Victoria Wang
Mentor: Prof. Ellen Veomett
SMC SRP '12
9/17/2012

Determining Conditions for the Edge Isoperimetric Inequality for Graphs in $\mathbb{Z}^2$

Abstract

The edge isoperimetric inequality problem is presented along with a programming and graphical approach to analyzing properties of the shapes of vertex sets that fulfill the condition of minimal edge boundary for a given vertex set size.

Background

Graph Theory

For those who are unfamiliar with graph theory, a brief overview should suffice.

A graph G(V,E) is defined as a combination of a set of vertices V (represented as points) and a set of edges E (connectors between these points), specifically some multiset of the subsets of V.

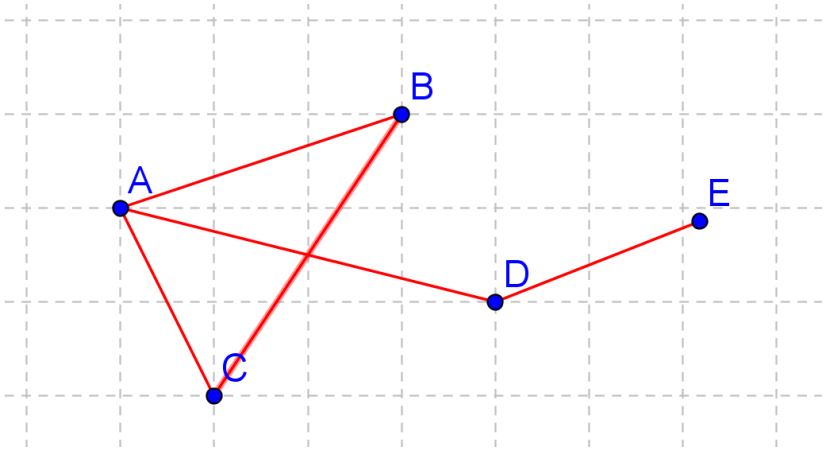


Figure 1

Figure 1 above is an example of a graph in which V = {A, B, C, D, E} and E = {(A,B),(B,C), (A,C), (A,D), (D,E)}.

The Edge Isoperimetric Inequality

Now most of us are familiar with the problem of finding the maximal area that can be circumscribed using a perimeter of fixed length. The solution can be proven using calculus to be that of a circle. Likewise, in a metric space, the solution of minimal volume is that of the Euclidean ball. (A metric space (X, d) is a set X, along with a function d:X*X → R+ that acts as a distance function.)

All other areas circumscribed by this fixed length is then less than or equal to the area of this circle, hence the inequality. An equivalent problem is that of finding the minimal length for a given fixed area.

Victoria Wang
Mentor: Prof. Ellen Veomett
SMC SRP '12
9/17/2012

This leads us into the edge isoperimetric inequality, which similarly, attempts to find a minimal edge boundary for a given vertex set of fixed size.

The edge boundary of a given set is given as $\partial e(A) = \{(x,y) \in E : |A \cap \{x,y\}| = 1\}$.

In other words, all of the edges that connect from the given vertex set to the surrounding vertices on a grid that are "one away" are part of the edge boundary. This includes vertices that are along a diagonal or to the left, right, above, or below a vertex in your set.

In my research we examined the properties of the shapes of a vertex set with minimal edges for a given number of vertices in $\mathbb{Z}^2$.

For example, let's us take seven as the size of the vertex set that we are attempting to find a minimal edge boundary for. In Figure 2 below, we are taking the vertex set in blue to be one example of the potential vertex set shapes of size 7. Figure 3 presents the same shape, with the edge boundary of the vertex set in red.

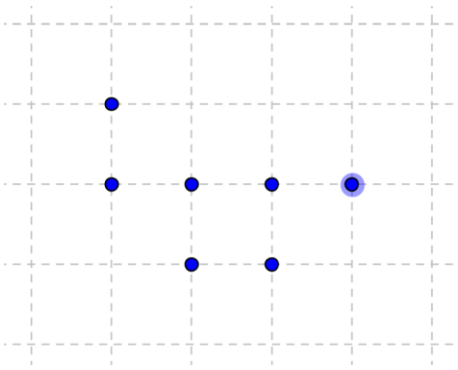


Figure 2

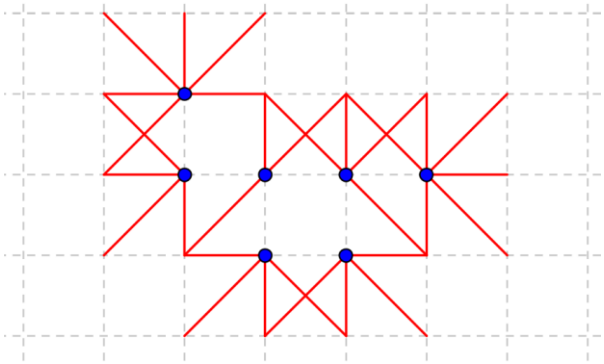


Figure 3

Examining vertices with minimal edge boundaries gives us many applications in higher mathematics.

Approach

We deciding to tackle the problem of determining properties of this minimal edge boundary vertex set by simultaneously pursuing two separate methods: computer programming and a theoretical approach to examining properties of a vertex set.

MATLAB programming: EdgeCalculator

Computer programming is an important approach for examining the properties of vertex sets because it allows for very fast generation of vertex sets in contrast to the tedious error-prone method of hand-drawing.

Victoria Wang
Mentor: Prof. Ellen Veomett
SMC SRP '12
9/17/2012

The MATLAB programming language was chosen due to familiarity with the language and the fact that I had easy access to it on my computer. Additionally, MATLAB has very simple plotting features which would quickly graph results, something that was important in this project.

In directly creating vertex set shapes of minimal edge boundary, it was first necessary to be able to generate all possible shapes of the given user-defined vertex set size. The edges of each of the possible shapes was then calculated and then sorted to find the minimal edge. The shape corresponding to this minimal edge as well as the edge number was then output to the user.

A general outline of the main functions and subfunctions utilized in the EdgeCalculator program is given below before they are examined in some more depth.

```
%EdgeCalculator program
%written by Victoria Wang


% DESCRIPTION:
% This program is meant to generate a set of all the unique free polyomino
% shapes given user input of a number of vertices. Also, it will calculate
% the number of edges for each shape, sort those shape in order of ascending
% number of edges, then output a picture of one of the shapes with a minimum
% number of edges as well as that number.


%           OUTLINE
%_____________________________________________
% calculating shapes:
%    The four functions below create an array of coords for shapes of given
size
% + 1)FindOpen - Creates an array with points that are 'open' to be added
%                   for the shape of next size
% + 2)GenShapes - adds points inductively using array of open points
generated from FindOpen

% + 3)translate2 - translate midpoint x and y values to origin along w/shape
% ~ 4)CheckRot - rotates each shape by 90, 180, 270, 360 and checks for
%                   repeats and deletes them (reflections as well)

% + 5)dist2 - calculates the distance between 2 points
% + 6)CalcEdges - Calculates the number of edges for a given set of vertices
% + 7)SortShapes - sorts in ascending order the shapes according to number of
edges
%_____________________________________________
```

FindOpen

In generating all the potential shapes, or polyominoes, of a vertex set the problem was attempted by utilizing the approach suggested by munshkr of the Parallel Stripes blog [3]. (Polyominoes, which are blocks of cells connected to one another by their sides. For example, the familiar tetris shapes are polyominoes of size four.

That is, we started with a base case of vertex set size 2. The points for this base case were defined within the program. Each subsequent shape of size n_i is then created inductively by the previous shape of size $n_{(i-1)}$. To do this, we must first find all the available "open points" that can be added to the previous set, then add each open point to that set, generating a new shape in the process.
For example, given a base case of size two where the two vertices lie on a horizontal line, the number of open points would be six: the point directly to the left of the first vertex, the two points directly above and below the first vertex, the two points directly above and below the second vertex, and the point directly to the left of the second vertex.

(In a sense, the open points for a vertex set is essentially it's vertex boundary excluding the diagonal elements)

The number of open points to be added was defined in the function FindOpen.

Determining the points available turned out to be quite a delicate procedure as it was dependent on the distribution of vertices and needed to avoid double counting.

The procedure was started by examining each vertical line around the vertex set. Initially, for each set, the vertices with the minimal x value were found for each shape. Then starting from the lowest y value with this corresponding x value, the coordinates of the open points on the left of these vertices were added to the available list of open points. Next, the top and bottom of these vertexes with the minimal x value coordinate were added into this list.

The difficult part involved the different lengths of the following vertical lines due to double counting issues. To take this into account, the maximal y coordinate of the vertical line is found and the difference between this point and that of the maximal y coordinate of the next vertical line is calculator, and if the next line is taller by more than 1 line, the vertices that are more than 1 above the previous maximal y-coordinate have the open points on their left added in.

Figures 4 through 6 show the first few steps in such a process.

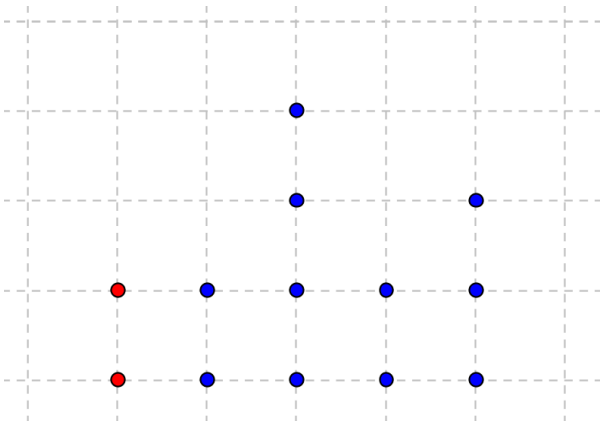


Figure 4

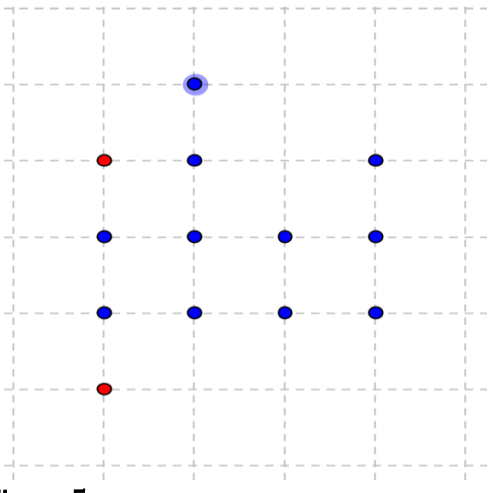


Figure 5

Victoria Wang
Mentor: Prof. Ellen Veomett
SMC SRP '12
9/17/2012

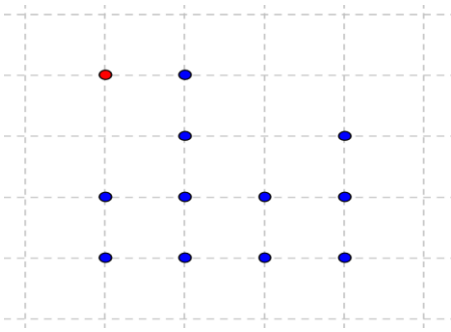


Figure 6

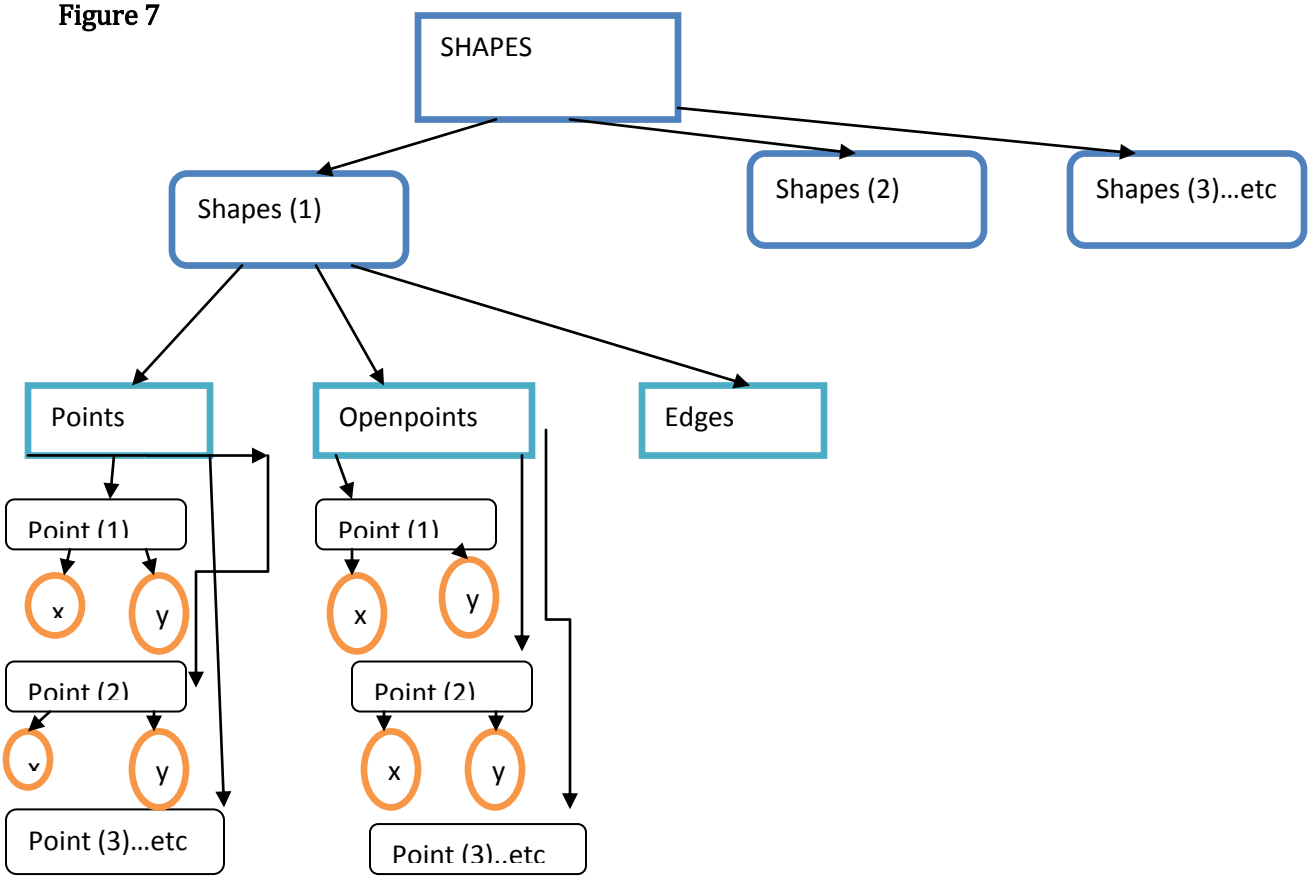
GenShapes

Next, each of the open points are added to the original vertex set. Each addition then generates a new shape. So for example if you had say Shape 1, and there were 3 possible open points, then this function would generate Shape 2 (shape 1 + first open point), Shape 3 (shape 1+second open point), and Shape 4 (shape 1+third open point).

This involved some rather complicated structure conversions since the original structure most be changed into an even large super-structure that has the old structures as its fields.

The overall format of the program utilized the structure Shapes, with fields Points, Openpoints and Edges as in Figure 4 below.

Figure 7



Victoria Wang
Mentor: Prof. Ellen Veomett
SMC SRP '12
9/17/2012

Translate2 and CheckRot

Next, the program had to eliminate redundancies, specifically rotations and reflections. This was done through the combination of the translate2 and CheckRot functions.

The translate2 function found the midpoint of each vertex shape and then translated all the vertices such that the new midpoint was now at the origin.

The CheckRot function would rotate each shape by 90 degrees four times as well as reflecting each rotated shape, checking these points against the other points each time. Thus each shape would each be check 8 times. If any redundancies were found, these redundancies would be deleted. These rotations were accomplished by using a Cartesian to polar conversion.

Dist2 and CalcEdges

The next portion was to calculate the edges of each unique shape. The CalcEdges function used the dist2 subfunction to determine the distance in between vertices within the vertex set.

The dist2 subfunction just involved a simple distance formula. It calculated the distance between two vertices using the distance formula sqrt((x2-x1)^2 + (y2-y1)^2).

The CalcEdges function worked by noticing that the maximal total of edges outgoing from a vertix is 8 as shown in Figure 8. Each additional vertex that is adjacent to the vertex observed "takes away" an edge and reduces the number of outgoing edges by one.

Thus, the number of edges was calculated by multiplying the number of vertices by 8 and subtracting one for each adjacent vertex (which was found by using the distance function to determine if the distance to another vertex was equal to 1 or the square root of 2).

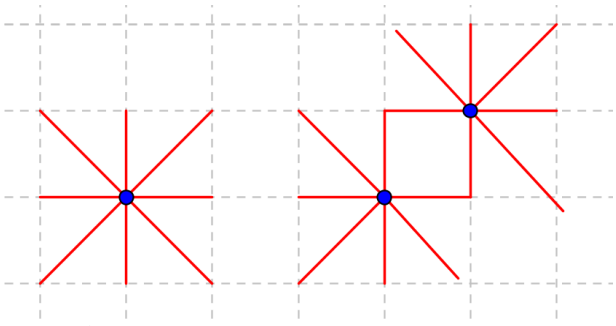


Figure 8

Sort

Lastly, a generic sort function was implemented of the type found within most introductory computer classes. This function would not only sort in ascending order the number of edges, but would rearrange the corresponding shapes as well.

Victoria Wang
Mentor: Prof. Ellen Veomett
SMC SRP '12
9/17/2012

Overall structure

The main program takes in a user generated input of the number of vertices, and utilizes all the functions in a loop to successively generate shapes up until the number specified by the user. Once accomplished, the program outputs a graph with the shape and number of edges.

Observations and Issues

The shapes generated for the minimal edge boundaries first few vertex points were the same as predicted by the minimal vertex boundaries, however we know that at higher sizes, these two differ and as shown by Bollobas and Leader [1], minimal edges sets are not nested. Unfortunately, the program became extremely slow at vertex sizes of 8 and above.

This is due to the fact that the main program as well as each function utilizes many nested loops that run through all the elements within the entire structure multiple times, making the time necessary to calculate it superexponential with each additional iteration.

For the future, a number of things can be done to mitigate this problem. This includes making the code more efficient. Also, as well be discussed next, the issue of compression can be brought in. Each set can be first compressed, and then checked for redundancies, leading to a large number of extraneous calculations being eliminated.

Theoretical Approach

For each vertex set of minimal edge boundary we are attempting to find a more mathematical description such as a sort of upper bound in the number of edges in relation to the structure of the vertex set shape.

The first thing to prove is something that we took as a sort of obvious assumption when designing our code: that vertex sets with spaces or 'gaps' in between do not minimize the edge boundary. This approach involves analyzing each vertex set piece by piece. The sets are split into horizontal lines and the outgoing edges that emanate from each line into nodes within the other lines are examined. This is a similar approach to the one taken by Veomett and Radcliff on the vertex isoperimetric inequality problem [2].

The proof of compression, or 'squeezing the gaps closed' will be shown graphically.

The definition of the compression that I will be using here will be this:

Given a set of points that all lie in a horizontal line, take the x-values of these points and renumber them using the following: Start at 0, then continue with 1,-1 , 2, -2, etc so that lines containing an even number of vertices will protrude to the right by a factor of 1.

A similar procedure is then done vertically to the y-values.

Figure 9 shows the effect of such a compression.

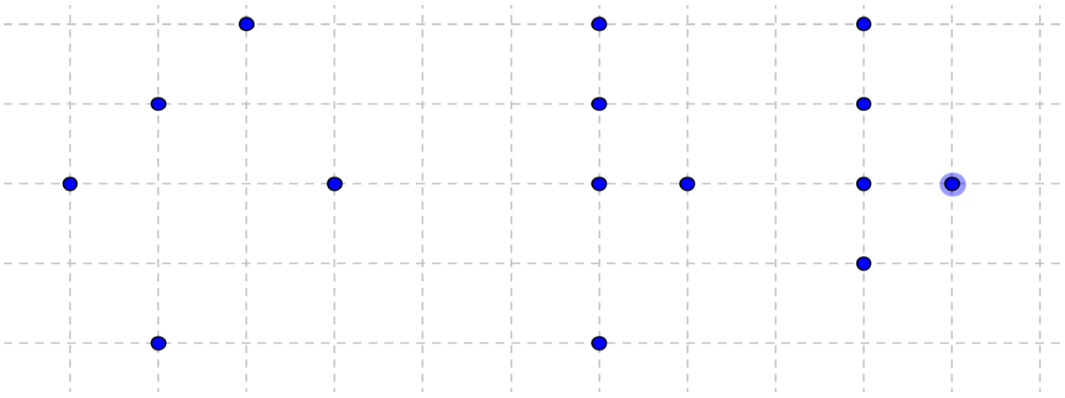


Figure 9

Using this definition, we first examine vertex sets with gaps prior to being compressed, and examine the effect of compression on the number of outgoing edges.

For a given horizontal line under examination, it will have two neighboring horizontal lines, one above and one below. The number of total edges outgoing from the line under consideration will thus be the number of edges that reach into the neighbor above, plus the number of edges that reach into the neighbor below, plus the two edges on either side of its endpoints (left and right).

Figure 10 below is an example of a vertex set with the maximal number of possible edges into the neighbor above. The line under considering contains the blue vertices. The orange vertices are also contained within our vertex set, but are not part of the line we are considering. They are instead part of the upper neighboring line.

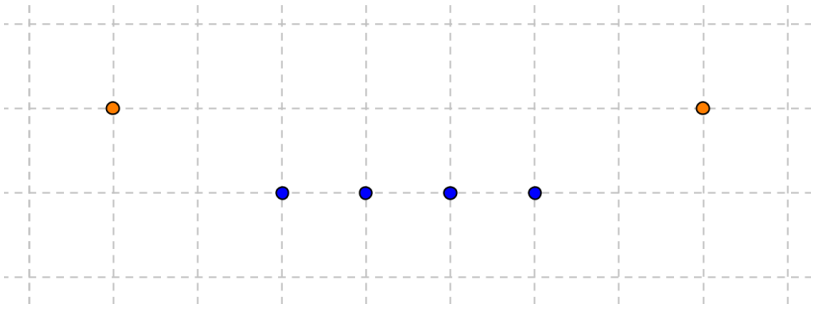


Figure 10

As we can see from Figure 11, the maximal outgoing vertices is then 3*the number of vertices in the observed line. Here, 4*3 or 12 edges.

Victoria Wang
Mentor: Prof. Ellen Veomett
SMC SRP '12
9/17/2012

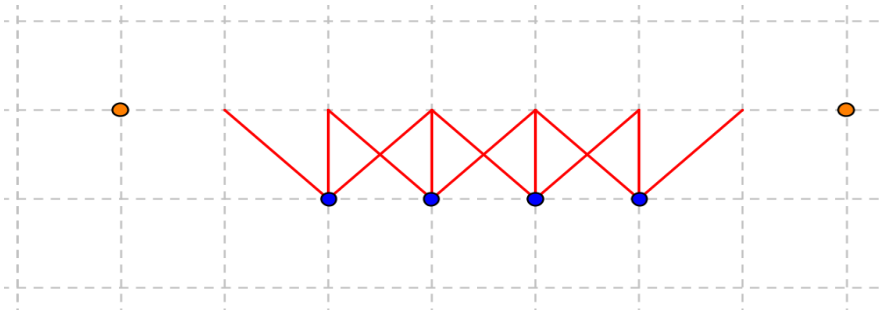


Figure 11

As shown in Figures 12, 13, and 14, the movement of the upper-right vertex in the upper neighboring line towards the direction of the observed vertices and minimizing the gap, decreases the number of outgoing edges.

As a general rule:
-For each vertex on a diagonal from an endpoint of the observed vertex line, subtract 1 from the maximal possible total edges.

-For each vertex right above an end vertex subtract an additional 2 edges to get a total of (1+2 or 3 less than the total possible maximal edges.

-For each vertex above one of the center vertices, subtract an additional 3 edges to get a total of (1+2+3 or 6 less than the total possible maximal edges.

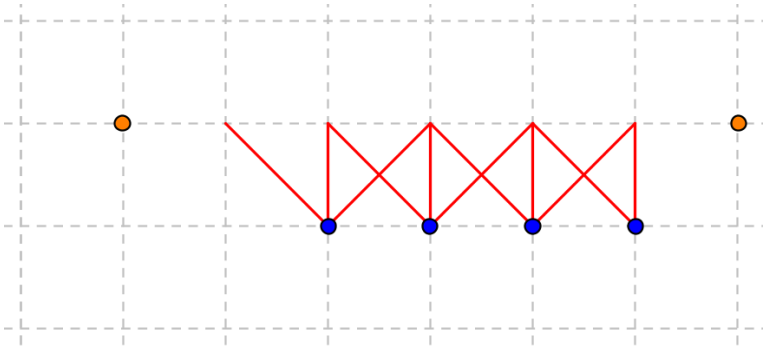


Figure 12

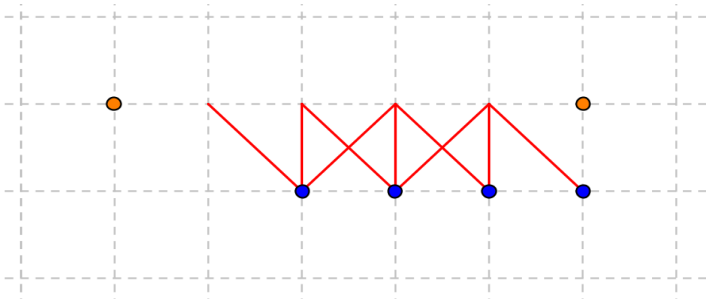


Figure 13

Victoria Wang
Mentor: Prof. Ellen Veomett
SMC SRP '12
9/17/2012

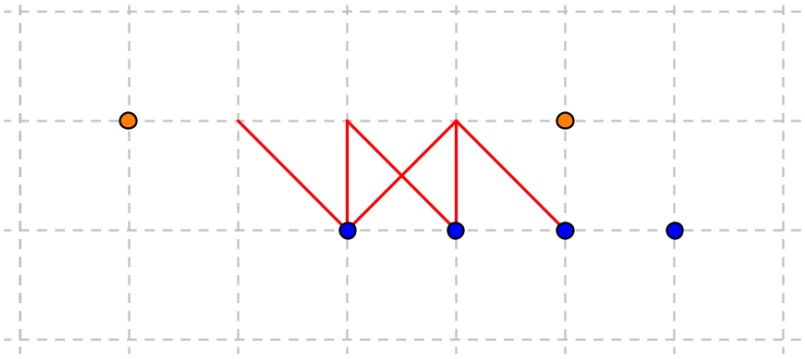


Figure 14

As you can see, each additional gap reduces the maximal number of edges outgoing from a given line, thus compression does not lead to an increase in the number of edges and either does nothing to it or decreases the number of possible edges.

Thus, gaps can be eliminated using the form of compression that has been previously proposed without fear that a possible shape of minimal edge boundary has been eliminated.

Now that these gaps are eliminated, let's take a look at the nature of shapes that do not contain any gaps. The same concept of horizontal neighbors can be used as before.

The compressed shapes can be broken down into more simple structures.

They are A)upper neighbor lines of equal length (as in Figure 15), B)the upper neighbor has one less vertex (as in Figure 16), C)the upper neighbor has one more vertex, the upper neighbor has two or more vertices (Figure 17) or the upper neighbor has two or less vertices.

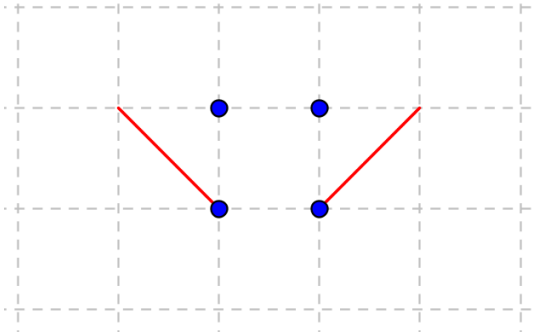


Figure 15

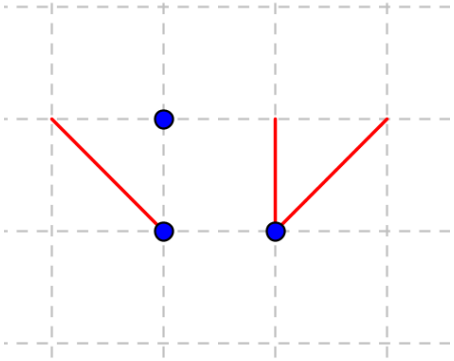


Figure 16

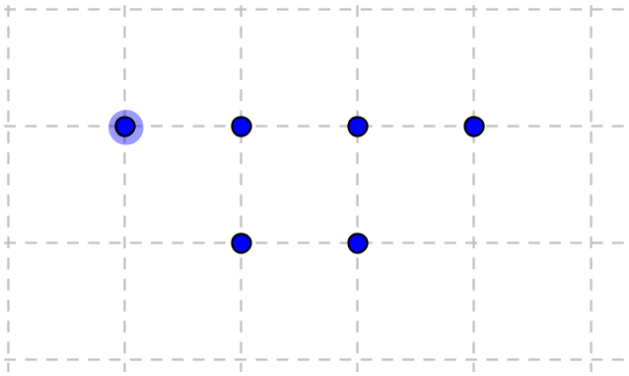


Figure 17

For condition A we can see that this implies that there are 2 outgoing edges into the upper neighboring line. Additionally for condition B this means 3 edges, 1 edge for condition C and 0 edges for condition D.

For condition E, the situation is a bit more complicated. If the given line protrudes by more than 2 vertices on one side, then for each protrusion there is 1 edge from the vertex directly below, 2 edges for one that pops out by 1, then 3 edges for each remaining additional vertex beyond that.

Overall we see that there does to appear to be certain conditions for the vertex sets of minimal edge boundary. For example, there probably should not be protrusions beyond two due to the increase in number of edges.

Conclusion

In this paper we have examined some of the properties of the edge isoperimetric inequality for a vertex set of fixed size using both graphical and computational methods. Our computational method ensures that all possible shapes for a vertex size are generated and the minimum edge shape is given. However, the computational approach is limited by loss of speed at higher vertex values.

Using graphical methods we have proved that vertex sets with gaps are not within the sets that contain the minimal edge boundary. We have also examined certain properties in regards to the shape of the compressed sets of minimal boundary.

Victoria Wang
Mentor: Prof. Ellen Veomett
SMC SRP '12
9/17/2012

References

[1] Bela Bollobas and Imre Leader. Edge-Isoperimetric Inequalities in the grid. Combinatorica, 11(4):299-314, 1991.

[2] Ellen Veomett and A.J. Radcliff. "Vertex Isoperimetric Inequalities for a Family of Graphs on $\mathbb{Z}^k$"

[3]"Generating Polyominoes." http://parallelstripes.wordpress.com/2009/12/20/generating-polyominoes/

Victoria Wang
Mentor: Prof. Ellen Veomett
SMC SRP '12
9/17/2012

Appendix

Sample of some of the subfunctions used

Openpoints

```
function openpoints = FindOpen (shapes, vertix)
%takes inputs of shapes structure and number of vertices
%outputs array openpoints with fields x and y which is added onto shapes
%structure

%Find minimum x value and min y value given that min x value

  minimum_x = shapes.point(1).x;
  miny_given_minx = shapes.point(1).y; %arbitrary assignment
  for i = 1:vertix
    if shapes.point(i).x < minimum_x
      minimum_x = shapes.point(i).x;
      miny_given_minx = shapes.point(i).y; %arbitrarily assign y value
    end
  end


  % add a possible open spot for each point to the left
  j = 0;

  for i = 1:vertix
    if shapes.point(i).x == minimum_x
      j = j+1;
      %shapes.openpoints = zeros(j) ;
      shapes.openpoints(j).x = minimum_x - 1;
      shapes.openpoints(j).y = shapes.point(i).y;
    end
  end

  j = j+1; % increment openpoints counter

    % find max and min y value for the given min x value
  for i = 1:vertix
    if shapes.point(i).x == minimum_x && shapes.point(i).y < miny_given_minx
      miny_given_minx = shapes.point(i).y;
    end
  end

  maxy_given_minx = miny_given_minx;
  for i = 1:vertix
```

```
    if shapes.point(i).x == minimum_x && shapes.point(i).y > maxy_given_minx
    maxy_given_minx = shapes.point(i).y;
    end
  end

  % add open spot to top and bottom
  shapes.openpoints(j).x = minimum_x; %Overloading problems?? should be ok
  shapes.openpoints(j).y = maxy_given_minx + 1;

  j = j+1;

  shapes.openpoints(j).x = minimum_x;
  shapes.openpoints(j).y = miny_given_minx - 1;



  % max x value

  maximum_x = shapes.point(1).x;
  for i = 1:vertix
    if shapes.point(i).x > maximum_x
      maximum_x = shapes.point(i).x;
    end
  end


prevx = minimum_x;

while prevx  ~= maximum_x


    %find the difference between max y value of next x value

    maxy_given_nextx = -500;

      %find max y value of next x value
      for i = 1:vertix
        if shapes.point(i).x == (prevx+1) && shapes.point(i).y > maxy_given_nextx
          maxy_given_nextx = shapes.point(i).y;
        end
      end
    %maxy_given_nextx

  diff = maxy_given_minx - maxy_given_nextx;

  %diff
```

```
% if + (1st x value's y value is greater), add those points

if diff > 1
   for i=1:diff
      j = j+1;
      shapes.openpoints(j).x = prevx+1;
      shapes.openpoints(j).y = maxy_given_nextx + i;
   end
end


% if negative then minus one,  the negative values each have one to lt

if diff < 0
   diff = -diff;
   diff = diff - 1;
   if diff ~= 0
      for i=1: diff
         j = j+1;
         shapes.openpoints(j).x = prevx;
         shapes.openpoints(j).y = maxy_given_minx + 1 + i;
      end
   end
end

%also for the bottom....

miny_given_nextx = 500;
%find the difference between min y value of next x value

   %find min y value of next x value
   for i = 1:vertix
      if shapes.point(i).x == (prevx+1) && shapes.point(i).y < miny_given_nextx
         miny_given_nextx = shapes.point(i).y;
      end
   end

%miny_given_nextx

diff2 = miny_given_minx - miny_given_nextx;
%diff2

% if - (1st x value's y value is greater), add those points
%j
if diff2 < -1
   diff2 = -diff2;
   for i=1:diff2-1
```

```
        j = j+1;
        shapes.openpoints(j).x = prevx+1;
        shapes.openpoints(j).y = miny_given_nextx -1- i;
      end
    end
    %j
    % if + then minus one,  the negative values each have one to lt

    if diff2 > 0
      diff = diff - 1;
      if diff ~= 0
        for i=1: diff
          j = j+1;
          shapes.openpoints(j).x = prevx;
          shapes.openpoints(j).y = miny_given_minx - 1 - i;
        end
      end
    end


   j = j+1;

   % add open spot to top and bottom
    shapes.openpoints(j).x = prevx+1;
    shapes.openpoints(j).y = maxy_given_nextx + 1;

    j = j+1;

    shapes.openpoints(j).x = prevx+1;
    shapes.openpoints(j).y = miny_given_nextx - 1;

    prevx = prevx+1;

end
    %reloop this

% at end also add all to the right of last row

for i = 1:vertix
    if shapes.point(i).x == maximum_x
      j = j+1;
      shapes.openpoints(j).x = maximum_x + 1;
      shapes.openpoints(j).y = shapes.point(i).y;
    end
end

   openpoints = shapes.openpoints;
```

```
end
```

CalcEdges

```
%CalcEdges
%Inputs array of shape vertices; Outputs number of Edges
% Input array of Shape Coordinates, numvertices, Output Number of Edges

function numEdges = CalcEdges(shapes, numvertices)
numEdges = numvertices*8; %Check to see if . is needed

   for i = 1:numvertices
      for j = 1:numvertices
         if dist2(shapes.point(i).x, shapes.point(i).y, shapes.point(j).x, shapes.point(j).y) == 1 ||
dist2(shapes.point(i).x, shapes.point(i).y, shapes.point(j).x, shapes.point(j).y) == sqrt(2)
            numEdges = numEdges -1;
         end
      end
   end
end
```