

Recognizing trusses

Project 2 from 624 Sketch Recognition course



November 5, 2017

Yan Bai, David Cross, Xien Thomas

**Project 2 Report**

**Team Members: Yan Bai, David Cross, Xien Thomas**

**Date: 11/5/2017**

**I. Algorithm**

Our algorithm was made up of several different smaller functions. Essentially, the idea of the algorithm was to take in the sketch, get the corners from the sketch using the iStraw algorithm, use these corners to break the sketch up into individual shapes (i.e. any group of connected strokes), evaluate each of these shapes to see how many triangles they contained, and detect a truss based on this information. I will cover each of these steps in more detail throughout this report, but this is the high-level overview. I will not detail the inner workings of iStraw, as there has been an entire paper written just about that algorithm itself and the team did not change the algorithm at all, so it is no different than the one mentioned in the paper on iStraw.

In the first part of our algorithm, we take in a sketch as a JSON object. This sketch is made up of points that are listed in groups by what stroke they belong to. Our algorithm takes all the information from the sketch and feeds it to iStraw, which we will treat as a black box here, which then produces a series of indices of corners. These indices indicated which point in the object is a corner. We then take these indices and create a list of points that are corners. Each point in this list has an x value, a y value, and a time value. For the purposes of this project, the time value was not used, but the team kept it in the point data structure for the sake of preserving the data structure given to us. We then proceed to feed these corners into the “Shape Separator” algorithm which, as its name implies, separates out which corners belong to which shape. Again, as mentioned before, here a shape is defined as a group of connected strokes.

The pseudocode for the shape separator is below:

//Function to separate out shapes within a sketch

function shapeSeparate(corners, allPoints)

{

threshold = 1;

var shapesList = [];

//if there's no line, we want to run the corner algo again but with the offending corner removed

//console.log(val);

//go through every corner in the sketch

for(var i = 0; i < corners.length; i++)

{

//Get corner you're working with

workingCorner = corners[i];

closestConnectedCorner = nearCorner(corners, workingCorner, allPoints, threshold);

alreadyExists = false;

for(n in shapesList)

{

for(point in shapesList[n])

{

xtest = shapesList[n][point].x;

ytest = shapesList[n][point].y;

//If point A is already in a shape, we need only add point B

if(shapesList[n][point].x == workingCorner.x && shapesList[n][point].y == workingCorner.y)

{

shapesList[n].push(closestConnectedCorner);

alreadyExists = true;

}

else if(shapesList[n][point].x == closestConnectedCorner.x

&& shapesList[n][point].y == closestConnectedCorner.y)

{

shapesList[n].push(workingCorner);

alreadyExists = true;

}

}

}

if(alreadyExists == false)

{

newShape = [];

newShape.push(workingCorner);

newShape.push(closestConnectedCorner);

shapesList.push(newShape);

//console.log(newShape);

}

}

return shapesList;

}

Because the pseudocode on its own is rather confusing, I will explain how this part of the algorithm works. The shape separation algorithm first opens a loop through every corner in the sketch, obtained from the iStraw algorithm. For each corner, the algorithm saves the value of the corner it is currently on, and then proceeds to find the nearest connected corner by calling the “nearCorner” function. This function’s pseudocode is below:

function nearCorner(corners, workingCorner, allPoints, threhold){

bottomDist = 0;

val = false;

while(val == false)

{

//Reset Distance

nearestDist = 9999;

//reset corner (why reset corner using the same corner?)

var nearestCorner = new node();

//Find all corners except the one you're using

for(var j = 0; j < corners.length; j++)

{

//Set current corner

currentCorner = corners[j];

//Make sure current corner isn't the one you're measuring from

if(currentCorner.x == workingCorner.x && currentCorner.y == workingCorner.y)

{

continue;

}

//Find euclidean distance between corner you're working with and

//the current corner

currentDist = euclideanDist(workingCorner, currentCorner);

if(bottomDist < currentDist && currentDist < nearestDist)

{

nearestDist = currentDist;

nearestCorner = currentCorner;

}

}

val = checkForLine(allPoints,workingCorner,nearestCorner,threshold);

if(val==false)

{

//val = true;

bottomDist=nearestDist+1;

}

else{

return nearestCorner;

}

}

}

Before we get into this function, it should be noted that the functions for Euclidean distance, checking for lines, and midpoints are all helper functions the team designed for this project. The “nearCorner” function essentially looks for the closest connected corner between the corner we are currently working with in the shape separation algorithm and the corner that is nearest to that one. To do this, we first loop through all the corners in the list of corners except for the corner we are working with and find out what the nearest corner is, storing the Euclidean distance to that corner. After finding this information, the function calls the “checkForLine” function which, as its name implies, checks if there is a line between two points. In this function, we find the midpoint between the two points, and then the midpoint between the midpoint and the first point and the midpoint between the midpoint and the second point (referred to in this paper as the “quarter point” and the “three quarter point”). We did this because we wanted to make sure that the line that existed between the two points was continuous. Using this information, we used a threshold value that is set across all functions to determine what the radius of the circle around the quarter point and three quarters point we would use to look for points should be. Essentially, the function looks in the general area of the quarter point (the exact size of this area defined by the threshold value) to see if there exists a point in the points list in the area. If there does, we have at least half a connecting line. We repeat this process for the three quarter point to check for the other half of the line. If both processes return true, then we know that there exists a line between the two points we used.

Coming back up to the shape separation function, we return true if there exists a line between the two corners and false if there doesn’t. We do all this to prevent the case where a corner from another shape is closer to the corner we’re working with than any corners in the shape. If there exists a line between these two corners, then we have found the closest connected corner and we return this point from the function. If there is no line between the closest corners, then we start the algorithm over again, but this time with a minimum distance required between the corners, which is the Euclidean distance between the corner we’re working with and its nearest neighbor plus one unit. By doing this, we make sure that the algorithm does not get stuck in a loop of finding the closest unconnected corner. We continue with this process, increasing the minimum distance until we find the closest connected corner. We then check each shape in the current list of shapes to see if either corner appears in any of the shapes in the shapes list. The idea being that if the first half of a pair of connected corners appears in a shape, then logically, the other half of that pair should be placed in that shape. We then do the same thing but inverted, checking if the second half of the pair appears in a shape. If it does, we add the first half to that shape. If neither point appears in any shapes in the shapelist, we add a new shape and add both points to it. Finally, we repeat this for every corner in the corner list, creating a list of shapes that represents the different groups of connected strokes in the sketch.

Taking our newly formed list of shapes, we can now detect triangles. The code for this segment of the algorithm is as follows:

function findTriangle(corners,allPoints,shapesList)

{

triangleCornerList=[];

for(shape in shapesList)

{

triangleCount = 0;

triangleList = [];

for (var i = 0; i < shapesList[shape].length; i++)

{

//Get corner you are working with

var workingCorner = shapesList[shape][i];

var shortestDist = 9999;

var secondShortestDist = 9999;

var nearestCorner = new node();

var secondNearestCorner = new node();

for (var j = 0; j < shapesList[shape].length; j++)

{

if (shapesList[shape][j].x == workingCorner.x && shapesList[shape][j].y == workingCorner.y)

{

continue;

}

currentDist = euclideanDist(workingCorner, shapesList[shape][j]);

if (currentDist < shortestDist)

{

shortestDist = currentDist;

nearestCorner = shapesList[shape][j];

}

else if (currentDist < secondShortestDist)

{

secondShortestDist = currentDist;

secondNearestCorner = shapesList[shape][j];

}

}

lineAB = checkForLine(allPoints,workingCorner,nearestCorner);

lineAC = checkForLine(allPoints,workingCorner,secondNearestCorner);

lineBC = checkForLine(allPoints,nearestCorner,secondNearestCorner);

if(lineAB == true && lineAC == true && lineBC == true)

{

triangle = [];

triangle.push(workingCorner);

triangle.push(nearestCorner);

triangle.push(secondNearestCorner);

triangleList.push(triangle);

triangleCount++;

}

}

if(triangleCount >= 2)

{

for (item in triangleList)

{

for (cornerPoint in item)

{

triangleCornerList.push(cornerPoint);

}

}

}

}

return triangleCornerList;

}

In this section of the algorithm, we aim to determine how many triangles are present in each shape. We do this by using an algorithm that is similarly structured to the one we used to separate out shapes. First, we loop through every shape in a sketch. For every shape, we loop through every corner tied to that shape. For every corner, we find the two closest connected corners using the same method we did for the shape separator. By doing this, we now have the Euclidean distance to each of these points from the starting point and the x and y values for each of these closest points. We also know, from using our line checking algorithm, that these points are connected to the original point. Therefore, the only other thing required to prove that there exists a triangle between these three points is a connection between the two closest points that is independent of the origin point. Thus, we again use our line checking algorithm to check if there is a line between these two points. If there is, we have a triangle and we add it to the list of triangles (a list that is similar to the shape list, but each triangle only has three corners). We also increment the number of triangles in this shape by 1. At the end of the shape, if there are two or more triangles present in the shape we assume it is a truss. This is because in the cases that our data is limited to, the vast majority of cases where there are 2 triangles or more indicates a truss. There are a few exceptions, but they are so few that we decided it was not worth it to use programming power to deal with them. One solution we discussed is featured in the “Other Routes Explored” section below.

Now that we have a list of shapes that contain trusses and those that don’t, we just have to figure out which strokes in the original truss are a part of the shapes that contain trusses. If a stroke contains even a single corner that is present in a shape that contains a truss, then logically that stroke is used to make a truss. Therefore, the last step in our algorithm is loop through all the strokes and each point in the stroke and see if any of them match the corners that are in shapes that contain trusses. If any point matches, the stroke is added to the final list of strokes that contain trusses. If there are no strokes in this list at the end of the algorithm, then the sketch contains no trusses. If there are, then we return that the sketch contains trusses and which strokes are used in these trusses.

**II. Preprocessing**

We did not perform any custom preprocessing for this project, other than the preprocessing that occurs within the iStraw algorithm, which resamples points to be a uniform distance away from each other. We used this resampled set of points for the rest of the program.

**III. Recognition Results**

None

**IV. Other Routes Explored / Future Work**

We have explored other routes of the problem of facing trusses. One main route we explored is the A to ^C to B to D to A approach. It is very confusing but the general idea is that if I can find the highest degree (number of connected neighbors), and use this point as a midpoint between two other points. If the shape is a truss then there should exist another path from that is not associated with that midpoint connecting the two points. This is a great truss finding algorithm we were working on until we were given the definition of a truss. We generally was looking at trusses from anything but we are given mechanix truss. Even though we could still use the complexity was really high and could be viewed again for further evaluation. One example of not using this algorithm is a that mechanix trusses do not have empty wedges. The algorithm would not helpful in trusses like these:

But the algorithm would be helpful in shapes with wedges:

Where the green circle represent the node with the highest degree

The pseudocode for this algorithm takes some trial and error because corners first to be established using either shortstraw or Istraw. In our implementation we used Istraw. With the given corners, more processing is needed to determine if the corners are in the same area; it is usually accomplish by going through each node (workingNode) and threshold the working node to check if other nodes are in the same range. Once the node is recognized then replace this node with the working node position.

Here is the pseudo code for recognizing node with the highest point:

function findHighNearCount(fnodes){

var high = 0;

for(var n in fnodes){

if(fnodes[n].near.length > high){

high = fnodes[n].near.length;

node=fnodes[n];

}

}

return node;

}

It assume the highest degree of a node is zero and go through each node and replace the index it come by and use this node as the node with the highest degree. Notice that this program does not check for multiple high nodes but check for a singular one.

//find path exist

function findPath(nodeList, c){

a = c.near[0];

length = c.near.length;

if(length == 1){

return false;

}

var n = 0;

while(true){

if((c.near[n].x != a.x) && (c.near[n].y != a.y)){

continue;

}

else{

var k=0;

while(true){

if((c.near[n].near[k].x == a.x) && (c.near[n].near[k].y == a.y)){

return true;

}

else if(c.near[n].near.length == k){

return false;

}

k++;

}

}

n++;

}

}

The pseudo code above is the main algorithm that calls function that have been explained before in previous pseudo codes. While going through each node we try to find a path that is connected to the highest point. Since it was established that there is a line between the two nodes we continue. Once we are on the highest degree node we then go through each node except the starting node to see if there is a line that goes directly towards the start and skip the high degree node. We would only worry about passing three sides because originally we thought a truss only have triangles in them.

This concludes the algorithm, it would stop short because fitting our algorithm to our definition of mechanix trusses does not work. One problem is that we cannot assume that trusses are a bunch of triangles put together. There are trusses that consists of trusses that have a square with a triangle support the left side and the right side of the square. In this case the algorithm would not be enough to support that the figure is actually a truss. Foundationally, this is a good algorithm. The idea of finding the highest degree point and create a path from the start to the sink and fining another back to start is good foundation for recognizing trusses.

V. Individual Contributions

**Yan Bai –** General brainstorming of solutions, initial coding of triangle finding algorithm, debugging

**Xien Thomas –** General brainstorming of solutions, debugging, iStraw set up, test server set up, second path algorithm(Furtue works), report writing (future works).

**David Cross –** General brainstorming of solutions, coding of shape separation algorithm, debugging, report writing

All three team members met for about 95% of sessions, so we all kind of jointly worked out solutions to the problem at hand, hence the general brainstorming statement.

All files are pushed through Github: https://github.com/davidpcross17/SketchRec2017