Project Report
Advanced Topics in Algorithms

João Malheiro
202004449

Sofia Avelino
202008485

June 15, 2024

Contents

1 Introduction

In modern wireless networking, ensuring optimal signal coverage within complex environments such as buildings requires sophisticated modeling of signal propagation. Traditional considerations include both the physical distance from the wireless modem and the number of walls the signal must penetrate.

This project focuses on the development and implementation of an algorithm to determine the region within an orthogonal polygon P that is illuminated by a given k-modem. A k-modem, as defined, is capable of transmitting a stable signal through up to k walls along a straight line. The illuminated region is defined by the set of points within P that can be connected to the modem by a line segment crossing at most k walls.

2 Objectives

The main objectives of this project are:

- **Algorithm implementation:** Implement an algorithm that propagates visibility cones to identify the illuminated region within the polygon P, using the given constraints of the modem's signal penetration capability.
- **Utilization of DCEL Representation:** Implement the algorithm using the Doubly Connected Edge List (DCEL) structure to handle the partitioned representation of the polygon's interior and exterior.
- **Analysis on Illumination:** Assess the accuracy of the algorithm in various scenarios by considering both the distance and wall-penetration factors.

The algorithm will take into account the simple structure of the polygon P, provided as a sequence of vertices in a counterclockwise order, and will utilize horizontal/vertical (H/V) partitions of the interior region of P. The focus is on ensuring the accurate representation of the illuminated region without handling degenerate cases where visibility cones reduce to rays. This approach aims to provide a robust solution for understanding and optimizing wireless signal coverage within complex geometric environments.

The chosen programming language for the implementation of the algorithm was Java.

3 Algorithm Design

3.1 Data Structure Definition

The program's implementation relies on several key data structures, defined as follows:

3.1.1 Vertex

A *Vertex* is represented by:

- Two doubles indicating the coordinates of the vertex.

3.1.2 Edge

An *Edge* contains the following attributes:

- **Origin Vertex**: The starting vertex of the edge.
- **Pointer to the Next Edge**: A reference to the subsequent edge in the sequence.
- **Pointer to the Previous Edge**: A reference to the preceding edge in the sequence.
- **Twin Edge**: The edge that runs in the opposite direction.
- **Incident Face**: The face to which this edge belongs.

3.1.3 Face

A *Face* is characterized by:

- **Centroid**: The coordinates of the center of the face.
- **Counter/ID**: A unique integer assigned to each face of the polygon (with the exterior face represented by 0).
- **Modem**: The intensity required for the face to be visible by the k-modem.
- **Outer Component**: An edge initially assigned to the face.

3.1.4 Doubly Connected Edge List (DCEL)

The *DCEL* structure includes:

- **List of Vertices**: All vertices within the polygon.
- **List of Edges**: All edges that make up the polygon.
- **List of External Edges**: Edges that define the boundary of the exterior rectangle.
- **List of Faces**: All faces, including the exterior face.

These data structures form the foundation of the algorithm, enabling efficient representation and manipulation of the polygon's geometric and topological properties.

3.2 Proposed Algorithm

The algorithm we implemented for visibility computation from a given guard vertex proceeds as follows:

1. **Ray Casting from the Guard Vertex:**
   - For each vertex in the polygon:
     - If the vertex is not the guard, compute the ray originating from the guard and passing through the current vertex.
     - Identify intersections of this ray with the polygon's external edges.
     - Consider only intersection points that lie beyond the current vertex along the ray.
     - Collect these intersection points.
2. **Creating New Edges:**
   - Connect sequential intersection points, ensuring that the resulting edges lie within the polygon.
   - Add these new edges to the DCEL (Doubly Connected Edge List) to form new partitions of the polygon.
3. **Visibility Calculation for Each Face:**
   - For each face of the updated polygon:
     - Compute its centroid.
     - Trace the segment between the guard vertex and the centroid and count the number of intersection points with the polygon's external edges.
     - Determine the face's visibility based on the number of intersection points, representing the traversal of light through a wall.
4. **Identifying Regions with Desired Maximum Visibility:**
   - Merge the faces with visibility less than or equal to the intended value.
   - Return the vertices delimiting the resulting region(s) in counterclockwise order.

This iterative process updates the DCEL (Doubly Connected Edge List), enabling determination of visible regions within the polygon relative to a specified modem originating from the guard vertex.

4 Implementation

The implementation of the proposed algorithm is organized into four main sections:

- DCEL Creation
- Polygon Partitioning
- Visibility Computation
- Face Merging

Each of these sections was crucial in achieving the desired results and will be succinctly described in this section.

4.1 DCEL Creation

The initial step in our algorithm implementation involved constructing a DCEL (Doubly Connected Edge List) from a list of vertices arranged in counter-clockwise order. This process encompassed creating edges and faces, and establishing connections among these components.

Below is a pseudocode snippet depicting the function designed for this purpose:

```
function createDCELFromPolygon(Vertex[] vertexList):
    Create and add exterior face f0 to the DCEL
    Create and add interior face f1 to the DCEL

    for each successive vertices v1 and v2 in vertexList:
        Create new HalfEdge edge with origin v1
        Create new HalfEdge edge with origin v2
        edge.twin <- twinEdge
        twinEdge.twin <- edge
        edge.incidentFace <- f1
        twinEdge.incidentFace <- f0
        Add v1 to the DCEL
        Add edge and twinEdge to DCEL

    for each successive HalfEdges e1, e2, e3:
        e2.prev <- e1
        e1.next <- e2
        e2.next <-e3
        e3.prev <- e2
```

This pseudocode outlines the process of initializing a DCEL by sequentially linking vertices with HalfEdges, setting up twin relationships, and establishing connections between consecutive HalfEdges to form faces. This foundational step lays the groundwork for subsequent operations in our algorithm.

4.2 Polygon Partitioning

After constructing the DCEL, our next task was to implement a function for adding new partitions. This function facilitated the incorporation of H/V partitions and the addition of edges resulting from intersecting rays originating from the guard vertex towards every other vertex with the polygon's external edges.

This section of our algorithm was notably intricate, involving numerous pointer adjustments and careful handling of special cases.

```
function addPartition(Vertex origin, Vertex end):
    add origin to DCEL vertices if not already in DCEL

    create list I of HalfEdges
    for each HalfEdge e:
        if e intersects the partition segment:
            add e to I
```

```
HalfEdge h <- null
max <- 0
for each HalfEdge e1 in I:
count <- 0
    f = e1.incidentFace
    for each HalfEdge e2 incident to f:
        if e2 intersects the partition segment:
            count <- count + 1
    if count > max:
        h <- e1
        max <- count
```

The initial segment of the *addPartition()* function determines the optimal starting edge that includes the origin vertex for the algorithm. It identifies all edges intersecting the new partition segment at the origin and evaluates each edge's incident face to count intersections with the partition edge. The edge incident to the face with the highest intersection count is chosen, ensuring the algorithm starts within a face guaranteed to contain part of the new partition segment.

This section is particularly significant when the origin vertex aligns with an existing vertex of the polygon. In such cases, multiple edges intersect with the origin vertex, necessitating the selection of the most suitable edge to initiate the algorithm.

Next, our task was to traverse the polygon and bisect every edge and face intersecting with the new partition segment. This step is intricate and involves handling special cases when intersections occur at existing vertices. Below, we provide a generalized pseudocode outline of this section, omitting specific considerations for these special cases, which were carefully addressed in our implementation.

```
# we begin with knowledge of the initial HalfEdge h
HalfEdge prev <- h.prev
HalfEdge next <- h
h <- h.next
Face f <- h.incidentFace

Vertex i <- origin

while i != end:
    i <- intersection(h, p) # p is the new partition segment
    if i is not null:
        h.incidentFace = f
        Create new HalfEdge newH with origin i
        Create new HalfEdge newH_twin with origin next.origin
        Create new HalfEdge nexth with origin i
        Create new HalfEdge h_twin with origin i

        nexth.incidentFace <- prev.incidentFace
        nexth.incidentFace.outerComponent <- nexth
        nexth.next <- h.next
        h.next.prev <- nexth

        h_twin.next <- h.twin.next;
        h.twin.next.prev <- h_twin;
        h.twin.next <- null; # to be set later
        h_twin.prev <- null; # to be set later
        nexth.twin <- h.twin;
        h.twin.twin <- nexth;
        h.twin <- h_twin;
        h_twin.twin <- h;

        newh.incidentFace <- f
        f.outerComponent <- newh
        h.next <- newh
        newh.prev <- h
```

```
            newh.next <- next
            next.prev <- newh

            newh_twin.incidentFace <- prev.incidentFace
            newh_twin.incidentFace.outerComponent <- newh_twin
            newh_twin.prev <- prev
            prev.next <- newh_twin
            newh_twin.next <- nexth
            nexth.prev <- newh_twin
            newh_twin.twin <- newh
            newh.twin <- newh_twin

            if i != end:
                Create and add new face f with outer component h_twin to the DCEL
            else:
                f <- nexth.twin.incidentFace
                nexth.twin.next <- h_twin
                h_twin.prev <- nexth.twin

            Add HalfEdges newh, newh_twin, nexth and h_twin to the DCEL

            prev <- nexth.twin
            h <- h_twin
            next <- h

        h.incidentFace <- f
        h <- h.next
```

With this function, we can efficiently add any partition to a given DCEL by selecting origin and endpoint vertices that reside on its external edges. This capability enables dynamic modification of the DCEL structure to accommodate new partitions within the polygon geometry.

4.3 Visibility Computation

Having integrated the capability to add partitions to the DCEL, we proceeded to the crucial phase where our algorithm could be systematically constructed. Leveraging this functionality, we computed new partitions induced by rays originating from the guard vertex towards each vertex and seamlessly integrated them into the DCEL structure, as illustrated in the pseudocode below.

```
# H/V partitions are assumed to be added before this function is called

function computeVisibility(Vertex guard):
    create new list of Vertices I

    for each original vertex v of the polygon:
        if v is not the guard:
            r <- ray(guard, v)
            for each external HalfEdge e of the DCEL:
                i = intersection(e,r)

                if i is not null and lies beyond v in the ray:
                    add intersection point to I

            add v to I
            sort I in descending order by distace to guard

            for successive vertices v1, v2 in I:
                if the line from v1 to v2 is inside the polygon and edge doesnt aleady exist:
                    addPartition(v1,v2)

    create new list of vertices I2
    for each face f in the DCEL:
        if f is not the external face:
```

```
        c <- f.centroid
        r <- ray(guard, c)
        for each external HalfEdge e of the DCEL:
            i = intersect(e,r)
            if i != null and I2 doesnt contain i:
                Add i to I2
        f.modem <- intersections.size()
        if f.modem is odd: #can happen because of intersection at vertices
            f.modem <- f.modem + 1
    else:
        f.modem = 100000
```

With the implementation of this function we were able to obtain the visibility of each face of the DCEL.

4.4 Face Merging

The last step in our algorithm, once we'd obtained the visibility of every face of the polygon, was to merge the neighboring faces with visibility less than or equal to the stipulated maximum. The implementation of this part was similar to that of *addPartition()*, but in reverse.

This implementation picks one face with the intended modem that neighbors at least one other face that won't be in the visibility region. This ensures we can select the edge between them as the initial edge to return to at the end, as it is guaranteed not to be deleted. The algorithm then iterates through the face's edges until it finds one that neighbors another face with the intended visibility. Once it finds that edge, it updates the pointers of the previous and next edges to point to the respective edges of the neighboring face and deletes the intermediate edge, successfully merging the faces. This is then repeated until the initial edge is reached again.

This method is repeated until there is no face with the intended modem left to merge, as is illustrated in the pseudocode below:

```
funtion mergeFaces(int k, Vertex guard):
    create list of faces fList
    for each face f in the DCEL:
        if f.modem <= k:
            add f ti fList
    if fList is not empty:
        order faces in fList by y component in descending order, and by x component in
            descending order as a tie breaker
    while fList is not empty:
        Face f <- fList.get(0)
        remove f from fList
        HalfEdge start <- null
        for HalfEdge e incident to f:
            if e.twin.incidentFace.modem > k:
                start = e
                break

        e = start.next
        while e != start:
            if e.twin.incidentFace.modem <= k:
                remove e.twin.incidentFace from fList
                e.prev.next <- e.twin.next
                e.twin.next.prev <- e.prev
                e.next.prev <- e.twin.prev
                e.twin.prev.next <- e.next

                e.twin.next.incidentFace <- f
                e.twin.prev.incidentFace <- f

                HaldEdge rem <- e
                 e <- e.twin.next
                remove rem and rem.twin from the DCEL

            else :
                e <- e.next

            e.incidentFace <- f
            remove e.incidentFace from fList
```

After merging all the faces with the intended visibility into new regions, we then obtain the vertices of these regions in counterclockwise order. For each region, we start at the same initial edge and iterate through to the next edge repeatedly until we reach the initial edge again. During this process, we print or collect the values of the vertices where each edge originates. This final step ensures that the vertices of the merged regions are correctly ordered and easily accessible.

5 Results

In order to test our algorithm, we applied it to a large number of different polygons, guard vertices, and maximum visibility values. After intensive testing and obtaining consistently accurate results, we concluded that it worked as intended.

In addition to printing the bounding vertices of each visible region, we also implemented a simple script in Python that creates an image from the results obtained by the algorithm. This allowed for visualization of the results in a more direct manner. This was a very helpful tool in testing the algorithm as it allowed us to analyze our results in a faster and more obvious way.

Here are some examples of results obtained for different polygons and visibility values:

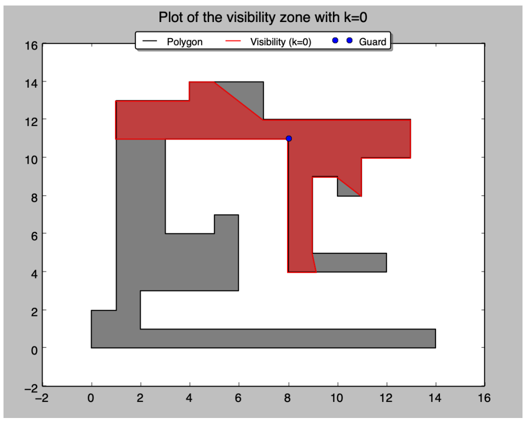

Figure 1: Visibility region for k=0

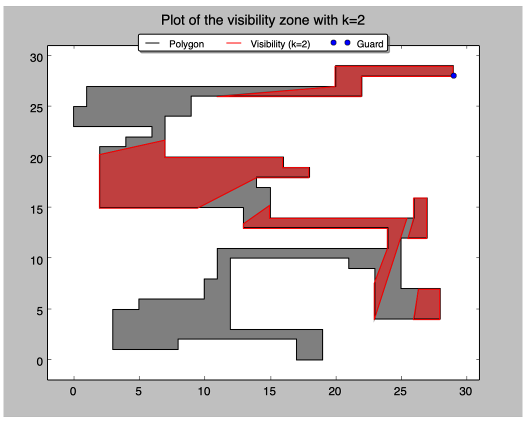

Figure 2: Visibility region for k¡=2

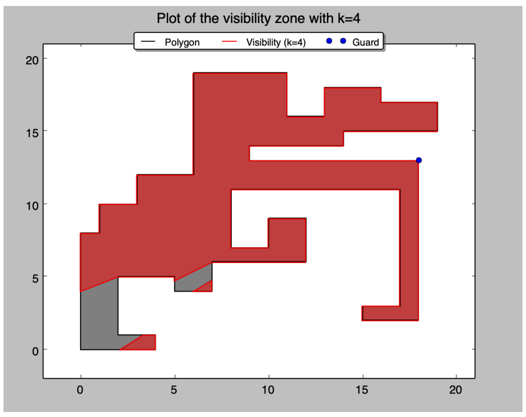

Figure 3: Caption

In addition to generating visualizations and confirming accuracy through extensive testing across diverse polygon scenarios and visibility parameters, it is noteworthy that the implementation of our algorithm presented significant challenges. Addressing various unforeseen special cases necessitated iterative adjustments to the algorithm. Despite these complexities, our team successfully overcame these challenges, achieving our objectives with robust outcomes.

6 Conclusion

Overall, the results obtained from the implementation of the proposed algorithm were highly successful in light of the objectives initially presented:

- **Algorithm Implementation:** We successfully implemented an algorithm capable of determining visible regions based on a modem's signal penetration capability.
- **Utilization of DCEL Representation:** The algorithm was effectively implemented using a Doubly Connected Edge List (DCEL) to represent the polygon and its partitions, leveraging this structure to enhance the algorithm's efficiency.
- **Analysis on Illumination:** Extensive testing demonstrated the algorithm's accuracy and its ability to achieve the desired results in visibility computation.

Furthermore, we exceeded our objectives by developing a visualization to enhance the presentation of our algorithm's outcomes, making them more interactive and user-friendly.

These findings validate that our approach not only fulfills the requirements but also offers a dependable and effective solution for illuminating orthogonal polygons.