Doubly Connected Edge List

Crossed with a KD Tree to allow for orthogonal range search

Daniel Latimer

Advisor: Dr. Bradford Nickerson

Table of Contents

Doubly Connected Edge List (DCEL)

The Doubly Connected Edge List is a winged edge data structure, meant for representing planar graphs. It provides queries to obtain the edges around a face, and edges incident on a vertex in O(m) time where m is the number of edges reported. It also provides the ability to obtain the faces on either side of an edge in O(1) time. Because it can obtain this information so quickly it is an ideal data structure for making local edits [1].

The data structure is comprised of a series of edges. Each edge is made up of six references, see Figure 1. Two of the references lead to sets of coordinates that define the start and end points of the edge. Another two of the references lead to the faces on either side of the edge. While the last two are references to other edges, the next counter clockwise edge on the Start vertex, and the next counter clockwise edge on the End vertex.

Edge	
Start:VertexPtr	End:VertexPtr
Face1:Integer	Face2:Integer
StartNextCCWEdge:EdgePtr	EndNextCCWEdge:EdgePtr

Figure 1

Construction

The data must be set up in a specific way. The preconditions to creating a DCEL are to have a valid set of lines that have a vertex at any point two lines cross; this would define a planar graph. The data must also be set up in a way to easily find all edges on a vertex.

Once we have the preconditions met, the next step would be to create a temporary data structure that is used in the construction of the DCEL. This edgeCycle data structure holds information about the order in which edges are incident in a clockwise manner around each vertex. The structure is an array of two integers. The first of which, containing a reference to the ending vertex of the particular edge around the vertex in question. The second integer is an index into the edgeCycle array of where to go to get the next clockwise edge around that vertex. The edgeCycle data structure is used in conjunction with another array called edgeCycleVertexIndex, it is of length n where n is the number of verticies in the map. This array gives you the index into edgeCycle for the first edge of a given vertex E.g.

```
// The third vertex
int vertex = 3;

// The index of the first edge in vertex 3's edge cycle
int firstEdgeCycle = edgeCycleVertexIndex[i];

// The first edge's ending vertex (with the starting vertex being 3)
int endingVertex = edgeCycle[firstEdgeCycle].vertex;

// The index into edgeCycle to get the next clockwise edge around vertex 3
int nextEdgeCycleIndex = edgeCycle[firstEdgeCycle].next;
```

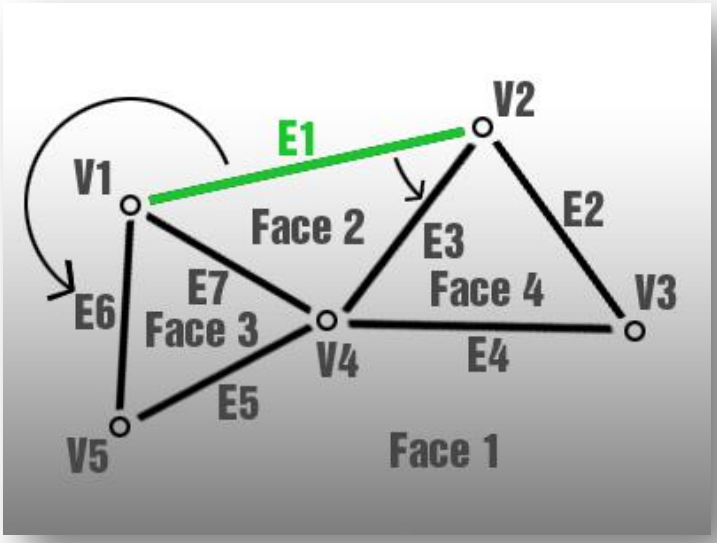

Figure 2

Once we have the edge cycles we can construct the edges of our DCEL. This is done by two functions, constructVertexCycles and constructFaceCycles. First we call constructVertexCycles. Its responsibility is to four variables in the Edge record: Start, End, StartNextCCWEdge and EndNextCCWEdge. It does this by iterating through the edgeCycle array, adding edges as it goes. It also uses the edgeCycle array to find which the next counter clockwise edge is on the start vertex and end vertex. The constructVertexCycles function also creates a list of indexes into the list of edges called m_firstOccuranceOfVertex where you can find the first occurrence of a vertex. After constructVertexCycles is complete we no longer need the edgeCycle and edgeCycleVertexIndex arrays.

In the example in Figure 2 we see edge 1 (E1) being processed. The edge data structure for edge 1 after running constructVertexCycles would look like this:

Edge { Start = 1, End = 2, Face1=NIL, Face2=NIL, StartNextCCWEdge=6, EndNextCCWEdge=3 }.

As you can see we still haven't initialized Face1 or Face2. It is the responsibility of the next function, constructFaceCycles, to do so. When we run the constructFaceCycles function we will iterate over the edges we have so far. Each time we find an edge that doesn't have face1 or face2 set we will assign a number for that face and use the next counter clockwise edge variables to traverse all edges on that face. For each edge encountered we will set its face number. The constructFaceCycles function also creates a list of indexes into the list of edges called m_firstOccuranceOfFace that gives you the first occurrence of a face. The data structure for edge 1 of our example after running constructFaceCycles would look like this:

Edge { Start = 1, End = 2, Face1=1, Face2=2, StartNextCCWEdge=6, EndNextCCWEdge=3 }.

We now have the list of edges and our two indexes m_firstOccuranceOfVertex, and m_firstOccuranceOfFace that make up our DCEL.

Purpose of Project

The purpose of this project was to implement a Doubly Connected Edge List and include functionality to perform queries on it. A secondary purpose was to include another data structure to allow for orthogonal range search. A KD-Tree was determined to be the most suitable data structure to allow for this. The main idea behind this project was to implement the data structure and try querying it with different datasets to get an idea of the time it takes to perform certain queries.

Range Search

The orthogonal range search process is performed by making use of the KD-Tree's ability to search for points within a multi-dimensional orthogonal range. We can use this ability to find all verticies within the search range. The next step is to find all edges incident on those verticies. The DCEL provides a very efficient way to do this for a single vertex. In my trials more time was spent creating a unique list of these edges then acquiring the edges themself. We then take those edges and create a unique list of the faces on either side of each. Again, removing duplicates took longer than creating the list. With the list of faces we query the DCEL for all edges that make up each face. We then have all edges surrounding areas who have at least one vertex in the search area. The search is an algorithm bounded by O(nlogm) which can be attributed to the sorting required to create a unique list.

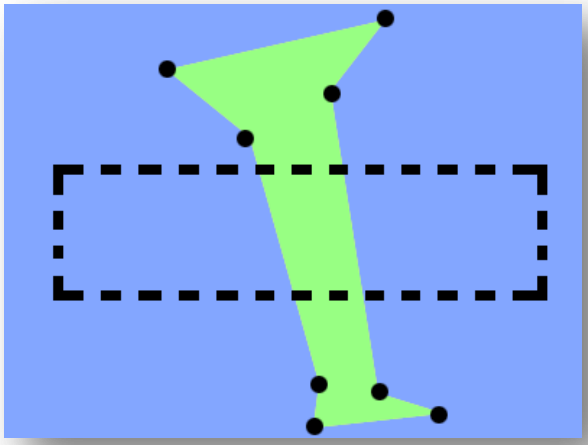

Figure 3

The weakness in this vertex based approach is we can miss areas that intersect our search rectangle, but do not have any verticies within it. Consider the example in Figure 3. We have a search rectangle that intersects both blue and green areas. However there are no verticies within the search area, so the query returns nothing. Obviously this query would be useful for some applications while not appropriate for others.

Software User Guide

The program I developed has 7 main functions:

1. Read a GML file and create the data structures to display it and perform queries on it.
2. Save the file in DCEL format.
3. Read the DCEL format to display it and perform queries again.
4. View the dataset.
5. Find all edges around a face.
6. Find all edges around a vertex.
7. Perform an orthogonal range search.

Read a GML File

To read a GML file we need one that contains lines or areas and it must be formatted in GML 1.0. You can use this online conversion tool to turn pretty much any vector format dataset into a GML 1.0 file.

http://mygeodata.eu/apps/converter/main_EN.html?dataType=vector

If you use this tool the resulting GML file should be as the application expects. I have provided several working GML files along with the source code and binary. I suggest using NorthAmerica.gml. It contains the political boundaries of north America (all provinces/states).

To read a GML File, open the application and select from the File menu the “Load .GML File” command. An open file dialog will be displayed and you can select the GML file you want to load.

At this point the application will go through the following steps to create the data structures needed. You can follow the progress in the applications’ status bar.

1. Load the GML File into main memory and parse its contents to get all lines / areas.
2. Create a Vertex/Edge map by finding all points that have the exact same spatial coordinates and considering them to be one vertex with multiple edges.
3. Create the DCEL:
    a. Create a list of edges, ordered clockwise for each vertex.
    b. Construct the vertex cycles as described in the DCEL construction section above.
    c. Construct the face cycles as described above.
    d. Query the DCEL to get the edges for each face and cache this information to be used each time the screen is printed.

The KDTree is not constructed until the user tries to use the orthogonal range search function since it takes the most time and is not necessarily needed or wanted each time we load a GML file.

Save DCEL File

Once we have a GML File opened we can save the resulting DCEL format to the disk to speed up loading the dataset next time. To perform a save, select from the File menu the “Save as .DCEL File” command.

A save file dialog will be displayed and you can choose a filename and location where you want the file to be saved.

Read DCEL File

To read a file that was saved as .DCEL we can use the "Load .DCEL File" command from the File menu. An open file dialog will be displayed and you can choose which .DCEL file you would like to load.

Viewing the dataset

There are several features written to help view the dataset you have opened.

Layers

When you open a GML or DCEL file the application will create several layers that can be optionally turned on or off by pressing the number keys above the letters on your keyboard (not the number pad on the side) . The most useful layers are the first two which print the areas (faces) and print the lines (edges). There are also layers that will print labels for each edge, labels for each vertex, and a list of faces with their comprising edges and an indication on which direction those edges must be travelled to construct the area. Additionally, each time you make a query to the data structure an extra layer will be added to the list so you can view that query or turn it off. Orthogonal range search only has one layer that is replaced if another range search is performed.

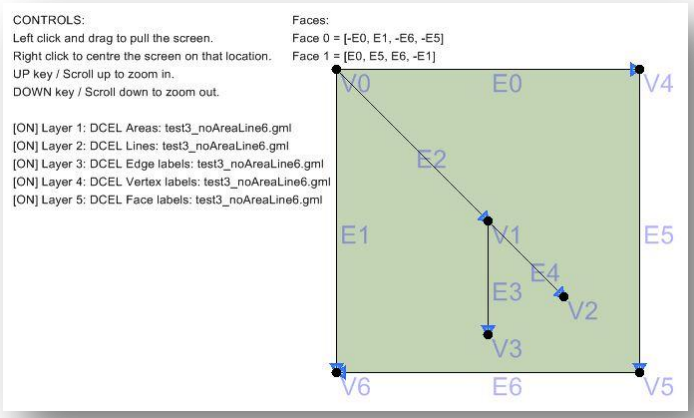

Figure 4

Pan/Zoom

The pan and zoom functions allow you to view the dataset in its entire precision. You can pan the data by right clicking on the map where you want the centre to be, or you can hold down the left mouse button and drag to move the map. To zoom you can either use the mouse's scroll wheel, or as some touchpads don't have a scroll wheel functionality I have included the option to press the up arrow to zoom in and the down arrow to zoom out.

Find edges around a face

You can query the DCEL to find all edges around a face by selecting the "Edges of Face" command from the "Query DCEL" menu. A dialog box will be displayed telling you the number of faces and asking you to input the face number you want to query. The application then performs the query and creates a layer to display the results to you.

Find edges around a vertex

You can perform this query by selecting the "Edges on Vertex" command from the "Query DCEL" menu. A dialog box will be displayed telling you the number of verticies and asking you to input the vertex number you want to query. The application then performs the query and creates a layer to display the results to you.

Orthogonal Range Search

The orthogonal range search uses multiple data structure queries to accomplish a vertex based search for areas intersecting the search rectangle. To perform an orthogonal range search you must select the "Orthogonal Range Search" command from the "Query DCEL" menu. This command may require a KDTree to be constructed if it has not already been constructed for this file. The KDTree is not saved along with the DCEL so even loaded DCELs will require a KDTree construction the first time they perform a range search. If a KDTree needs to be created you will see a popup window telling you. Press okay, and watch the progress in the status bar below. It can take a bit of time, but watching the percent complete increase makes it go by a little quicker than not having that information.

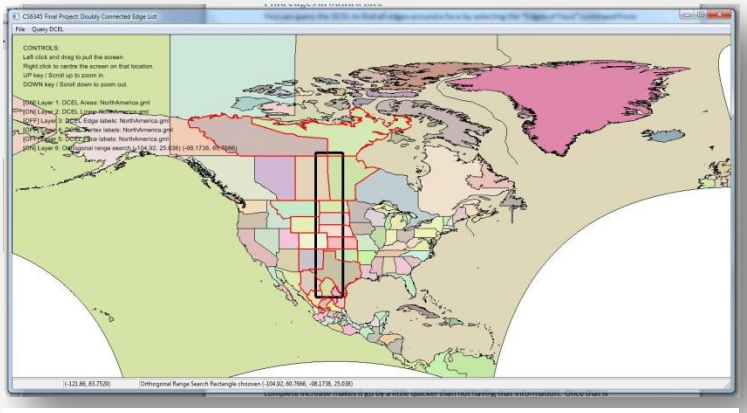

Figure 5 – Range search box indicated as black rectangle. Edges found are printed in red.

Once that is complete a popup window gives us instructions on how to perform the range search. It will tell you to press okay and then click on the map where you want the points that define the rectangle to search. Once you have clicked the map twice the application has the two points that define the search area. It then goes off and performs a range search on the KDTree to find all verticies in range. Taking each vertex it queries the DCEL to get the edges incident on it and adds it to a unique list of edges. It next finds all faces on either side of the edges and adds them to a unique list of faces. Finally, it queries the DCEL to get all edges around those faces. We then save all those edges and create a layer to optionally display them. The layer prints all those edges in red and prints the search rectangle as a black 3px wide box. Once all that is complete a report is generated and displays information about how long each query took as well as the total time required. It is displayed in a popup window

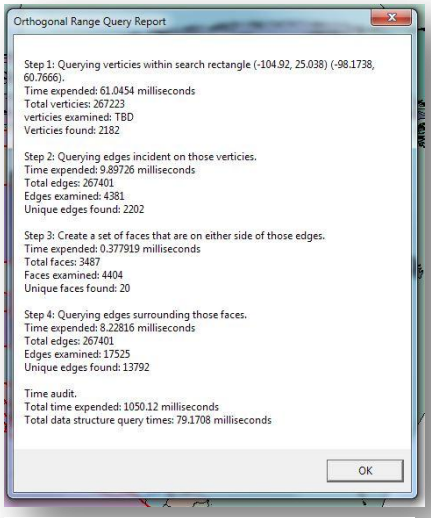

Figure 6 – Detailed report of time to perform range query.

similar to the image in Figure 6.

Testing and Evaluation

Test datasets were crucial in the construction of this application. The datasets I used to test this application are included in the source and binary distributions. Some test cases are shown in Figure 7 and the more interesting ones are described below.

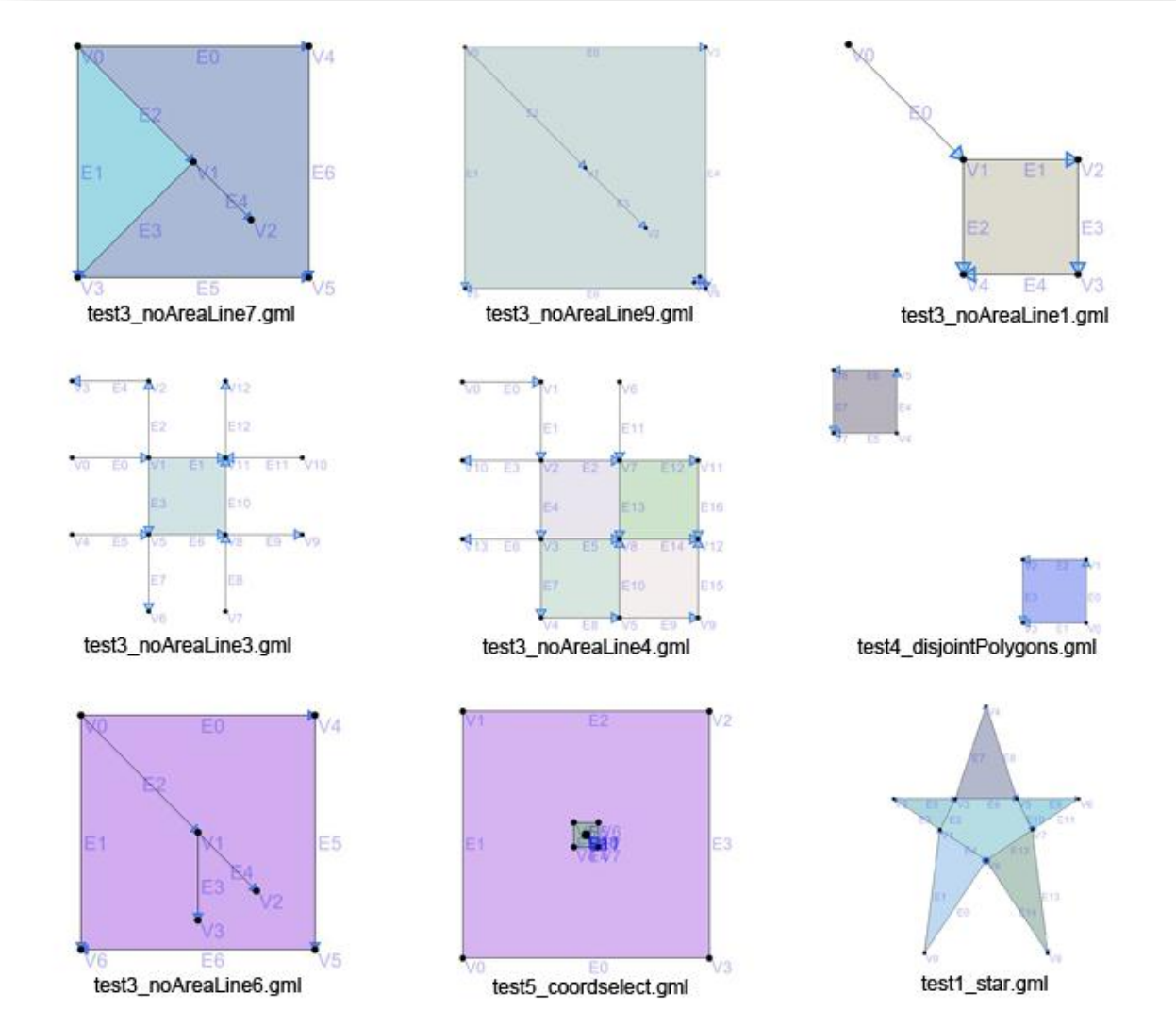

Figure 7

In implementing the DCEL for generic GML data I ran into disjoint polygons. As the DCEL isn't designed to handle these properly some adjustments to the queries were required as to not run into infinite loops. You can see the disjoint polygon test case in Figure 7 and can open it by loading

test4_disjointPolygons.gml. You can notice by turning on layer 5 that we still have problems as the exterior face is created twice. The first is created as we traverse the exterior of one square and then another is created when we traverse the second square's exterior.

Another problem I ran into was the existence of dangling edges in my data sets. As these datasets are amalgamations of lines, we sometimes have areas with lines protruding out or into them. This was causing the query to get the edges of a face to run into infinite loops as the edge would have the same face on either side. You can see a few of the test cases I wrote while attempting to fix this query in Figure 7. All files named similarly to test3_noAreaLineX.gml are tests created to help debug the problem.

When trying to get the range search working I needed to convert mouse clicks on the screen into their latitude and longitude to create the search rectangle. I created test5_coordselect.gml to have coordinates that were easy to remember for each point so I could test the click to latlong feature.

References

[1] Ryan Holmes. "The DCEL Data Structure for 3D Graphics". Internet: http://www.holmes3d.net/graphics/dcel/. December 18, 2012.

[2] "MyGeodata Converter - vector". Internet: http://mygeodata.eu/apps/converter/main_EN.html?dataType=vector. December 18, 2012.

[3] H. Samet. Foundations of multidimensional and metric data structures. Elsevier/Morgan Kaufmann, 2006

[4] D.E. Muller and F.P. Preparata. "Finding the intersection of two convex polyhedral", Theoretical Computer Science 7, 1978, pp.217-236.

[5] Denis Ponomarenko. " Simple MFC independent Open-Save File Dialog class". Internet: http://www.codeproject.com/KB/dialog/OSFdialogclass.aspx?display=Print, March 13, 2007. December 18, 2012.

[6] "Autoserial library". Internet: http://home.gna.org/autoserial/. December 18, 2012.

[7] Marcin Kalicinski. "RapidXml". Internet: http://rapidxml.sourceforge.net/. December 18, 2012.