Final Project - Symmetric Clock Tree

by

Marc Brown

University of Michigan-Dearborn

Instructor: Dr. Riadul Islam

Honor Code:

I have neither given nor received unauthorized assistance on this graded report.

X____

Department of Electrical and Computer Engineering
College of Engineering and Computer Science
The University of Michigan-Dearborn

Abstract

For my final project, I explored buffered clock tree construction through simulating the performance of skew minimization by structural optimization. Using the Python Programming language, I'll explored a specific type of clock-tree structure, called *symmetrical structure*. At each level of a symmetric clock tree, the number of branches, wire-length, and inserted buffers are almost the same. The deliverable for this project will be an algorithm that, when given a list of sink location, generate a symmetric clock tree. This project is based on the research done by Xin-Wei Shih and Yao-Wen Chang in their paper, Fast Timing-Model Independent Buffered Clock-Tree Synthesis.

Department of Electrical and Computer Engineering
College of Engineering and Computer Science
The University of Michigan-Dearborn

Final Project

Explain the procedure of what you did during Lab. Just one paragraph may be enough.

++

Part 1)

The first step in my project was to go through the research, and convert their white paper into a technical specification outlining the project. Since I chose Python as my programming language, I began by first determining the classes that needed to be implemented in order to create a symmetric clock tree. The hierarchy goes as follows:

- Sinks (stored as x,y coordinates) are added to SinkGroups
- SinkGroups have a central centroid (stored as x,y coordinates)
- Our SymmetricClockTree has Sinks, Centroids, and Sink Groups.

Once I'd determine the high level class structure, I had to determine what properties each of these objects needed. The class models can be found below:

<u>Sink</u>

- init(self, x,y)
- setLevel(self, level)
- getLevel(self)

<u>Centroid:</u>

- init(self, x, y)

SymmetricClockTree:

- num_sinks: total number of sinks in the tree
- sinks: Array of Sink Objects
- centroids: Array of Centroid Objects
- sinksGroups: Array of SinkGroup Objects
- standardizedWireLength: length of longest wire in tree.
- cluster_size: defined size of our kmeans cluster.

Part 2)

Once I'd designed my class structure, the next step was to create an algorithm that would result in the desired outcome. A high level overview of my algorithm is as follows:

1. Read in sink locations from file.
    a. For each sink location, create a Sink object.
2. Group Sinks into SinkGroups
    a. I knew I was going to use K-Means clustering to group the sinks together, but I wanted to find a method that would help me find the optimal number of clusters given the number of sinks in the tree. The method I tried was silhouette analysis. This method tries many different tree cluster size configurations, and computes a silhouette average between -1 and 1. This silhouette average represents how close any given node in a cluster is to being apart of a different cluster. The value of this algorithm is that it allows you to find an optimal cluster size solely based on the number of nodes in the cluster. To determine the clock size configurations, I

found all factors of the number of sinks in the tree, and use these as the test clock sizes.

b. Once I'd determined the optimal cluster size for KMeans, I perform KMeans clustering.

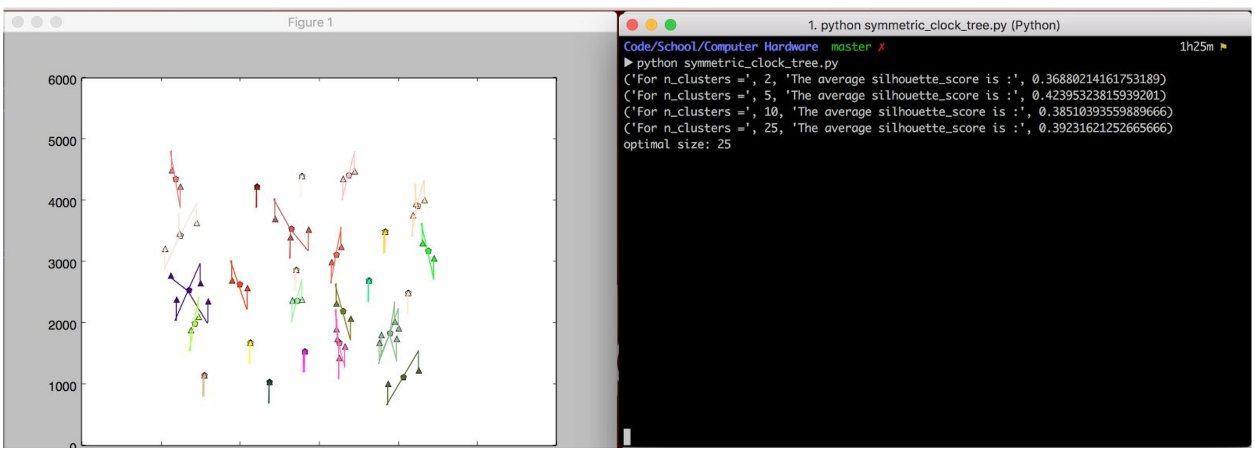
c. Determine each cluster's center location. Turn these into your Centroid locations, and create a Sink Group for each Centroid.

d. Once you have a Sink Group, add Sinks to their correct Sink Group.

e. As you add Sinks to their Group, compute the distance between that sink, and it's center, and store the longest distance.

```
def groupSinks(self):
    # Get all sinks locations
    sinks = self.getSinkLocations()
    # determine best cluster size
    num_clusters = self.determineNumberOfClusters()
    # Perform KMeans clustering and identify cluster centers.
    kmeans = KMeans(n_clusters=num_clusters).fit(sinks)
    centroids = kmeans.cluster_centers_

    # For each identified cluster center:
    # Create a new Centroid
    # Add Centroid to our tree
    # Create a new SinkGroup
    for i, location in enumerate(centroids):
        centroid = Centroid(location[0], location[1])
        self.addCentroid(centroid, i)
        self.createSinkGroup(centroid, i)

    # Once we've made a SinkGroup for each cluster center
    # We'll use the list of groupings, and add each Sink to their designated group.
    self.addSinksToGroups(kmeans.labels_)
    print "required wire length:", self.standardizedWireLength
```

```python
def determineNumberOfClusters(self):
    range_n_clusters = self.findAllFactors()
    scores = {}
    sink_locations = self.getSinkLocations()

    for n_clusters in range_n_clusters:
        # Initalize the clusterer with n_clusters value
        # and a random generator seed of 10 for reproducibility.
        clusterer = KMeans(n_clusters=n_clusters, random_state=10)
        cluster_labels = clusterer.fit_predict(sink_locations)

        # The silhouette_score gives the average value for all the samples.
        # This gives a perspective into the density and separation of the formed
        # clusters

        silhouette_avg = silhouette_score(sink_locations, cluster_labels)

        scores[n_clusters] = silhouette_avg

    # Get cluster size that's closest to the median of all the silhouette average.
    averages = list(scores.values())
    median = statistics.median(averages)
    closest_value_to_median = min(averages, key=lambda x:abs(x-median))
```



Part 3)

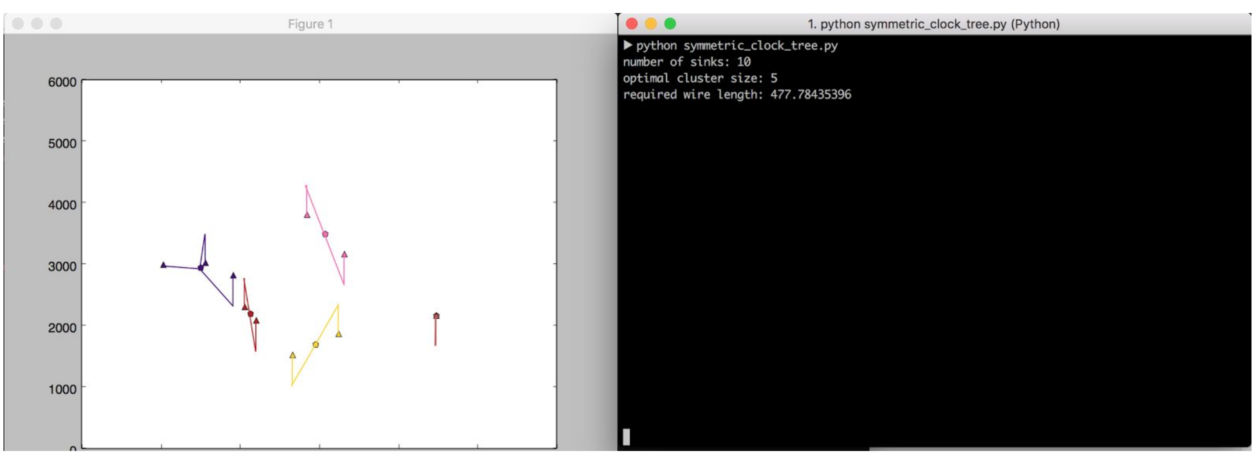
Once I'd created all the SinkGroups, the final step was to draw the plot of our SinkGroups, taking into account the required wire length. In order to ensure all Sink had the same clock delay, we had to make sure the wire lengths were equal. This means that all wires had to be the same length, as the wire that was the longest distance from their centroid. Draw the Sinks and

Centroids was easy, drawing the unique wire lengths was more difficult, since I had to draw dynamic line segments, instead of a direct line between two points. To draw these line segments:

1. I determined which direction the centroid was relative to the current sink.
2. Determined which direction the sink had the most clearance on.
3. Keep track of which direction we're traveling, and how much further we have left to travel.
4. If we reach a boundary, turn, and start at step one.

```
def drawSnakeLine(self, start, goal, goal_wire_length, group):

    bounding_thresholds = {
        "down": self.minXCoordinate,
        "up": 3*self.maxYCoordinate/2,
        "left": self.minXCoordinate,
        "right": 3*self.maxXCoordinate/2
    }

    pen = start[:]
    last_pen_position = pen[:]

    direction_traveled = { "up": False, "down": False, "left": False, "right": False }

    # turning tracks which axis we're traveling on.
    turning = False

    continue_drawing_line = True

    # booleans that track which point we're going to or coming from.
    connect_to_start = True
    connect_to_goal = False

    # Initialize our wire length
    current_wire_length = 0

    while continue_drawing_line:
        # get relative direction
        direction = self.getCurrentLocationRelativeToCentroid(start, goal, turning)

        # initialize recalculated_direction
        recalculated_direction = direction

        # determine the best direction to begin drawing
        direction_to_travel = self.getNextDirectionToTravel(start, goal, turning)

        # movingInPositiveDirection tracks which axis we're moving in
        movingInPositiveDirection = False if (direction_to_travel == "left" or direction_to_travel == "down") else True
        # movingHorizontally tracks which direction we're moving in
        movingHorizontally = False if (direction_to_travel == "up" or direction_to_travel == "down") else True

        # index tracks which pen position we're changing (x or y)
        index = 0 if movingHorizontally else 1

        # change sets the amount we increment / decrement each time.
        change = 5 if movingInPositiveDirection else -5

        # overshoot_counter tracks how many times we've crossed from the pos -> neg axis, relative to the goal location.
        overshoot_counter = 0

        # we allow ourselves to cross the axis 5 times
```

```python
while overshoot_counter < 5:
    # check if we've hit the graph bounds.
    # if we have, we'll trip a boolean so we don't keep trying the same direction
    if movingInPositiveDirection and pen[index] + change > bounding_thresholds[direction_to_travel]:
        self.directionTraveled[direction] = True
        break
    elif not movingInPositiveDirection and pen[index] + change < bounding_thresholds[direction_to_travel]:
        self.directionTraveled[direction] = True
        break

    # check to see if we've passed the goal on this axis.
    elif recalculated_direction != direction:
        overshoot_counter += 1

    # update the pen location and current_wire_length
    pen[index] += change
    current_wire_length += abs(change)

    # check if we've reached our goal wire length
    if current_wire_length >= goal_wire_length:
        connect_to_goal = True
        break

    # check our pen's location relative to the centroid
    recalculated_direction = self.getCurrentLocationRelativeToCentroid(pen, goal, turning)

# Check what type of point we're connecting
if connect_to_start:
    self.drawLineBetweenTwoPoints(start, pen, group)
    last_pen_position = pen[:]
    connect_to_start = False

elif connect_to_goal:
    self.drawLineBetweenTwoPoints(pen, goal, group)
    continue_drawing_line = False

else:
    self.drawLineBetweenTwoPoints(last_pen_position, pen, group)
    last_pen_position = pen[:]

# toggle our turning variable
turning = not turning
self.resetDirectionTravel()
```



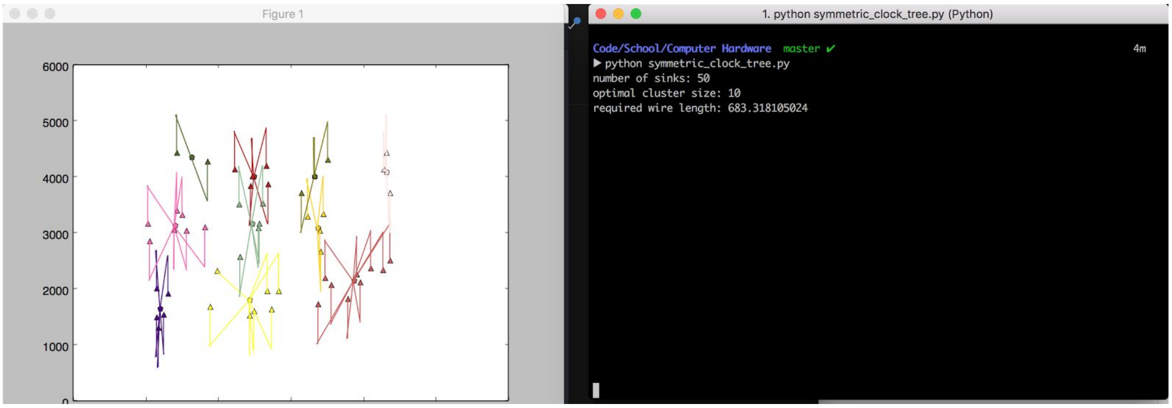


Figure 1
1. python symmetric_clock_tree.py (Python)
Code/School/Computer Hardware master ✔
4m
▶ python symmetric_clock_tree.py
number of sinks: 50
optimal cluster size: 10
required wire length: 683.318105024
6000
5000
4000
3000
2000
1000

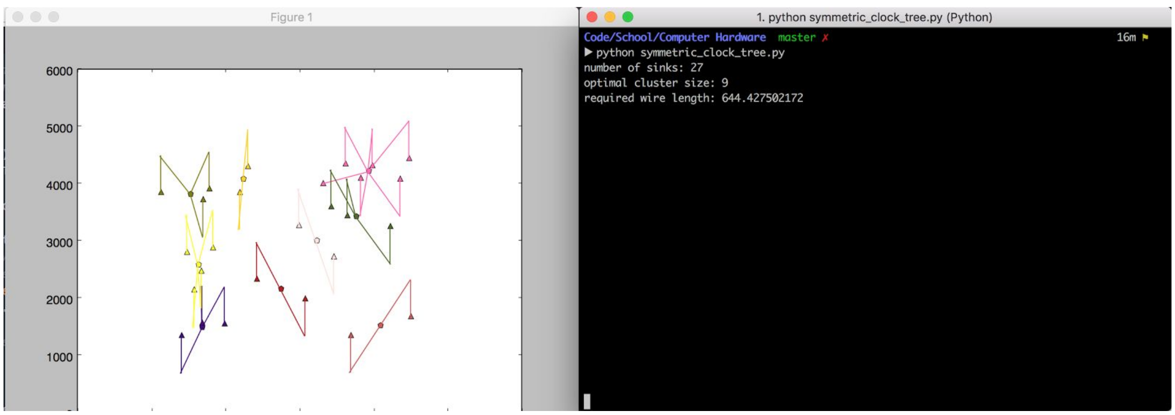


Figure 1
1. python symmetric_clock_tree.py (Python)
Code/School/Computer Hardware master ✗
16m
▶ python symmetric_clock_tree.py
number of sinks: 27
optimal cluster size: 9
required wire length: 644.427502172
6000
5000
4000
3000
2000
1000

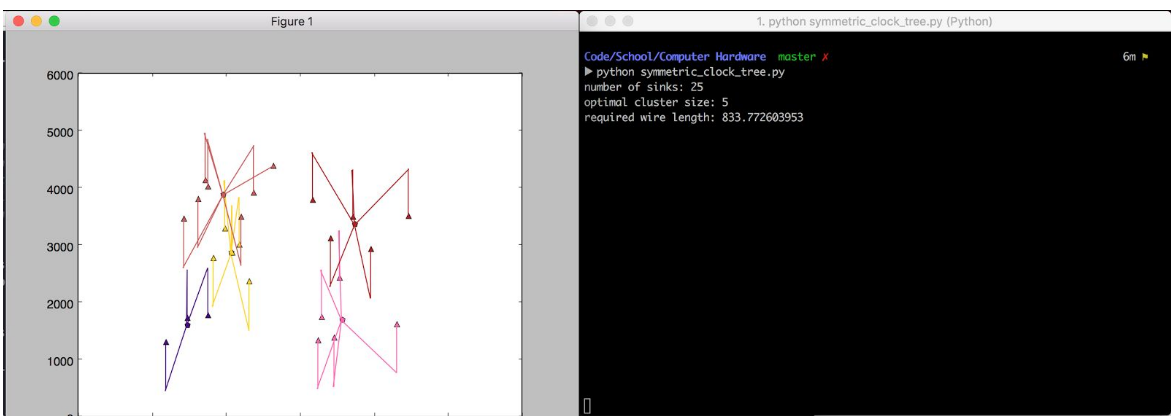
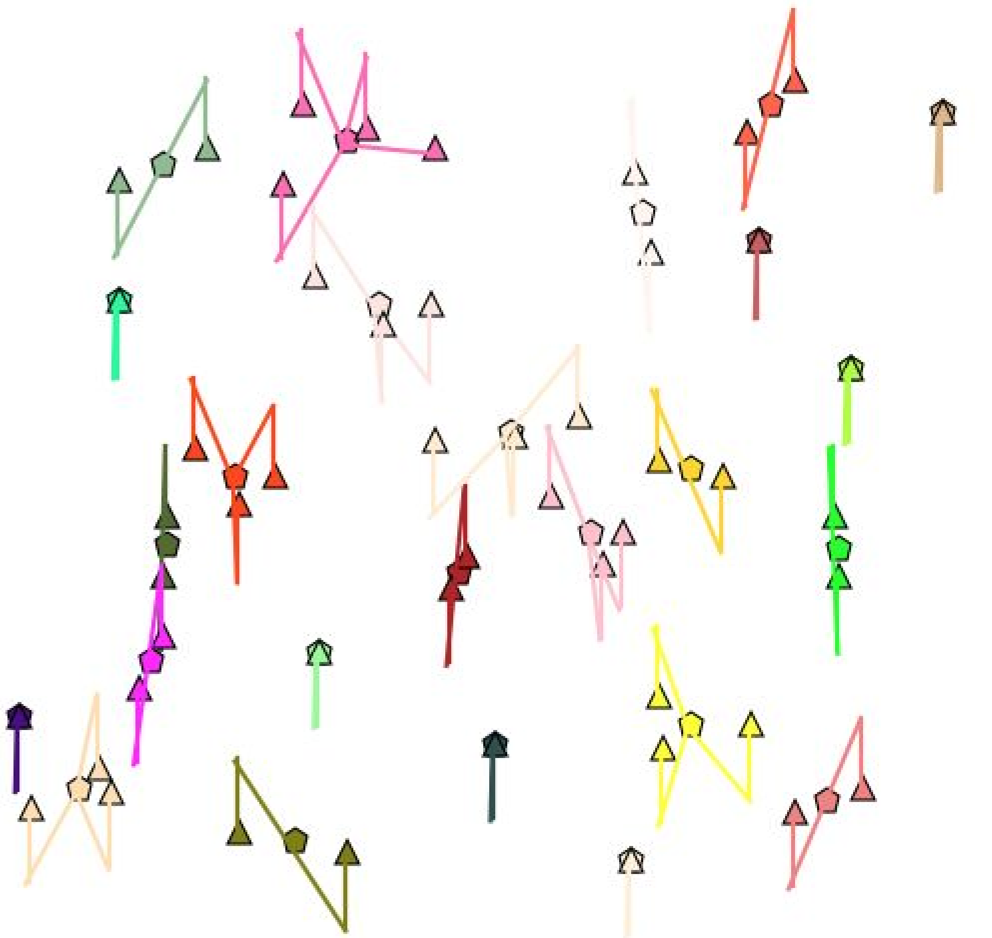


Figure 1
1. python symmetric_clock_tree.py (Python)
Code/School/Computer Hardware master ✗
6m
▶ python symmetric_clock_tree.py
number of sinks: 25
optimal cluster size: 5
required wire length: 833.772603953
6000
5000
4000
3000
2000
1000



Conclusion

In conclusion, I was able to implement a Symmetric clock tree using only sink locations and python. While I do have a working solution, I would update my clustering method. I found that silhouette analysis, while helpful in determining the number of clusters to create, was not as efficient as other methods of grouping the sinks together, in that it used more wire than other algorithms would have. Moving forward, I'm going to try different clustering algorithms, in order to find an optimal solution.

Code

```
import random

from sklearn.cluster import KMeans

from sklearn.metrics import silhouette_samples, silhouette_score

import numpy as np

import matplotlib

from matplotlib import pyplot as plt

import math

import statistics


# Used to represent the Sinks on our board

class Sink:

    # Each sink is initialized with an x y coordinate

    def __init__(self, x, y):

        self.location = [x, y]


    # Setter method for the level of the tree this sink is on.

    def setLevel(self, level):

        self.level = level


    # Getter method for the level of the tree this sink is on.

    def getLevel(self):
```

```
        return self.level


class Centroid:


    # Every Centroid is initialized with an x y coordinate.
    def __init__(self, x, y):
        self.location = [x, y]


# Represents the traits of our plot.
# Each element in the arrays correspond to a SinkGroup.
class PlotAttributes:
    def __init__(self, n):
        # matplot markers:
https://matplotlib.org/api/markers_api.html
        self.references = n
        self.colorList = self.generateColorList()
        self.CentroidAttributes =
self.generateCentroidAttributes()
        self.SinkAttributes = self.generateSinkAttributes()
        self.LineAttributes = self.generateLineAttributes()


    def generateColorList(self):
```

```
        colors = []

        n = self.references

        counter = 0


        for name, val in matplotlib.colors.cnames.items():

            if counter == n:

                break

            colors.append(name)

            counter += 1


        return colors


    def generateCentroidAttributes(self):

        colors = self.colorList

        attrs = []


        for color in colors:

            attrs.append( {"color": "{}".format(color),
"marker": "p"})


        return attrs
```

```
    def generateSinkAttributes(self):
        colors = self.colorList
        attrs = []

        for color in colors:
            attrs.append({"color": "{}".format(color), "marker":
"^"})

        return attrs

    def generateLineAttributes(self):
        colors = self.colorList
        attrs = []

        for color in colors:
            attrs.append({"color": "{}".format(color), "marker":
"-"})

        return attrs

    def getCentroidAttributes(self, id):
        return self.CentroidAttributes[id]
```

```
    def getSinkAttributes(self, id):
        return self.SinkAttributes[id]

    def getLineAttributes(self, id):
        return self.LineAttributes[id]


# This class represents our SymmetricClockTree, and holds the
majority of our logic.
class SymmetricClockTree:

    # Each tree is initalized with the number of sinks, and the
number of levels
    # Can't handle multiple levels just yet, so we default to
one
    def __init__(self, num_sinks, num_levels=1):
        self.num_sinks = num_sinks
        self.sinks = []
        self.centroids = []
        self.sinkGroups = {}
        self.minXCoordinate = 0
```

```
        self.maxXCoordinate = 5000
        self.minYCoordinate = 0
        self.maxYCoordinate = 5000
        self.standardizedWireLength = 0
        self.directionTraveled = { "up": False, "down": False,
"left": False, "right": False }
        self.cluster_size = -1

    def setClusterSize(self, n):
        self.cluster_size = n

    # Helper method to calculate the distance between two
points.
    def DistanceToGoal(self, start, goal):
        return math.sqrt(
            ((goal[0] - start[0]) ** 2) +
            ((goal[1] - start[1]) ** 2)
        )

    # reset's our directionTraveled Dictionary
    def resetDirectionTravel(self):
```

```
        self.directionTraveled = { "up": False, "down": False, 
"left": False, "right": False }

    # getter method for directionTraveledexa
    def getDirectionTraveled(self):
        return self.directionTraveled

    # method that allows you to input a list of sink locations.
    # expects list[[x, y]]
    def addSinksByLocation(self, sink_locations):
        self.minXCoordinate = 0
        self.maxXCoordinate = 0
        self.minYCoordinate = 0
        self.maxYCoordinate = 0

        for location in sink_locations:
            self.sinks.append(Sink(location[0], location[1]))

            # update horizontal plot constraints
            if location[0] < minXCoordinate:
                minXCoordinate = location[0]
            elif location[0] > maxXCoordinate:
```

```
                maxXCoordinate = location[0]


            # update vertical plot constraints
            if location[1] < minYCoordinate:
                minYCoordinate = location[1]
            elif location[1] > maxYCoordinate:
                maxYCoordinate = location[1]


    # Helper method for generating random sink locations.
    def generateRandomSinkLocations(self):
        radius = 1000
        rangeX = (self.maxXCoordinate/5,
(9*self.maxXCoordinate/10))
        rangeY = (self.maxYCoordinate/4,
(9*self.maxYCoordinate/10))
        qty = self.num_sinks  # or however many points you want


        # Generate a set of all points within 200 of the origin,
to be used as offsets later
        # There's probably a more efficient way to do this.
        deltas = set()
        for x in range(-radius, radius+1):
```

```
            for y in range(-radius, radius+1):
                if x*x + y*y <= radius*radius and x*x <=
(self.maxXCoordinate*2) and y*y <= (self.maxYCoordinate):
                    deltas.add((x,y))

        excluded = set()
        i = 0
        while i<qty:
            x = random.randrange(*rangeX)
            y = random.randrange(*rangeY)
            if (x,y) in excluded: continue
            self.sinks.append(Sink(x, y))
            i += 1
            excluded.update((x+dx, y+dy) for (dx,dy) in deltas)

    # We perform KMeans cluster to group our sinks into
pre-defined cluster sizes.
    # TODO: Make the cluster sizes dynamic (based on number of
sinks per grouping)
    def groupSinks(self):
        # Get all sinks locations
        sinks = self.getSinkLocations()
```

```
        # determine best cluster size

        num_clusters = self.determineNumberOfClusters()

        # Perform KMeans clustering and identify cluster
centers.

        kmeans = KMeans(n_clusters=num_clusters).fit(sinks)

        centroids = kmeans.cluster_centers_


        # For each identified cluster center:

        # Create a new Centroid

        # Add Centroid to our tree

        # Create a new SinkGroup

        for i, location in enumerate(centroids):

            centroid = Centroid(location[0], location[1])

            self.addCentroid(centroid, i)

            self.createSinkGroup(centroid, i)


        # Once we've made a SinkGroup for each cluster center

        # We'll use the list of groupings, and add each Sink to
their designated group.

        self.addSinksToGroups(kmeans.labels_)

        print "required wire length:",
self.standardizedWireLength
```

```
    def determineNumberOfClusters(self):
        range_n_clusters = self.findAllFactors()
        scores = {}
        sink_locations = self.getSinkLocations()

        for n_clusters in range_n_clusters:
            # Initalize the clusterer with n_clusters value
            # and a random generator seed of 10 for
reproducibility.
            clusterer = KMeans(n_clusters=n_clusters,
random_state=10)
            cluster_labels =
clusterer.fit_predict(sink_locations)

            # The silhouette_score gives the average value for
all the samples.
            # This gives a perspective into the density and
separation of the formed
            # clusters
```

```
            silhouette_avg = silhouette_score(sink_locations,
cluster_labels)

            scores[n_clusters] = silhouette_avg

        # Get cluster size that's closest to the median of all
the silhouette average.
        averages = list(scores.values())
        median = statistics.median(averages)
        closest_value_to_median = min(averages, key=lambda
x:abs(x-median))

        for size, avg in scores.items():
            if avg == closest_value_to_median:
                print "number of sinks:", self.num_sinks
                print "optimal cluster size:", size
                return size

    # Helper method which finds all factors of a number.
    # This will be used to determine the number of sinks grouped
per level in the tree.
```

```
    # Algorithm found here:
https://stackoverflow.com/a/6800214/5464998
    def findAllFactors(self):
        n = self.num_sinks
        factors = sorted(set(reduce(list.__add__,
                    ([i, n//i] for i in range(1, int(n**0.5) + 1) if
n % i == 0))))

        return factors[1:(len(factors)-1)]

    # Returns the location of each of the sinks in our clock
tree.
    def getSinkLocations(self):
        locations = []
        for sink in self.sinks:
            locations.append(sink.location)
        return locations

    # Adds a Centroid to our tree
    def addCentroid(self, centroid, i):
        self.centroids.append(centroid)
```

```
    # Creates a dictionary entry, which represents our
SinkGroups.
    # SinkGroups are indexed by their group_id (i)
    def createSinkGroup(self, centroid, i):
        sinkGroupDict = {
            "centroid_location": [centroid.location[0],
centroid.location[1]],
            "sinks": []
        }

        self.sinkGroups[i] = sinkGroupDict

    # Adds each sink to their designated SinkGroup by group_id
    def addSinksToGroups(self, group_ids):
        for i, group_id in enumerate(group_ids):
            sink = self.sinks[i]
            self.sinkGroups[group_id]["sinks"].append(sink)
            self.maybeUpdateWireLength(group_id, sink)

    # Checks to see if we have a new longest wire length
    # If we do, we'll update the distance value
    def maybeUpdateWireLength(self, group_id, sink):
```

```
        centroid_location =
self.sinkGroups[group_id]["centroid_location"]

        # calculate wire length between sink and centroid
        # sqrt((x2 - x1)^2 + (y2 - y1)^2)
        wire_length = self.DistanceToGoal(sink.location,
centroid_location)

        # set max distance if applicable
        if (wire_length > self.standardizedWireLength):
            self.standardizedWireLength = wire_length

    # Draws the plot of tree.
    def makePlot(self):
        self.setAxis()
        self.plotSinkGroups()
        self.showPlot()

    # Sets the axes for our plot.
    def setAxis(self):
        # ([minX, maxX, minY, maxY])
```

```
        plt.axis([self.minXCoordinate,
(6*self.maxXCoordinate/5), self.minYCoordinate,
(6*self.maxXCoordinate/5) ])


    # Shows our plot
    def showPlot(self):
        plt.show()


    # Plots each of our sink groups
    def plotSinkGroups(self):
        # Get all our plots attributes
        plot_attributes = PlotAttributes(self.sinkGroups)


        for group_id in self.sinkGroups:
            centroid_location =
self.sinkGroups[group_id]["centroid_location"]
            # plot centroid (x, y, attributes)
            plt.plot(
                centroid_location[0],
                centroid_location[1],

color=plot_attributes.getCentroidAttributes(group_id)["color"],
```

```
marker=plot_attributes.getCentroidAttributes(group_id)["marker"]

            )

            # plot sinks (x, y, attributes)
            for sink in self.sinkGroups[group_id]["sinks"]:
                plt.plot(
                    sink.location[0],
                    sink.location[1],

color=plot_attributes.getSinkAttributes(group_id)["color"],

marker=plot_attributes.getSinkAttributes(group_id)["marker"]
                )

                self.drawConnection(sink.location,
centroid_location, group_id)

    # draws the line that connects a sink to a centroid
    # takes into account the required (longest) wire length when
    # devising a path
```

```
    # wip
    def drawConnection(self, start, goal, group_id):

        # next, we're going to snake the wire from the sink to
the centroid,

        desired_distance = self.standardizedWireLength
        exact_distance = self.DistanceToGoal(start, goal)

        # Too close
        if exact_distance != desired_distance:
            self.drawSnakeLine(start, goal, desired_distance,
group_id)
        else:
            # Equidistant, so draw direct connection
            self.drawLineBetweenTwoPoints(start, goal, group_id)

    def drawSnakeLine(self, start, goal, goal_wire_length,
group):

        bounding_thresholds = {
            "down": self.minXCoordinate,
```

```
            "up": 3*self.maxYCoordinate/2,
            "left": self.minXCoordinate,
            "right": 3*self.maxXCoordinate/2
        }

        pen = start[:]
        last_pen_position = pen[:]

        direction_traveled = { "up": False, "down": False,
"left": False, "right": False }

        # turning tracks which axis we're traveling on.
        turning = False

        continue_drawing_line = True

        # booleans that track which point we're going to or
coming from.
        connect_to_start = True
        connect_to_goal = False

        # Initialize our wire length
```

```
        current_wire_length = 0


        while continue_drawing_line:
            # get relative direction
            direction =
self.getCurrentLocationRelativeToCentroid(start, goal, turning)


            # initialize recalculated_direction
            recalculated_direction = direction


            # determine the best direction to begin drawing
            direction_to_travel =
self.getNextDirectionToTravel(start, goal, turning)


            # movingInPositiveDirection tracks which axis we're
moving in
            movingInPositiveDirection = False if
(direction_to_travel == "left" or direction_to_travel == "down")
else True
            # movingHorizontally tracks which direction we're
moving in
```

```
            movingHorizontally = False if (direction_to_travel
== "up" or direction_to_travel == "down") else True

            # index tracks which pen position we're changing (x
or y)
            index = 0 if movingHorizontally else 1

            # change sets the amount we increment / decrement
each time.
            change = 5 if movingInPositiveDirection else -5

            # overshoot_counter tracks how many times we've
crossed from the pos -> neg axis, relative to the goal location.
            overshoot_counter = 0

            # we allow ourselves to cross the axis 5 times
            while overshoot_counter < 5:
                # check if we've hit the graph bounds.
                # if we have, we'll trip a boolean so we don't
keep trying the same direction
                if movingInPositiveDirection and pen[index] +
change > bounding_thresholds[direction_to_travel]:
```

```
                    self.directionTraveled[direction] = True
                    break
                elif not movingInPositiveDirection and
pen[index] + change < bounding_thresholds[direction_to_travel]:
                    self.directionTraveled[direction] = True
                    break

                # check to see if we've passed the goal on this
axis.
                elif recalculated_direction != direction:
                    overshoot_counter += 1

                # update the pen location and
current_wire_length
                pen[index] += change
                current_wire_length += abs(change)

                # check if we've reached our goal wire length
                if current_wire_length >= goal_wire_length:
                    connect_to_goal = True
                    break
```

```
                # check our pen's location relative to the
centroid
                recalculated_direction =
self.getCurrentLocationRelativeToCentroid(pen, goal, turning)

            # Check what type of point we're connecting
            if connect_to_start:
                self.drawLineBetweenTwoPoints(start, pen, group)
                last_pen_position = pen[:]
                connect_to_start = False

            elif connect_to_goal:
                self.drawLineBetweenTwoPoints(pen, goal, group)
                continue_drawing_line = False

            else:
                self.drawLineBetweenTwoPoints(last_pen_position,
pen, group)
                last_pen_position = pen[:]

            # toggle our turning variable
            turning = not turning
```

```
            self.resetDirectionTravel()


    # Helper method to draw a line between two points
    def drawLineBetweenTwoPoints(self, start, goal, group):
        plot_attributes = PlotAttributes(self.num_sinks)

        plt.plot(
            [start[0], goal[0]],
            [start[1], goal[1]],

color=plot_attributes.getLineAttributes(group)["color"]
        )


    # Helper method which calculates current location relative
to centroid
    def getCurrentLocationRelativeToCentroid(self, start, goal,
turning=False):
        if turning:
            return "left" if start[0] > goal[0] else "right"
        else:
            return "down" if start[1] > goal[1] else "up"
```

```
    # Helper method to determine which direction we'll travel in
    def getNextDirectionToTravel(self, start, goal,
turning=False):

        # Algorithm checks to see which direction we have the
most space to travel in.
        # Once one is selected, we do a lookup to see if we've
already tried to go in
        # that direction during this iteration.
        # If we have, we try the opposite direction.
        # If we've tried both, we'll call the function and move
in a different
        # orientation.
        if turning:
            possible_dir = "right" if start[0] > goal[0] else
"left"
            if self.directionTraveled[possible_dir] == False:
                return possible_dir
            else:
                new_dir = "left" if possible_dir == "right" else
"right"
```

```
                if self.directionTraveled[new_dir] == False:

                    return new_dir

                else:

                    self.resetDirectionTravel()

                    return getNextDirectionToTravel(start, goal,
False)


        else:

            possible_dir =  "up" if start[1] > goal[1] else
"down"


            if self.directionTraveled[possible_dir] == False:

                return possible_dir

            else:

                new_dir = "up" if possible_dir == "down" else
"down"


                if self.directionTraveled[new_dir] == False:

                    return new_dir

                else:

                    self.resetDirectionTravel()
```

```
                    return getNextDirectionToTravel(start, goal,
True)



# Initialize a symmetric clock tree, with n sinks.
tree = SymmetricClockTree(40)

# Uncomment the line below to input custom data
# Method expects locations to be a list of x, y coordinates:
[[x,y], [x,y],...]
# tree.addSinksByLocation(location)

tree.generateRandomSinkLocations() # Comment out if passing
custom data.

tree.groupSinks()
tree.makePlot()

# Add ability to set the number of clusters manually.
# Data is nm
```