Vectorization

Deborah Alves, Grace Gee, Victoria Gu, Styliani Pantela

Overview

Bitmap images can appear pixelated when zoomed in, so we want to be able to represent bitmaps in a high-quality format that can scale to any size without any loss of quality. In order to accomplish this, we implemented the Potrace algorithm for vectorizing and scaling bitmap images. (http://potrace.sourceforge.net/potrace.pdf).

We implemented an algorithm based on the paper mentioned above to vectorize images of 2 colors (black and white). Our main goal was to get a bitmap input from the user and return the optimal vectorized polygons. In the end, we were able to output higher resolution images from bitmaps that can be manipulated without loss of quality. In the future, we could adapt our program to export the vectors to image editing programs, or display in different colors and formats.

The process involved three steps. The first was constructing a directed graph from the bitmap and finding all the closed paths on the graph on paths and boundaries. The second was finding an optimal polygon from the closed paths. The third (which was an extension) was constructing smooth Bezier curves from the polygons.

How to use

Installation:
Need to have ocaml already installed on your computer. Put all of the files in the same folder.
Execution:
Navigate to the folder containing all the files, open the terminal and type “make output”. Then execute:
./output INPUT_FILE OUTPUT_FILE SCALE
  INPUT_FILE: direction to input file, from the current folder
  OUTPUT_FILE: name of output file, without extension. It will
  be saved in testsoutputs/OUTPUT_FILE.svg
  SCALE: optional argument to choose the scale to see the image.
  Default is 50 (50 times bigger than original bitmap)

The output files are contained in the folder called “testoutputs. Open an internet browser and type in file:///HOME_DIRECTORY/PATH_TO_TESTOUTPUTS/OUTPUT_FILE, where HOME_DIRECTORY is the address of your home directory, and PATH_TO_TESTOUTPUTS is the path to the “testoutputs” from your homedirectory,

and OUTPUT_FILE is the file name you specified when you executed ./output..., with the extension “svg”. For example, if you specified “smiley” as the name of your output, then “smiley.svg” would be your output file. The output file will contain four images. The top-middle image will be the original bitmap image. The top-left image will be the bitmap with the grid outline on top. The bottom-left image will be the directed graph representation of the bitmap produced from step 1. The bottom-right image will be the images represented as vectorized polygons. The top-rightmost image will be the final output with some polygons represented by Bezier curves.

Planning

First & Second Milestone

We originally planned to at least implement the first two parts of the algorithm. The first involved reading in a bitmap, turning it into a directed graph, and generating a list of closed paths from the directed graph. The second part involved taking the list of paths, and for each path, generating all the possible segments that could approximate segments of the path, generating all possible polygons from the list of possible segments, and finding the optimal polygon to approximate the path. We had those two parts done and tested and found the source of our stack overflow problem.

Final Milestone

Perry suggested that we implement Dijkstra’s Algorithm on our step 2 to solve our stack overflow problem. We implemented DIjkstra’s Algorithm as well as the paper’s formulation for converting our optimized polygon to Bezier Curves.

Teamwork

Deborah and Stella worked on taking a bitmap, converting it into a directed graph, and then outputting the closed paths. They implemented functions for taking a bitmap and converting it to a matrix (that stores the coordinates and colors of the pixels in the bitmap), taking a matrix and outputting a directed graph and taking a graph and figuring out all the closed paths of the graph.

Grace and Victoria worked on taking the closed paths and generating the optimal polygon that fits the paths. Generating possible segments, calculating the penalty for each segment and the whole polygon, generating the polygon, and filtering to find the optimal polygon.

Then, Deborah and Victoria implemented a variation of Dijkstra’s Algorithm, which has a much faster run-time and more efficient memory-usage than our previous implementation for finding the optimal polygon. This involved studying the algorithm in

depth and figuring out how to adapt it to the modules that we already had.

Stella and Grace worked on the algorithms for converting closed paths to Bezier curves as described in the paper. This involves calculating float vertices from our int vertices. These are better approximations of the optimal polygon. Finally, by using the corner analysis technique described in 2.3.3 we constructed Bezier Curves.

Design and Implementation

We designed our modules early on by breaking the algorithm into three separate steps, as described above. We used the ocaml programming language in our implementation and made extensive use of abstractions in case we wanted to change our data structures, and also to allow independence between the implementation of different parts. Our module designs worked well when we implemented them except for part 2 when we were using an algorithm to find the optimal polygon but ran into space and running-time issues.

By the first project checkpoint, we had the first two parts of the algorithm working on small input images, but on larger images it caused segmentation faults. We figured out that the way we implemented the second part of the algorithm was inefficient in its use of memory, so that was the probable cause for our segmentation faults on large images. We were generating all possible segments (storing as a list of lists) and all possible polygons (also as a list of lists) and then comparing the polygons to find the optimal one. We found a more efficient implementation by implementing a variation of Dijkstra's algorithm, which takes in a weighted directed graph (formed from all of the possible segments) and finds the shortest possible cycle from a source vertex back to itself (a simple path). This is one possible polygon. We act this on all of the vertices, but we minimize memory usage by only storing two possible polygons at a time, comparing them and discarding the polygon with the larger weight before it iterates again. The final polygon it outputs is the polygon with the minimum number of sides (first criterion) and minimal penalty (criterion only used in the case that the number of sides of two polygons are equal).

We outputted our final vectorized images in SVG format, a form of mark-up language to visualize our final vectors. The documentation is listed here http://www.w3.org/TR/SVG/paths.html

More details on each of the steps in our algorithm follow:

Step 1:

1) Get a bitmap as an input

2) Read the bitmap and create a biColor matrix to read the pixels

3) Read the value (Black or White) of each pixel, from the bitmap bytes, and insert in the matrix. Use the matrix from the bitmap to construct a directed graph, as described in the paper. The vertices are corners of pixels (with 4 adjacent pixels not all of the same color) and the edges are set between two adjacent vertices and are determined by the position of the colors, as described in the paper.

4) Use the graph to extract the different closed paths and their colors, and return a list of paths that will be used in the next step. First, get the left most, top post vertex (using the right implementation of get_vertex). Getting the outermost path is important to guarantee that we output them in the correct order (without wrong overlaps of paths). We follow the arrows (edges) from this vertex, until we get back to the first vertex. Extract this path from the graph (by extracting the edges, and then the vertices without edges) and "save" it in the path list. Repeat the process until the graph is empty.

Step 2: int vertices -> “straight segments” -> optimized polygon

1) Get a path (vertex list) as an input

2) Read in the vertices and find “straight paths” by finding the “direction”. Read in a closed path represented by a vertex list and compute the direction for each pair (North, South, East, West). For each pair, compute the constraints on the next pixel so that each pixel will need to satisfy a “three-way” constraint. This will give us a vertex list representing the straight path.

3) Use the “straight paths” and by a variation of Dijkstra’s algorithm, we find the optimal polygon structured as a list of vertices. Use the “straight paths” and penalties algorithm to find the optimal polygon for the given list of vertices. The penalties algorithm utilizes the standard deviation of the Euclidean distance between vertices, cyclic difference, and makes sure that the polygon with the least number of sides is chosen.

4) Dijkstra’s algorithm (variation): takes in a weighted directed graph (formed from all of the possible segments) and finds the shortest possible cycle from a source vertex back to itself (a simple path). We act this on all of the vertices, but we minimize memory usage by only storing two possible polygons at a time, comparing them and discarding the polygon with the larger weight before it iterates again. The comparison is determined by the paper: first we check the number of sides, and then the sum of penalties. The "compare" function for these "weights" is defined under the module WIntGraph.

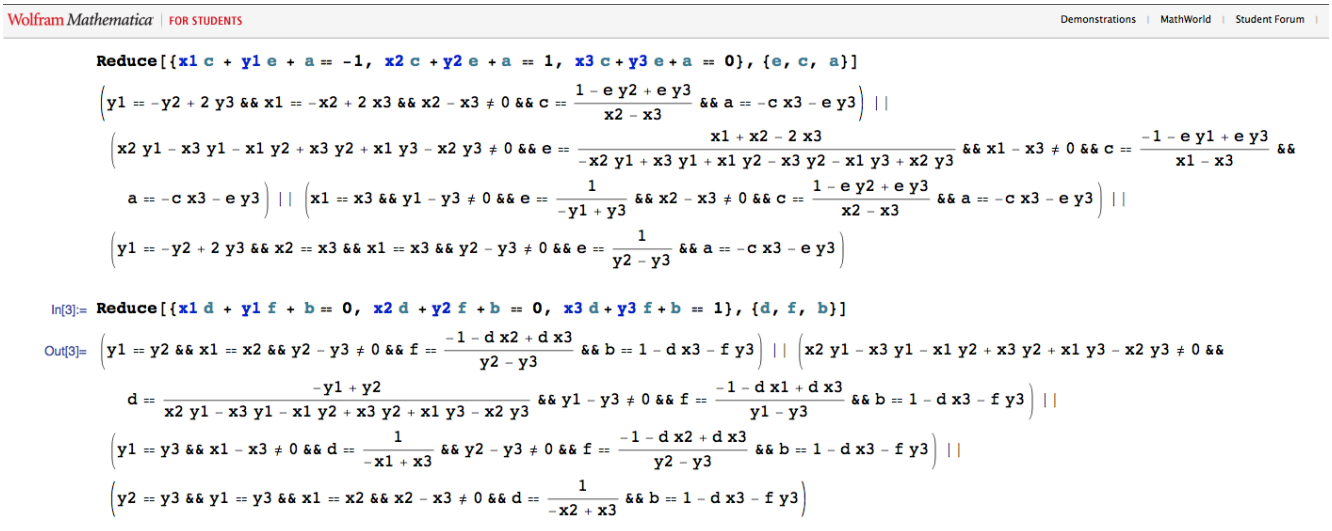
Step 3: int vertices -> approximated float vertices -> vectors

1) To calculate the adjusted vertices, we find the intersection of consecutive lines defined by neighboring vertices and find a point within 1 unit of the original vertex. The lines are found by calculating the center of gravity between consecutive vertices of the of the optimal polygon. This will give us a point on the line. To find the slope, we find the expected value of various elements of the original polygon list as defined by the paper.

2) Once we find the adjusted polygon, we can find the “c”, a point on the unit square around the adjust polygon vertices that is used to find alpha that is used to find critical points of the Bezier curve. Depending on the value of alpha, we can determine whether the curve should be smooth or a corner.

3) Using the adjusted vertices and alpha values, we can adapt to our vector type defined as (point * curve). The curve associated to the point describes the curve from this point to the next one. This declaration adapts well to the signature of curves in the SVG file.

According to 2.3.2 when we calculate the control point of the Bezier curve with respect to alpha, this calculations gives us the point in another coordinate system. To return to our coordinate system we need to find the affine map responsible for the transformation and take the inverse of that map. To do that we need to solve 2 systems of 3 equations. We do that with the Reduce method of Mathematica. Note that Mathematica returns all of the possible cases we should consider. In the cases when we do not have a unique solution for all 3 variables, the three points we are considering (x1,y1),(x2,y2) and (x3,y3) are proved to be collinear which is a case that never happens according to our implementation.



Reflection

Planning: We set concrete and reachable goals for our project and each of our milestones, and were well on track to finish our project before the deadline. We split the work into separate modules so that we were able to split into pairs to work on each of them, and that worked out well. It wasn't hard to connect each of the parts in the end. Finishing the first two parts early on gave us enough time to fix our inefficient optimizing function.

If we had more time, we would still program in ocaml but add more cool extensions like allowing for smooth shading and user interface. Some user interactions that we were thinking of adding but didn't get to were: despeckling (taking off polygons that are too small), turn policies, and general choice of constants, as suggested in the paper. In addition, it would be feasible to modify the algorithm to accept bitmaps with two colors different from black or white. By breaking these multicolor bitmaps into bi-color bitmaps, we can apply our algorithm and vectorize multicolor bitmaps.

Advice for future students

Make sure you fully understand what's needed for the implementation (choose an algorithm for which you can find really good explanations for how it works). Prepare to read and re-read papers. Sometimes the details are not explicitly stated and you would have to judge for yourself the best way to deal with corner cases.

First Spec Doc

Project Name: Vectorization

Group names and email address:

Styliani Pantela: stylianipantela@college.harvard.edu
Grace Gee: gracegee@college.harvard.edu
Victoria Gu: vgu@college.harvard.edu
Deborah Alves: dalves@college.harvard.edu

Brief Overview

Problem: To make higher resolution images from bitmaps that can be manipulated without loss

of quality.

How to address: We are going to vectorize bitmaps images using the algorithm described in: http://potrace.sourceforge.net/potrace.pdf The process involves 3 steps: 1. Constructing a directed graph from the bitmap and finding all the closed paths on the graph on paths and boundaries 2. Finding an optimal polygon from the closed paths. 3. Constructing smooth Bezier curves from the vectors.

Feature List

Core Features: Construct an algorithm based on paper mentioned above to vectorize images of 2 colors.

Add-ons: If we finish the core features, we may implement the despeckling or expand to more colors (such as on the gray scale).
**Final comment: did not get to this but did get to Bezier curves

Draft Technical Specification

We plan on using SVG, a form of mark-up language to visualize our final vectors. The documentation is listed here http://www.w3.org/TR/SVG/paths.html

The user would be only be able to retrieve the input and outputs in between the steps. Details are outlined below:

Step 1:
Construct a directed graph from the bitmap and find all the closed paths on the graph

- input: bitmap
- output: closed paths
- corners integer coordinates
- define vertices
- define edge
- path closed, length
  - find closed path with edges only once
- picking pixels of diff color
- invert all pixel colors
  - recursively go over
- turning
- despeckling
  - parameter t pixels

Modules and types:

User accessible: Bitmap, Path

- Bitmap (original bits)
- BiColor Matrix
- type vertex: (int, int)
- type BiColor= | Black | White

‘b Graph - (directed) representation of vertices with position
(data structure- possiblly adjacency matrix representation)
Path ‘a- (list of vertices* BiColor)

FCN_MATRIX(bitmap)-> ‘a matrix of bool/int of color
FCN_GRAPH(matrix)-> graph
graph: set of vertices and edges
vertex: tuples (locations of the four pixels surrounding it)
FCN_PATHCOLLECTION(graph) -> set of paths
rec g: Graph -> Pathlist
calls f
rec f: Graph -> (Path * Graph) option
paths inversion: look for other color
order the paths

Step 2: Finding an optimal polygon from the closed paths.
input: closed path
output: optimal polygon approx
straightpaths
distance
Trick: compute for every pair the constraint on future points.
find “possible segment” to make polygon
find optimal polygon by # of segments
“penalty” of each segment using standard deviation (yea easy math)
make table ahead of time, can be done in O(n)
penalty can be found in O(1) now
associate penalty with # of arrows
compare them

Modules and types:

User accessible: Path, Polygon

type Polygon
type VList

fun h: Path list -> optimal polygon list
List.map (functions composed) (path list)

fun path_making: VList -> straight path

fun optimize: polygon -> optimal polygon

fun penalty: closed path->penalty
fun max_distance: (vertex, vertex)-> float (‘b)
fun approximates: (x1,y1) (x2,y2) (list): bool
fun cyclic_difference
fun subpath: i j vlist: vlist
fun straight vlist: segment option (segment = point 1, point 2)
fun possible_segment

Step 3: Constructing smooth Bezier curves from the vectors
- input: polygon associated with closed path
  - make points a_k
    - calculate line that approximates that minimizes squares of distance to line
    - matrix math!
  - Bezier curves
- Optimize vertex adjustment to fit bitmap
- Calculate corners
  - input: adjusted polygon, threshold parameter alpha
  - get midpoints
  - determine final curve from alpha (smaller -> corners)
- Make curves
  - input: Bezier curve segments and straight lines, epsilon (tolerance)
  - can be optional since doesn't really change shape of curve

Modules and types:

User accessible: Polygon, Vector

```
type Vector = (Point * Curve) list
type Point = float * float
type Curve = | Line | Bezier of Point
```

What's next?

We plan on using OCaml for coding and SVG for drawing out the vectors.

Second Spec Doc

Project Name: Vectorization

Group names and email address:

Styliani Pantela: stylianipantela@college.harvard.edu
Grace Gee: gracegee@college.harvard.edu
Victoria Gu: vgu@college.harvard.edu
Deborah Alves: dalves@college.harvard.edu

Brief Overview

Problem: To make higher resolution images from bitmaps that can be manipulated without loss of quality.

How to address: We are going to vectorize bitmaps images using the algorithm described in: http://potrace.sourceforge.net/potrace.pdf The process involves 3 steps: 1. Constructing a directed graph from the bitmap and finding all the closed paths on the graph on paths and

boundaries 2. Finding an optimal polygon from the closed paths. 3. Constructing smooth Bezier curves from the vectors.

Feature List

Core Features: Construct an algorithm based on paper mentioned above to vectorize images of 2 colors. Our main goal is to get a bitmap input from the user and return the optimal vectorized polygons.

Add-ons:
1) Step 3 - improve the polygons by using Bezier curves
2) User interactions with the output: let the user zoom in the images to compare the bitmap and the outputted vector to compare the quality ** Final comment: we allowed user input for what to scale images to, and user could manually zoom in on the ouput images within the browser
3) Accept multicolor bitmaps - create an algorithm to break multicolor bitmaps into bicolor bitmaps to apply our algorithm and vectorize multicolor bitmaps. **Final comment: did not get to multicolor bitmaps
4) User interactions defined on the paper: dispeckling, turn policies, and general choice of constants, as suggested in the paper.
** Final comment, only user interaction right now is allowing the user to choose what factor to scale the image by

Draft Technical Specification

We plan on using SVG, a form of mark-up language to visualize our final vectors. The documentation is listed here http://www.w3.org/TR/SVG/paths.html

The user would be only be able to retrieve the input and outputs in between the steps. Details are outlined below:

Modules and Types (placed in types.ml)

```
(* General matrix module *)
module type MATRIX =
sig
 type elt
 type matrix

 val default : elt

 (* returns the dimensions of the matrix *)
 val dimensions : matrix -> (int * int)

 (* set an element of the matrix
  * returns the same matrix if the position given
  * does not fit the matrix dimensions *)
 val set : matrix -> int -> int -> elt -> matrix

 (* gets the element in the given position - consider the
```

```
  * int entries mod the dimensions *)
 val get : matrix -> int -> int -> elt

 (* initialize a matrix with the given dimensions. all
  * elements will be the default element *)
 val initialize : int -> int -> matrix

end

(* Module for Directed Graphs *)
module type DIRECTEDGRAPH =
sig

 type vertex
 type graph

 (* a graph with no vertices *)
 val empty : graph

 (* returns the out-neighborhood of a vertex *)
 val outneighbors : graph -> vertex -> vertex list

 (* returns the in-neighborhood of a vertex *)
 val inneighbors : graph -> vertex -> vertex list

 (* add a vertex into the graph - without edges *)
 val add : graph -> vertex -> graph

 (* removes a vertex (and its edges) from the graph -
  * if vertex not found returns graph unchanged *)
 val remove : graph -> vertex -> graph

 (* inserts/changes an edge in the graph, from the first to the
  * last vertex. if vertex is not found  returns the graph
  * unchanged *)
 val set_edge : graph -> vertex -> vertex -> graph

 (* removes the edge from the first to the second vertex
  * if vertex is not found, return graph unchanged *)
 val remove_edge : graph -> vertex -> vertex -> graph

 (* get a vertex from the graph *)
 val get_vertex : graph -> vertex

end


(* BiColor type for binary pixels *)
type biColor = Black | White ;;

(* Paths declaration *)
```

```
 type 'a vertex = ('a * 'a);;
 type 'a path = ('a vertex list * biColor);;

(* types for the vector display *)
type point = float * float ;;
type curve = Line | Bezier of point ;;
type vector = (point * curve) list ;;
```

Step 1:

1) Get a bitmap as an input
2) Read the bitmap and create a biColor matrix to read the pixels
3) Use the matrix from the bitmap to construct a directed graph, as described in the paper
4) Use the graph to extract the different closed paths and their colors, and return a list of paths that will be used in the next step

Implementation:
- Read the bitmap bytes to get dimensions of the bitmap and initialize biColor matrix
- Read the value (Black or White) of each pixel, from the bitmap bytes, and insert in the matrix
- Use the bitmap to construct the graph: iterate through the vertices (possible corners of pixels with 4 adjacent pixels not all of the same color), add them to our graph and connect with the edges (direction of the edge is determined by the position of the colors, as described in the paper)
- Use the graph to get the list of paths. First, get the left most top post vertex (using the right implementation of get_vertex). Follow the arrows (edges) from this vertex, until we get back to the first vertex. Extract this path from the graph (by extracting the edges, and then the vertices without edges) and "save" it in the path list. Repeat the process until the graph is empty.

Step 2:

1) Get a path (vertex list) as an input
2) Read in the vertices and find “straight paths” by finding the “direction” and computing for every pair the constraint on future points.
3) Use the “straight paths” and by minimizing penalties, we find the optimal polygon structured as a list of vertices.

Implementation:
- Read in a closed path represented by a vertex list and compute the direction for each pair (North, South, East, West).
- For each pair, compute the constraints on the next pixel so that each pixel will need to satisfy a “three-way” constraint. This will give us a vertex list representing the straight path.
- Use the “straight paths” and penalties algorithm to find the optimal polygon for the given list of vertices. The penalties algorithm utilizes the standard deviation of the Euclidean distance between vertices, cyclic difference, and makes sure that the polygon with the least number of sides is chosen.

```
Module type POLYGON =
sig
```

```
        type Polygon (path list)
        type direction= South | North | East | West
        val find_direction -> vertex -> vertex -> direction
        val direction -> vertex list -> direction list
        val total -> vertex list -> vertex list
        val straight -> vertex list -> bool
        val optimize -> vertex list -> vertex list
        val penalty -> vertex list -> ‘a
        val max_distance -> vertex -> vertex -> ‘a
        val approximates -> vertex -> vertex list -> bool
        val path_making -> vertex list -> vertex list
        val cyclic_difference -> vertex -> vertex -> ‘a
end
```

Partial functions for Part 2

```
let find_direction: (v1 : vertex) (v2: vertex) : direction =
        let ((a, b), (c, d)) = (v1, v2) in
        match (c-a), (d-b) with
        | (0, 1) -> North
        | (1, 0) -> East
        | (0, -1) -> South
        | (-1, 0) -> West

let direction: (vlist: vertex list) : direction list =
        let vlist2= vlist @ (List.hd vlist) in
        let rec ind (sub: vertex list) : vertex list =
                match (List.tail sub) with
                | [] -> []
                | hd2::tl -> (find_direction (List.hd sub) hd2) :: ind tl
        in ind vlist

let total (lst: vertex list): (vertex list) =
        List.map (functions composed) lst

let straight (v: vertex list) : bool =

let path_making (vertex list): (vertex list) =

let optimize: polygon -> optimal polygon

let penalty (vertex list): ‘a =

let max_distance (v1: vertex) (v2: vertex): ‘a
        match (v1, v2) with
        |((a,b),(c,d)) -> max (c-a) (d-b)

let approximates (v: vertex) (lst: vertex list): bool

let cyclic_difference (vertex1) (vertex2): ’a =
```

Step 3:

1) Get the midpoints of the polygon - the endpoints of the curve
2) Decide the parameters based on the algorithm described in the paper
3) Adapt to the vector type
4) Curve optimization

Implementation:
- We will implement a function that gets 3 points a, b and c (as the points b_(i-1), a_i and b_i from the paper), to determine the paramater alpha, just like in the paper.
- Using the parameter, we will change the polygon to a vector, to match our type signature. Our signature is a list of pairs (point * curve). The curve associated to the point describes the curve from this point to the next one. This declaration adapts well to the signature of curves in the SVG file.

What's next?

Checkpoint - Sunday 4/15
Alpha:
Deborah and Stella: Implementation of Step 1 and the display to show the functionality
Grace and Victoria: Implementation of Step 2

Checkpoint - Sunday 4/22
Beta:
Implementation of Step 3, with the displays to accept user input and show the output

Checkpoint - Sunday 4/29
Finished project:
Video of a presentation and possible add-ons

Repository

We will be using bitbucket to share code.

CS51 Project Checkpoint 1

Progress
Deborah and Stella worked on taking a bitmap, converting it into a directed graph, and then outputting the closed paths. Grace and Victoria worked on taking the closed paths and generating the optimal polygon that fits the paths (please refer to step2.ml).

Part 1: We have implemented functions for taking a bitmap and converting it to a matrix (that stores the coordinates and colors of the pixels in the bitmap), taking a matrix and outputting a directed graph and taking a graph and figuring out all the closed paths of the graph.

Part 2: We have implemented and tested all our functions in ocaml for generating possible segments, calculating the penalty for each segment and the whole polygon, generating the

polygon, and filtering to find the optimal polygon. We wrote many tests for all of our functions along the way and spent a long time debugging, and now all our asserts pass.

Problems

1. We ran out of time while testing how to read a bitmap, but it is about to work.

2. All asserts pass and given a vertex list of a closed path we can find the optimal approximating polygon.

Teamwork

We divided into pairs to work on the first and second parts of our project. We are good. :)

Plan
We plan on taking a look at the possible extensions that we can implement next week:

1) Adjust the endpoints of the segments.

2) Possibly implementing Bezier curves.

3) Accept multicolor bitmaps - create an algorithm to break multicolor bitmaps into bicolor bitmaps to apply our algorithm and vectorize multicolor bitmaps.
**Final comment: did not get to this but did get to implement Bezier curves
4) User interactions defined on the paper: despeckling, turn policies, and general choice of constants, as suggested in the paper. Let the user zoom in the images to compare the bitmap and the outputted vector to compare the quality
**Final comment: did not get to this but did get to implement Bezier curves

Checkpoint 2

- **Progress:** Describe what you have done over the past week. What is done? Does it always work? Is it still buggy?

  The first two parts are complete and working. We have started implementing the third part and a more efficient implementation of part 2 and will finish it through the week.
- **Problems:** What has given you trouble? Your TF may be able to help you overcome conceptual confusion or technical difficulties.

  Step3: In the part 2.3.1 of the algorithm, we are calculating the center of gravity that will be one point of the line and then the we find the eigenvector corresponding to the biggest eigenvalue of the 2x2 matrix. We are not sure what eigenvector to get if the matrix is the identity, which means that we only have one eigenvalue and many eigenvectors. Which eigenvector should we choose then?
- **Teamwork:** How have you organized working together on a larger project? What problems have arisen, how have you dealt with these problems, and how can

you improve moving forwards?
Stella and Grace working on Bezier curves
Deborah and Victoria finish up Dijkstra's Algorithm

- **Plan:** What things do you need to do next week to finish? Include a schedule specifying what needs to get done, when it will be finished, and who will be doing it.
  Finish up on Bezier curves and Dijkstra's Algorithm