a face not only encompasses geometric attributes, as outlined in Section 2.1, but can also encompass additional attribute information. These attributes might include population density, average precipitation, or the type of vegetation within the corresponding region.

By incorporating attribute information within each face, the representation becomes more comprehensive, allowing for a more nuanced analysis of the underlying data. This extension enhances the applicability of the data structure within diverse domains, including geographic information systems (GIS) and spatial analysis.

3.3.1 General Idea. Building upon the Line Sweep Algorithm, this algorithm expands its functionality to address the calculation of map overlays. Central to this expansion is the identification of intersections and the computation of new half-edges. However, the significance of this algorithm lies in its ability to determine the enclosed face for each half-edge, establish the outer boundary, and define the inner boundaries, referred to as holes, within each region. To accomplish this, auxiliary algorithms, including Tarjan's strongly connected component algorithm, are utilized as integral steps in the process.

The implementation of this algorithm adheres to a pseudocode, which guided our development efforts. The pseudocode, obtained from [1], is available in Appendix 8. It provides a comprehensive outline for executing the algorithm, ensuring clarity and coherence throughout the implementation process.

3.3.2 Implementation Details. Having initialized the necessary fields pertaining to vertex and half-edge records, the subsequent objective was to compute the pertinent information regarding the resulting faces in the map overlay. Each face, except for the unbounded face, should possess a distinct outer boundary. Therefore, the number of face records to be generated should correspond to the number of outer boundaries, with an additional record for the unbounded face.

However, a significant challenge emerged in distinguishing between cycles that represented outer boundaries and those that denoted inner boundaries (holes) within specific faces. Additionally, it was essential to identify which boundary cycles enclosed the same face, indicating when a cycle corresponded to the boundary of a hole within a particular face.

To address these challenges, we followed the approach outlined in the literature [1]. We constructed a graph, denoted as G, which served as an auxiliary data structure. The construction of this graph required developing a custom implementation. Each boundary cycle, whether outer or inner, was represented as a node in the graph. Additionally, an extra node was designated to represent the imaginary outer boundary of the unbounded face.

To establish connections between boundary cycles that enclosed the same face, specific conditions were considered. An edge was created between two cycles if and only if one cycle represented the boundary of a hole, and the other cycle had a half-edge immediately to the left of the leftmost vertex of the corresponding hole cycle. Alternatively, if this condition was not met, the node representing the cycle would be linked to the node representing the unbounded face. Further clarity on this connection condition can be found in Figure 5, while an example of the resulting graph for a given subdivision is depicted in Figure 7.

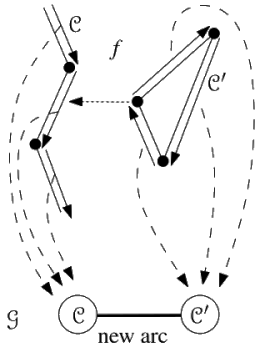

Figure 5. Connection between two cycle nodes of G.

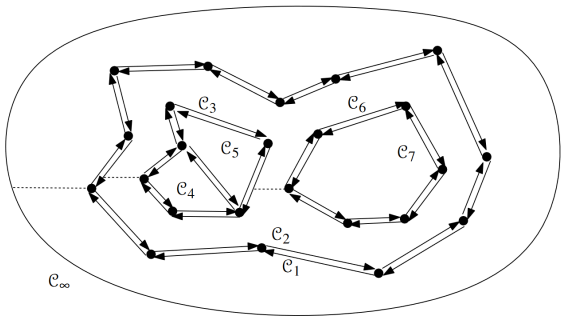

Figure 6. Representation of a subdivision C

To successfully implement the aforementioned process, two key tasks had to be accomplished. Firstly, it was necessary to determine whether a cycle represented an outer boundary or not, and secondly, the leftmost vertex of a specific boundary cycle had to be identified. The orientation of the half-edges comprising the cycle (clockwise or counterclockwise) was utilized to ascertain whether a cycle represented an outer boundary. On the other hand, the leftmost vertex of a boundary cycle could be easily determined by examining the coordinates of the vertices belonging to the cycle.

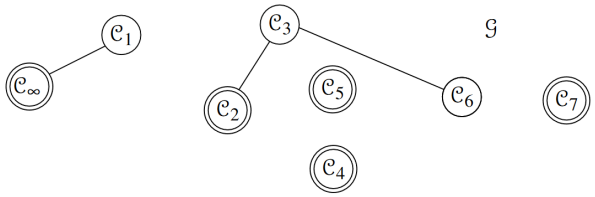

Figure 7. The graph G for the subdivision C

Having established the *OuterComponent* and *InnerComponents* for each face record, as well as the *IncidentFace* for each half-edge, one final task remained. Each face in the overlay needed to be labeled with the attributes of the faces in the original subdivisions that contained it. The solution to compute this information was left as an exercise in the referenced literature [1], serving to test the understanding of the overall approach.

Our proposed solution involved a series of key steps. Firstly, when computing the new half-edges resulting from intersections, we assigned their incident face based on the incident face of the corresponding original half-edges. Subsequently, for each identified boundary cycle, we created a new face, denoted as f, and combined the attributes of the incident faces of the edges in that cycle. This merging process ensured that if two faces were merged, their attributes would be preserved in at least one of the edges of the new boundary cycle. The resulting face incorporated the previously determined boundary cycle, the identified inner components, and the combined attributes from the original individual faces within the cycle. Finally, this face was added to the final Doubly-Connected Edge List (DCEL), guaranteeing the inclusion of the relevant embedded information.

By following this approach, we successfully computed the necessary face information, allowing for a comprehensive representation of the overlay result in the DCEL.

With the completion of the aforementioned steps and the resolution of various underlying challenges, the computation of the map overlay, integrating the two maps, was successfully achieved.

4 Results

Having completed the implementation of the algorithm, we proceeded to conduct thorough testing to evaluate its effectiveness and capabilities. While the applications described

in [1] are diverse, our focus was specifically on the GIS domain, as mentioned in Section 2.3. In this section, we will discuss our attempt to apply the algorithm to real-world geographical data, as well as the challenges we encountered during the process. By using a simplified example, we were able to gain deeper insights into the algorithm's functionality, which we will outline in the following subsections.

4.1 Attempted use of the United States Dataset

For our real-world data experiment, we identified a comprehensive collection of GIS datasets known as Geo-Maps, which encompasses information about various natural features such as borders, rivers, lakes, and coastlines on a global scale. The Geo-Maps dataset is readily accessible in the corresponding Github Repository. To provide a tangible example, we have included a sample dataset in Figure 8. This dataset demonstrates the type of geographic information we utilized in our analysis.

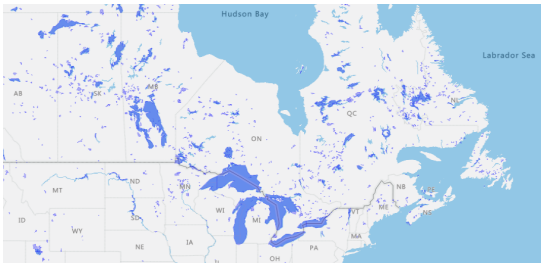

Figure 8. Visualization of sample data present in a GeoJSON dataset pertaining to the lakes of the world, with a focus on the Northeastern U.S.A and East Canada. This visualization was obtained via Github's built-in visualizer. Each lake is highlighted in dark blue.

We specifically focused on a dataset that encompassed the border of the United States, along with various regional and cultural subdivisions within its boundaries, such as the Midwest and the South. Our objective was to compute the overlay between these subdivisions and examine whether the original map information was preserved in their union. However, we encountered a significant challenge in handling the massive amount of data and parsing it efficiently to generate the initial DCEL.

Loading each region alone took approximately 15 to 20 minutes, and when combined with the entire U.S. border, it became a cumbersome process, especially during the testing stage. The extensive loading times rendered this approach impractical, and as a result, we had to abandon the idea of plotting and verifying the correctness of such a large-scale example.

4.2 Visualizing the result

To simplify the case while still utilizing real-world data, we decided to focus on a localized example in the vicinity of the Faculty of Engineering of the University of Porto (FEUP) and the nearby Hospital de São João. Our study specifically targeted three regions: the Faculty of Engineering, the Hospital grounds, and the surrounding area of the Faculty of Sports (FADEUP), which is in close proximity to FEUP. To achieve this, we created two custom GeoJSON files. One file represented the area encompassing the hospital and FEUP, while the other file depicted the two faculties without the hospital. These datasets were constructed using the online tool geojson.io and can be observed in Figures 9 and 10. The properties included in each dataset primarily consisted of the region's name and were mainly utilized to test the correct storage of attributes during the algorithm's execution.

Figure 9. Visualization of the area covering FEUP and the hospital, seen in geojson.io.

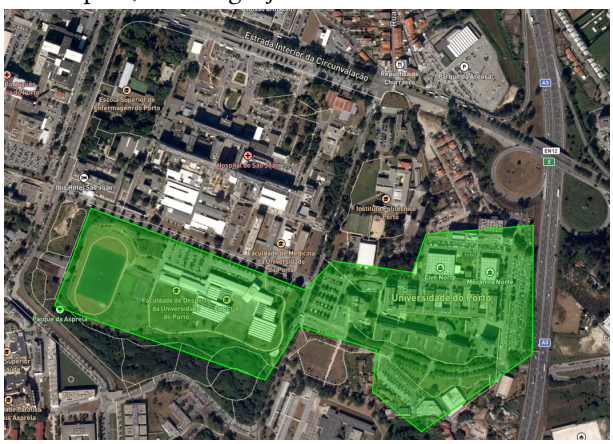

Figure 10. Visualization of the area covering FADEUP and FEUP, seen in geojson.io.

The presence of slight inaccuracies in the clipping process provides an interesting aspect to our test, as it allows us to examine the algorithm's ability to identify and assign the same ID to regions that have already been traversed and contain identical information.

To visualize the results of the overlay, we utilized Matplot++, a simple visualization library. By feeding the data to Matplot++, it plotted polygons based on the provided coordinates. To ensure the accuracy of our results, we iterated through the resulting Doubly-Connected Edge List (DCEL) face by face. Each face was plotted as a unique polygon by following its bounding half-edges. Additionally, we checked the attribute information of each face to determine if a face with the same attributes had already been created. If so, these faces were assigned the same color. On the other hand, if a face had unique attributes, a new random color was assigned. To further confirm the correct passing of attribute information, we stored the attributes as captions for each face. The visualization results can be observed in Figure 11.

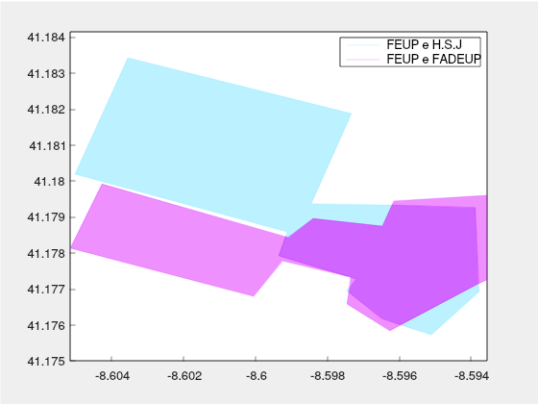

Figure 11. Visualization of the Overlay of both DCELs in Matplot++.

The generated image provides valuable insights into the accuracy and effectiveness of our implementation. Several important observations can be made from the visualization. Firstly, the traversal of each face's boundary accurately produces the corresponding polygons, even for the intersecting regions. This demonstrates the correctness of the boundary computation. Secondly, the varying colors at the intersection points indicate the successful calculation of the overlay, effectively capturing the shared and distinct areas between the polygons. Lastly, the distinct colors assigned to each polygon highlight the correct transfer of attribute information, validating the attribute processing step.

These findings not only affirm the accuracy of our implementation across its various components but also illustrate the potential applications and possibilities of such an overlay. By leveraging attribute information, this approach can be extended to various types of maps, enabling expressive and informative results.

However, it is important to acknowledge the limitations of our study. Due to the absence of alternative DCEL implementations, we were unable to perform a benchmark comparison against related work. Additionally, the impractical duration of parsing JSON files for larger scales, such as the entire United States, hindered the calculation of meaningful metrics for evaluating the resulting overlays.

5 Future Work and Conclusions

In summary, the process of constructing the data structure for this project, the doubly-connected edge list (DCEL), initially seemed straightforward. However, implementing the algorithms that utilized this structure proved to be more challenging than anticipated. One of the main hurdles we faced was the lack of detailed and comprehensive documentation, which hindered our understanding of critical aspects. Our access to relevant literature was limited, relying on a single book that sometimes provided vague explanations and incomplete coverage. Additionally, the scarcity of alternative implementations further complicated the development process.

The complexity of the algorithms, combined with their geometric nature and ambiguous textual descriptions, made it difficult to fully grasp certain key concepts. As a result, we encountered difficulties in achieving our initial goal of creating an interactive and user-friendly visualization tool, as outlined in our initial presentation. However, since these additional features were not the core focus of the project, we consider the project to be a success.

Despite the challenges faced, we successfully implemented the data structure and algorithms. The obtained results are promising and serve as a solid foundation for future work and further development. With more comprehensive documentation and additional resources, it is likely that the project could have yielded even more favorable outcomes.

6 Self/Peer Assessment

Based on the effort invested in this project and the previous projects, it is assessed that the workload was indeed divided evenly among group members. Each member's contribution is considered to be of equal importance and is valued at 1/3 of the overall project's contribution.

Regarding the expected individual grades, we have taken into consideration both the presentations and the continuous assessment. Based on these factors, the anticipated grades for each team member are as follows:

- Margarida Vieira - 18
- Nuno Costa - 19
- Tiago Rodrigues - 18

References

[1] Mark De Berg, Otfried Cheong, Marc Van Kreveld, and Mark Overmars. 2008. *Computational geometry: Algorithms and applications.* Springer Berlin Heidelberg. 1–386 pages. https://doi.org/10.1007/978-3-540-77974-2

7 Line Sweep

7.1 $FindIntersections(S)$

Input: A set S of line segments in the plane.

Output: The set of intersection points among the segments in S, with for each intersection point the segments that contain it.

1. Initialize an empty event queue Q.
2. Insert the segment endpoints into Q; when an upper endpoint is inserted, the corresponding segment should be stored with it.
3. Initialize an empty status structure T.
4. **while** Q is not empty
5. **do** Determine the next event point p in Q and delete it.
6. $HandleEventPoint(p)$

7.2 $HandleEventPoint(p)$

1. Let $U(p)$ be the set of segments whose upper endpoint is p; these segments are stored with the event point p. (For horizontal segments, the upper endpoint is, by definition, the left endpoint.)
2. Find all segments stored in T that contain p; they are adjacent in T. Let $L(p)$ denote the subset of segments found whose lower endpoint is p, and let $C(p)$ denote the subset of segments found that contain p in their interior.
3. **if** $L(p) \cup U(p) \cup C(p)$ contains more than one segment
4. **then** Report p as an intersection, together with $L(p)$, $U(p)$, and $C(p)$.
5. Delete the segments in $L(p) \cup C(p)$ from T.
6. Insert the segments in $U(p) \cup C(p)$ into T. The order of the segments in T should correspond to the order in which they are intersected by a sweep line just below p. If there is a horizontal segment, it comes last among all segments containing p.
7. (Deleting and re-inserting the segments of $C(p)$ reverses their order.)
8. **if** $U(p) \cup C(p) = \emptyset$
9. **then** Let s_l and s_r be the left and right neighbors of p in T.
10. $FindNewEvent(s_l, s_r, p)$
11. **else** Let s' be the leftmost segment of $U(p) \cup C(p)$ in T.
12. Let s_l be the left neighbor of s' in T.
13. $FindNewEvent(s_l, s', p)$
14. Let s'' be the rightmost segment of $U(p) \cup C(p)$ in T.
15. Let s_r be the right neighbor of s'' in T.
16. $FindNewEvent(s'', s_r, p)$

7.3 $FindNewEvent(s_l, s_r, p)$

1. **if** s_l and s_r intersect below the sweep line, or on it and to the right of the current event point p, and the intersection is not yet present as an event in Q
2. **then** Insert the intersection point as an event into Q.

8 Map Overlay

8.1 $MapOverlay(S_1, S_2)$

Input: Two planar subdivisions S_1 and S_2 stored in DCELs
Output: The overlay of S_1 and S_2 stored in a DCEL D

1. Copy the DCELs for S_1 and S_2 to a new DCEL D.
2. Compute all intersections between edges from S_1 and S_2 with the *LineSweep* algorithm. In addition to the actions on T and Q required at the event points, do the following:
   - Update D as explained above if the event involves edges of both S_1 and S_2.
   - Store the half-edge immediately to the left of the event point at the vertex in D representing it.
3. (Now D is the DCEL for $O(S_1, S_2)$, except that the information about the faces has not been computed yet.)
4. Determine the boundary cycles in $O(S_1, S_2)$ by traversing D.
5. Construct the graph G whose nodes correspond to boundary cycles and whose arcs connect each hole cycle to the cycle to the left of its leftmost vertex and compute its connected components. (The information to determine the arcs of G has been computed in line 2, second item.)
6. **for** each connected component in G
7. **do** Let C be the unique outer boundary cycle in the component and let f denote the face bounded by the cycle. Create a face record for f, set $OuterComponent(f)$ to some half-edge of C, and construct the list $InnerComponents(f)$ consisting of pointers to one half-edge in each hole cycle in the component. Let the $IncidentFace()$ pointers of all half-edges in the cycles point to the face record of f.
8. Label each face of $O(S_1, S_2)$ with the names of the faces of S_1 and S_2 containing it, as explained above.