THAPAR INSTITUTE OF ENGINEERING AND TECHNOLOGY

MECHANICAL ENGIEERING DEPARTMENT

RESEARCH INTERNSHIP:

NANYANG TECHNOLOGICAL UNIVERSITY, SINGAPORE

MIDWAY REPORT

SUBMITTED TO:	UNDER GUIDANCE OF:	SUBMITTED BY:
Dr. R.K Duvedi	Dr. Cai Yiyu	Shivu Chauhan
MED, TIET, Patiala	MAE, NTU, Singapore	MTX2, 101809031

TABLE OF CONTENTS

CHAPTER 1

Study of Python Language and it’s Data Structures

- Aim:

  I was asked about my previous experience in the field of Robotics Algorithms Programming and implementation, Python Programming and Development in ROS. Coming from a mechatronics background I was well versed with ROS and development in C++ and MATLAB. However, Python was a new programming language for me.

  I had to learn python language and it’s various Data structures techniques to complete the tasks that were to be assigned in internship.

- Relevance of the project undertaken:

  Python is a high-level programming language with most of its syntax in plain English, hence easily comprehendible and faster, efficient working. It is used in many domains not limiting to Web dev, app dev, CP, computer vision, AI & ML, Electronics, finance.

  Like other modern programming languages, Python also supports several programming paradigms. It supports object oriented and structured programming fully. Also, its language features support various concepts in functional and aspect-oriented programming. At the same time, Python also features a dynamic type system and automatic memory management. The programming paradigms and language features help one to use Python for developing large and complex software applications.

  Python can create prototype of the software application rapidly. Also, we can build the software application directly from the prototype simply by refactoring the Python code. Python even makes it easier for us to perform coding and testing simultaneously by adopting test driven development (TDD) approach. We can easily write the required tests before writing code and use the tests to assess the application code continuously. The tests can also be used for checking if the application meets predefined requirements based on its source code.

Knowledge of python programming is beneficial for an aspiring Mechatronics engineer, for automating mechanical tasks and designing mechatronic systems.

- Project Objectives:

To learn basic syntax of Python like conditionals, loops followed by advanced concepts of python like OOPS, data processing, reg expressions, to be able to use python data structures likes Stacks, queues, lists, dicts, trees, graphs, modelling etc.
to produce good quality code and aid in faster development of algorithms.

- Results:

During the first initial weeks I started learning python through online resources mostly from YouTube, Code academy, and the official documentation of the language. I practiced online tutorials and started off solving problems like Tic tac game programming, Object oriented programming in python, Using python libraries like Matplotlib, pandas for data visualization. I realized the best Faculty Supervisor comments way to learn programming language is by ‘do-and learn’ approach. After two weeks I was well versed with basics of python, syntax, classes and object, Regression expressions, loops, data visualization, stacks, queues, etc.

I learned about:
➢Complexity (time and space)
➢ Stacks and Queues both static and dynamic, Priority
➢ Tree methodology, BST, Graphs ,
➢ Dynamic programming in Python
➢ Graph theory

➢ Lists and Doubly connected edge lists for CGAL 5.0

➢ Reflections:

Knowledge of python programming is of utmost importance for any prospective mechatronics engineer. I learned about automation in python, mostly in mechatronics domain and I learned about programming controllers, sensors, actuators using python. Gained knowledge of data structures and dynamic programming which is used as entrance tests for many product-based companies. I also learned about functioning of Cython which is a hybrid language of Python & C++ combining Python efficiencies with C++'s speed.

CHAPTER 2
Understanding Algorithms

➢ Aim

To understand the importance of Algorithms in real world applications Using python to construct them and be able to comprehend them with the previously acquired knowledge of Data structures.

➢ Relevance of the Project Undertaken

Algorithms are used in every part of computer science. They form the field's backbone. In computer science, an algorithm gives the computer a specific set of instructions, which allows the computer to do everything, be it running a calculator or running a rocket. Computer programs are, at their core, algorithms written in programming languages that the computer can understand. Computer algorithms play a big role in how social media works: which posts show up, which ads are seen, and so on. These decisions are all made by algorithms. Google's programmers use algorithms to optimize searches, predict what users are going to type, and more. In problem-solving, a big part of computer programming is knowing how to formulate an algorithm.

Algorithmic thinking, or the ability to define clear steps to solve a problem, is crucial in many different fields. Even if we're not conscious of it, we use algorithms and algorithmic thinking all the time. Algorithmic thinking allows us to break down problems and conceptualize solutions in terms of discrete steps. Being able to understand and implement an algorithm requires students to practice structured thinking and reasoning abilities.

➢ Project Objectives

To start from the basics of Algorithmic Programming by familiarizing with most popular and simpler algorithms used today. To move on subsequently to harder algorithms formulated, to break Algorithms into simpler steps and program each step with its condition in python language. Ability to comprehend algorithms using data structures in python.

➢ Results

Learned and programmed the following algorithms in python:

- Dijkstra's Algo: Given a graph and a source vertex in the graph, find shortest paths from source to all vertices in the given graph. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

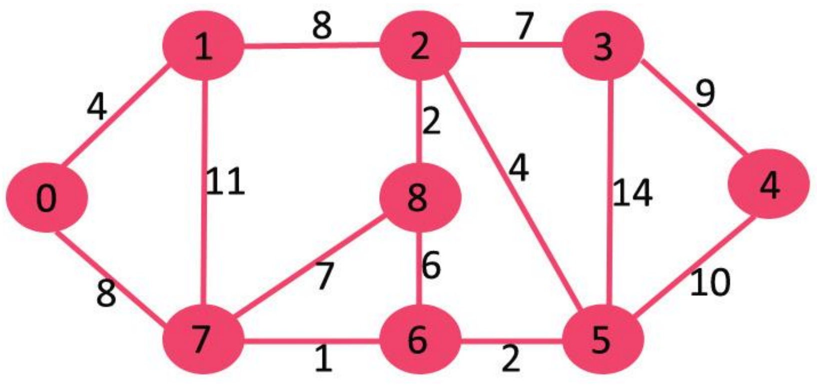


Figure 1: Dijkstra's Algo

- A* Algorithm: A* Search algorithm is one of the best and popular technique used in path-finding and graph traversals. Informally speaking, A* Search algorithms, unlike other traversal techniques, it has “brains”. What it means is that it is really a smart algorithm which separates it from the other conventional algorithms. This fact is cleared in detail in below sections.
  And it is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently (approximation).

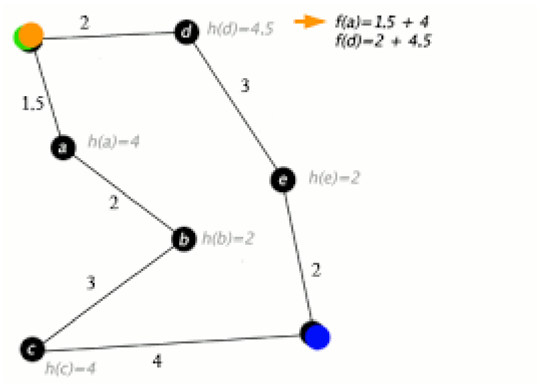


Figure 2: A Algorithm*

- Bubble Sort
- Binary Search
- Selection Sort
- Insertion Sort
- Merge Sort
- Heap Sort
- Quick Sort

- Reflection

I learned some of the most famous algorithms in the field of computer science particularly in navigations and searches domain. I learned how to comprehend algorithms break them into smaller parts and implement its programming in python. Also gained knowledge on complexity of algorithms in terms of Big (O) notation in time and space. How to reduce computation time and increase efficiency in programs.

CHAPTER 3

Understanding the Art gallery problem

- Aim

I was assigned to solve the famous ‘Art Gallery Problem’ posed by Victor Klee in 1977, it is an NP Hard problem in Computational Geometry. Solving the problem by forming an algorithm using data structures in python . Aim to reduce the complexity and the computation.

- Relevance of the Project

One of the main goals in robotics is to create robots that are able to mimic human behaviour, a large component of which is motion sensors and navigation. Robots with these capabilities would be able to localize themselves in a new environment, which would solve a myriad of problems, one of the simplest being guarding an art gallery. The art gallery problem was one of the earliest and most influential problems in sensor placement. The problem was first posed to V´aclav Chv´atal by Victor Klee in 1973 and was stated as: *Consider an art gallery, what is the minimum number of stationary guards needed to protect the room*? In geometric terms, the problem was stated as: given a n-vertex simple polygon, what is the minimum number of guards to see every point of the interior of the polygon? Chv´atal was able to prove that for simple polygons [n/3] guards are necessary and sufficient to guard the gallery when there are n vertices in the polygon. However, this proof was very complex and used the method of induction. In 1978 Steve Fisk constructed a much simpler proof via triangulation, which is a method of decomposing a polygon into triangles, and coloring of vertices. The many real-life applications of this problem not only inspired the mathematics community to find a tighter bound due to real-life restrictions, but also inspired many variations of the problem that modelled real-life situations

- Project Objectives:

  The Art Gallery problem is a famous problem in the field of Computational Geometry. The variant of the problem discussed in this report is defined as follows: given a polygon(possibly with holes) P, the goal is to find the smallest set of point guards $G \subset P$ such that every point $p \in P$ is at least seen by one of the guards $g \in G$. A guard sees a point within P if the segment between the guard and the point is contained in P.

- Results

  I read several research papers from various journals and conferences on computing and computational geometry. Gained knowledge on various modifications of the art gallery problem and how it has been addressed by various researchers in the field. I was given access to NTU's e-library resources and professors research databases. I read about various applications of this problem in field on Robotics and sensor placements, Also in BIM (Building Infrastructure Management) and LiDAR placements in the structures.

- Reflection:

  Computational geometry is a relatively new field in computer science originating in the latter part of the 20th century. The support for this in python is relatively less. The international community is also small. I feel lucky enough to get exposure to this field, it is used in Geographic Information systems, CAD / CAM, Robotics etc. C++ has a very huge and versatile support library for computational geometry by the name of CGAL. Most of its features are restricted to C++ only but by using

Swigwin 5.0 we can code some of its functionality in python also. For python we need to develop modules for triangulations, dual graph formations and 3 coloring ourselves to use them efficiently in other codes.

CHAPTER 4

Beginning Python Implementations'

- Aim

To produce the Art gallery problem in python by reviewing the algorithms in research papers. By reviewing various data structures and deciding which ones to employ in programming techniques. Also, to reduce the complexity in Big (O) notation in time and space.

- Relevance of the Project Undertaken

With increasing complexities in computer algorithms, the amount of data usage is increasing, this can affect the performance of the application and can create some areas of concern:

*Processing speed***:** To handle very large data, high-speed processing is required, but with growing data processor may fail to achieve required processing speed.

*Data Search***:** Getting a particular record from database should be quick and with optimum use of resources.

*Multiple requests***:** To handle simultaneous requests from multiple users

In order to work on concern areas, data structures are used. Data is organized to form a data structure in such a way that all items are not required to be searched and required data can be searched instantly.

*Efficient Memory use***:** With efficient use of data structure memory usage can be optimized, for e.g. we can use linked list vs arrays when we are not sure about the size of data. When there is no more use of memory, it can be released.

Reusability: Data structures can be reused, i.e. once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.

Abstraction: Data structure serves as the basis of abstract data types; the data structure defines the physical form of ADT(Abstract Data Type). ADT is theoretical and Data structure gives physical form to them.

- Project Objectives

To run the Art gallery problem solution in the python IDE (Sublime Text used) . To successfully find the minimum number of guards required to guard an arbitrary gallery with n vertices , the vertices can be entered in clockwise or anti clockwise order in the form of (x,y) coordinates in the program.

- Results

Sublime text was installed in the pc with REPL inbuilt function for user friendly input of the program.

Programming was started initially using DCEL data structure for entering the data of the polygon .Initially I used the DCEL structure for taking the input of the coordinates of the art gallery in (x, y) plane from the user, DCEL helped in triangulating the polygon and setting flags on each vertex of the triangle formed after visiting it when traversing the gallery.

I realized that Graph theory should be used as through it we can color each vertex of the triangles formed after triangulation of the polygon with a unique color and then we can make a function which traverses through the polygon and each triangle and count the frequency of the colours, the least frequent color would be the coordinates where we will place the guards for guarding and hence the solution.

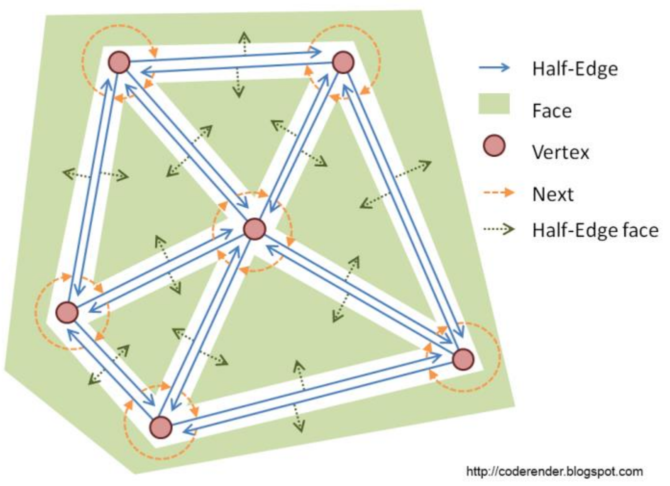


Figure 3: DCEL Structure for a polygon

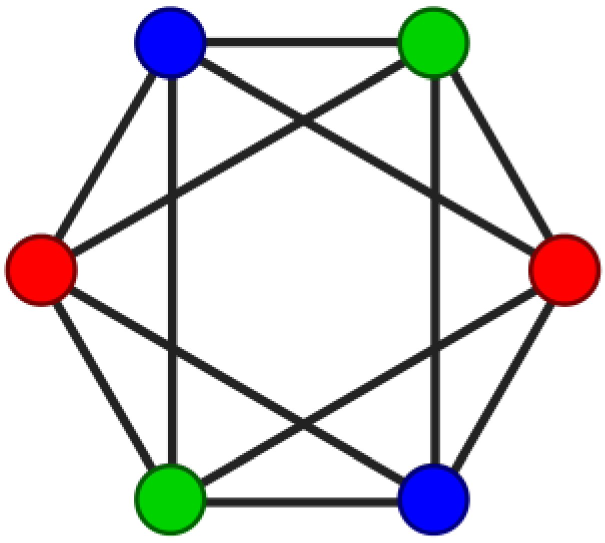


Figure 4: 3-Coloring of Graph

Firstly the DCEL Solution Implemented in python:

```
import numpy as np
DEBUG = False

class Point:
    def __init__(self, coordinates,auxData=None):
        self.data=auxData
        self.coords = coordinates
        self.edge = None
        self.ear = False
        self.next = None
        self.prev = None
        self.color= -1
    def ___str___(self):
        return str(self.ID)
    def __getitem__(self,key):
        return self.coords[key]
    def scale(self, k1, k2):
        self.coords = list(self.coords)
        self.coords[0] = int(self.coords[0] * k1)
        self.coords[1] = int(self.coords[1] * k2)
        self.coords = tuple(self.coords)
    def __hash__(self):
        return hash(id(self))
    def getData(self):
        return self.data
    def setData(self, auxData):
        self.data = auxData
    def getCoords(self):
        return Point(self.coords)
    def setCoords(self):
        self.coords = coordinates
    def getOutgoingEdges(self):
        visited = set()
        out = []
        here = self.edge
        while here and here not in visited:
            out.append(here)
            visited.add(here)
            temp = here.getTwin()
            if temp:
                here = temp.getNext()
            else:
                here = None
        return out
    def getIncidentEdge(self):
        return self.edge
    def setIncidentEdge(self, edge):
        self.edge = edge
    def __repr__(self):
        return 'DCEL.Point with coordnates (' +
str(self.coords[0])+','+str(self.coords[1])+')'

class Edge:
    def __init__(self, auxData=None):
        self.data = auxData
        self.twin = None
        self.origin = None
        self.face = None
        self.next = None
        self.prev = None
    def __hash__(self):
        return hash(id(self))
    def getTwin(self):
        return self.twin
    def setTwin(self, twin):
        self.twin = twin
    def getData(self):
        return self.data
    def setData(self, auxData):
        self.data = auxData
    def getNext(self):
        return self.next
    def setNext(self, edge):
        self.next = edge
    def getOrigin(self):
        return self.origin
    def setOrigin(self, v):
        self.origin = v
    def getPrev(self):
        return self.prev
    def setPrev(self, edge):
        self.prev = edge
    def getDest(self):
        return self.twin.origin
    def getFace(self):
        return self.face
    def getFaceBoundary(self):
        visited = set()
        bound = []
        here = self
        while here and here not in visited:
            bound.append(here)
            visited.add(here)
            here = here.getNext()
        return bound
    def setFace(self, face):
        self.face = face
    def clone(self):
        c = Edge()
        c.data,c.twin,c.origin,c.face,c.next,c.prev =
self.data,self.twin,self.origin,self.face,self.next,self.prev
    def __repr__(self):
        return 'DCEL.Edge from Origin: DCEL.Point with coordinates (' +
str(self.getOrigin().coords[0])+','+str(self.getOrigin().coords[1])+')' +
'\nDestination: DCEL.Point with coordinates (' +
str(self.getDest().coords[0])+','+str(self.getDest().coords[1])+')'

class Face:
    def __init__(self, auxData=None):
        self.data = auxData
        self.outer = None
        self.inner = set()
        self.isolated = set()
    def __hash__(self):
        return hash(id(self))
    def getOuterComponent(self):
        return self.outer
    def setOuterComponent(self, edge):
        self.outer = edge
    def getData(self):
        return self.data
    def setData(self, auxData):
        self.data = auxData
    def getOuterBoundary(self):
        if self.outer:
            return self.outer.getFaceBoundary()
        else:
            return []
    def getOuterBoundaryCoords(self):
        original_pts = self.getOuterBoundary()
        return [x.origin.coords for x in original_pts]
    def getInnerComponents(self):
        return list(self.inner)
    def addInnerComponent(self, edge):
        self.inner.add(edge)
    def removeInnerComponent(self, edge):
        self.inner.discard(edge)
    def removeIsolatedVertex(self,Point):
        self.isolated.discard(Point)
    def getIsolatedVertices(self):
        return list(self.isolated)
    def addIsolatedVertex(self,Point):
        self.isolated.add(Point)

class DCEL:
    def __init__(self):
        self.exterior = Face()
    def getExteriorFace(self):
        return self.exterior
    def getFaces(self):
        result = []
        known = set()
        temp = []
        temp.append(self.exterior)
        known.add(self.exterior)
        while temp:
            f = temp.pop(0)
            result.append(f)
            for e in f.getOuterBoundary():
                nb = e.getTwin().getFace()
                if nb and nb not in known:
                    known.add(nb)
                    temp.append(nb)
            for inner in f.getInnerComponents():
                for e in inner.getFaceBoundary():
                    nb = e.getTwin().getFace()
                    if nb and nb not in known:
                        known.add(nb)
                        temp.append(nb)
        return result

    def getEdges(self):
        edges = set()
        for f in self.getFaces():
            edges.update(f.getOuterBoundary())
            for inner in f.getInnerComponents():
                edges.update(inner.getFaceBoundary())
        return edges

    def getVertices(self):
        verts = set()
        for f in self.getFaces():
            verts.update(f.getIsolatedVertices())
            verts.update([e.getOrigin() for e in f.getOuterBoundary()])
            for inner in f.getInnerComponents():
                verts.update([e.getOrigin() for e in inner.getFaceBoundary()])
        return verts


def buildSimplePolygon(points):
    d = DCEL()
    if points:
        exterior = d.getExteriorFace()
        interior = Face()
        verts = []
        for p in points:
            verts.append(Point(p))
        innerEdges = []
        outerEdges = []
        for i in range(len(verts)):
            e = Edge()
            e.setOrigin(verts[i])
            verts[i].setIncidentEdge(e)
            e.setFace(interior)
            t = Edge()
            t.setOrigin(verts[(i+1)%len(verts)])
            t.setFace(exterior)
            t.setTwin(e)
            e.setTwin(t)
            innerEdges.append(e)
            outerEdges.append(t)

        for i in range(len(verts)):
            innerEdges[i].setNext(innerEdges[(i+1)%len(verts)])
            innerEdges[i].setPrev(innerEdges[i-1])
            outerEdges[i].setNext(outerEdges[i-1])
            outerEdges[i].setPrev(outerEdges[(i+1)%len(verts)])
        interior.setOuterComponent(innerEdges[0])
        exterior.addInnerComponent(outerEdges[0])
    return d
```

- Reflection:

DCEL didn't help in keeping track that which vertex was visited the greatest number of times and which was visited least frequently, this assumption was important as we have to place least number of guards to guard the gallery.

I had to change approach.

CHAPTER 5

Solving Art Gallery in Minimum Computations

- Aim

To solve the art gallery problem by using other algorithms and programming techniques instead of using DCEL and Monotone partitioning which didn't yielded the results expected as seen above.

- Relevance of the project undertaken

  Debugging is an important part of computer programming. We have to change algorithms and data structure techniques when they fail to deliver the expected results.
  It is important to study the system in depth in order to understand the system. It helps the programmer to construct different representations of systems that are to be debugged.
  Backward analysis of the problem traces the program backward from the location of failure message in order to identify the region of faulty code. We need to study the region of defect thoroughly to find the cause of defects.
  Forward analysis of the program involves tracking the program forward using breakpoints or print statements at different points in the program. It is important to focus on the region where the wrong outputs are obtained.

  I learnt a great deal on debugging through this project.

- Project Objectives

  To run the debugged Art gallery problem solution in the python IDE. To successfully find the minimum number of guards required to guard an arbitrary gallery with n vertices , the vertices can be entered in clockwise or anti clockwise order in the form of (x,y) coordinates in the program.

- Results

  I now programmed two different modules and imported them to a third main program where the results were displayed.

  The first module is for triangulation of the polygon using the Two Ears Miesters theorem :
  In geometry, the two ears theorem states that every simple polygon with more than three vertices has at least two ears, vertices that can be removed from the polygon without introducing any crossings. The two ears theorem is equivalent to the existence of polygon triangulations.

  Polygon_traingulate.py:

```
import math
import sys
from collections import namedtuple

Point = namedtuple('Point', ['x', 'y'])

def check_clockwise(polygon):
    s = 0
    counts = len(polygon)
    for i in range(counts):
        point = polygon[i]
        point2 = polygon[(i + 1) % counts]
        s += (point2.x - point.x) * (point2.y + point.y)
    return s > 0

def check_ear(p1, p2, p3, polygon):
    return Nopoints_check(p1, p2, p3, polygon) and check_convex(p1, p2,
p3) and area_triangle(p1.x, p1.y, p2.x, p2.y, p3.x, p3.y) > 0

def Nopoints_check(p1, p2, p3, polygon):
    for pn in polygon:
        if pn in (p1, p2, p3):
            continue
        elif check_inside(pn, p1, p2, p3):
            return False
    return True

def check_inside(p, a, b, c):
    area = area_triangle(a.x, a.y, b.x, b.y, c.x, c.y)
    area1 = area_triangle(p.x, p.y, b.x, b.y, c.x, c.y)
    area2 = area_triangle(p.x, p.y, a.x, a.y, c.x, c.y)
    area3 = area_triangle(p.x, p.y, a.x, a.y, b.x, b.y)
    areadiff = abs(area - sum([area1, area2, area3])) <
(math.sqrt(sys.float_info.epsilon))
    return areadiff

def area_triangle(x1, y1, x2, y2, x3, y3):
    return abs((x1 * (y2 - y3) + x2 * (y3 - y1) + x3 * (y1 - y2)) / 2.0)

def check_convex(prev, point, next):
    return sum_triangle(prev.x, prev.y, point.x, point.y, next.x, next.y) < 0

def sum_triangle(x1, y1, x2, y2, x3, y3):
    return x1 * (y3 - y2) + x2 * (y1 - y3) + x3 * (y2 - y1)

def Earclip(polygon):

    ear_vertex = []
    triangles = []

    polygon = [Point(*point) for point in polygon]

    if check_clockwise(polygon):  # Make polygon anticlockwise
        polygon.reverse()

    counts = len(polygon)  # Find ear vertices and add to ear_vertex
    for i in range(counts):
        prev_point = polygon[i - 1]
        point = polygon[i]
        next_point = polygon[(i + 1) % counts]
        if check_ear(prev_point, point, next_point, polygon):
            ear_vertex.append(point)

    while ear_vertex and counts >= 3:
        ear = ear_vertex.pop(0)
        i = polygon.index(ear)
        prev_index = i - 1
        prev_point = polygon[prev_index]
        next_index = (i + 1) % counts
        next_point = polygon[next_index]

        polygon.remove(ear) #Removing ear vertices from polygon
        counts -= 1
        triangles.append(((prev_point.x, prev_point.y), (ear.x, ear.y),
(next_point.x, next_point.y)))
        if counts > 3:
            prev_prev_point = polygon[prev_index - 1]
            next_next_index = (i + 1) % counts
            next_next_point = polygon[next_next_index]
            matches = [
                (prev_prev_point, prev_point, next_point, polygon),
                (prev_point, next_point, next_next_point, polygon),
            ]
            for match in matches:
                q = match[1]
                if check_ear(*match):
                    if q not in ear_vertex:
                        ear_vertex.append(q)
                elif q in ear_vertex:
                    ear_vertex.remove(q)
    return triangles
```

The second module is for 3 graph coloring and graph formation :

Graph_coloring.py

```
class graph():


    def __init__(self, vertices):

        self.Vertex = vertices

        self.Graph = [[0 for column in range(vertices)]

                      for row in range(vertices)]


    # check  current color assignment for vertex v

    def Check_color(self, v, colour, c):

        for i in range(self.Vertex):

            if self.Graph[v][i] == 1 and colour[i] == c:

                return False

        return True


    # recursive function to solve coloring problem
    def Graph_util(self, m, colour, v):
        if v == self.Vertex:
            return True

        for c in range(1, m+1):
            if self.Check_color(v, colour, c) == True:
                colour[v] = c
                if self.Graph_util(m, colour, v+1) == True:
                    return True
                colour[v] = 0

    def Check_graph_color(self, m):
        colour = [0] * self.Vertex
        if self.Graph_util(m, colour, 0) == False:
            return False

        #answer
        color_array = []
        for c in colour: color_array.append(c)
        return color_array
```

Combining them in third file which assigns color to the vertices of the triangulated polygon and finds the least frequent color and the position where the guards need to be placed.

Main.py:

```
from polygon_triangulate import *
from Graphcoloring import *
```

```
# Cartesian coordinates for the vertices of a CLOSED polygon, in either
clockwise or counter-clockwise order

polygon = [(0,0),(1,0) , (1,1) , (2,4) , (3,1) , (4,1) , (5,4) , (6,1) , (7,1) ,
(8,4) , (9,1) , (10,1) , (10,0)] # n = 5


print(polygon) # the input


# Triangulate the polygon

triangles = Earclip(polygon) # Returns 2D list of triangle vertex
coordinates


# Add ^those vertices to a dict, without repeating any, mapping each to a
index from 0 to n-1,

vertices = {}

for x, y, z in triangles:

    if x not in vertices: vertices[x] = len(vertices)

    if y not in vertices: vertices[y] = len(vertices)

    if z not in vertices: vertices[z] = len(vertices)


# initialize adjacency matrix of the triangle mesh to zeroes

num_vertices = len(vertices)


adjacency_matrix = [[0 for i in range(num_vertices)] for j in
range(num_vertices)]


# populate adjacency matrix

for x, y, z in triangles:
    adjacency_matrix[vertices.get(x)][vertices.get(y)] = 1
    adjacency_matrix[vertices.get(x)][vertices.get(z)] = 1

    adjacency_matrix[vertices.get(y)][vertices.get(x)] = 1
    adjacency_matrix[vertices.get(y)][vertices.get(z)] = 1

    adjacency_matrix[vertices.get(z)][vertices.get(x)] = 1
    adjacency_matrix[vertices.get(z)][vertices.get(y)] = 1

print(triangles) #Outputs the cartesian coordinates of resulting triangles
print(vertices) #Outputs the vertices of resulting triangles
for x in adjacency_matrix: print(x) #Outputs the adjacency matrix of the
vertices

# Tricolor the vertices
g = graph(num_vertices)
g.Graph = adjacency_matrix
colors = g.Check_graph_color(3) # returns array with colors, in the order
of vertices as in the dict vertices

# map colors to vertices
colored_vertices = {}
n = 0
for i in vertices.keys():
    colored_vertices[i] = [colors[n]]
    n+=1

# count frequency of colors and store in dict
```

```
color_freq = {}
for i in colors:
    if i in color_freq: color_freq[i]+=1
    else: color_freq[i] = 1

# determine least frequent color
frequency = num_vertices
for i in color_freq.keys():
    if color_freq[i] < frequency:
        frequency = color_freq[i]
        least_frequent_color = i

# output array with cartesian coordinates for camera placement
chosen_vertices = []
for i in colored_vertices.keys():
    if colored_vertices[i] == [least_frequent_color]:
        chosen_vertices.append(i)
print(num_vertices)
print("Guards to placed at" , chosen_vertices) # the output
```

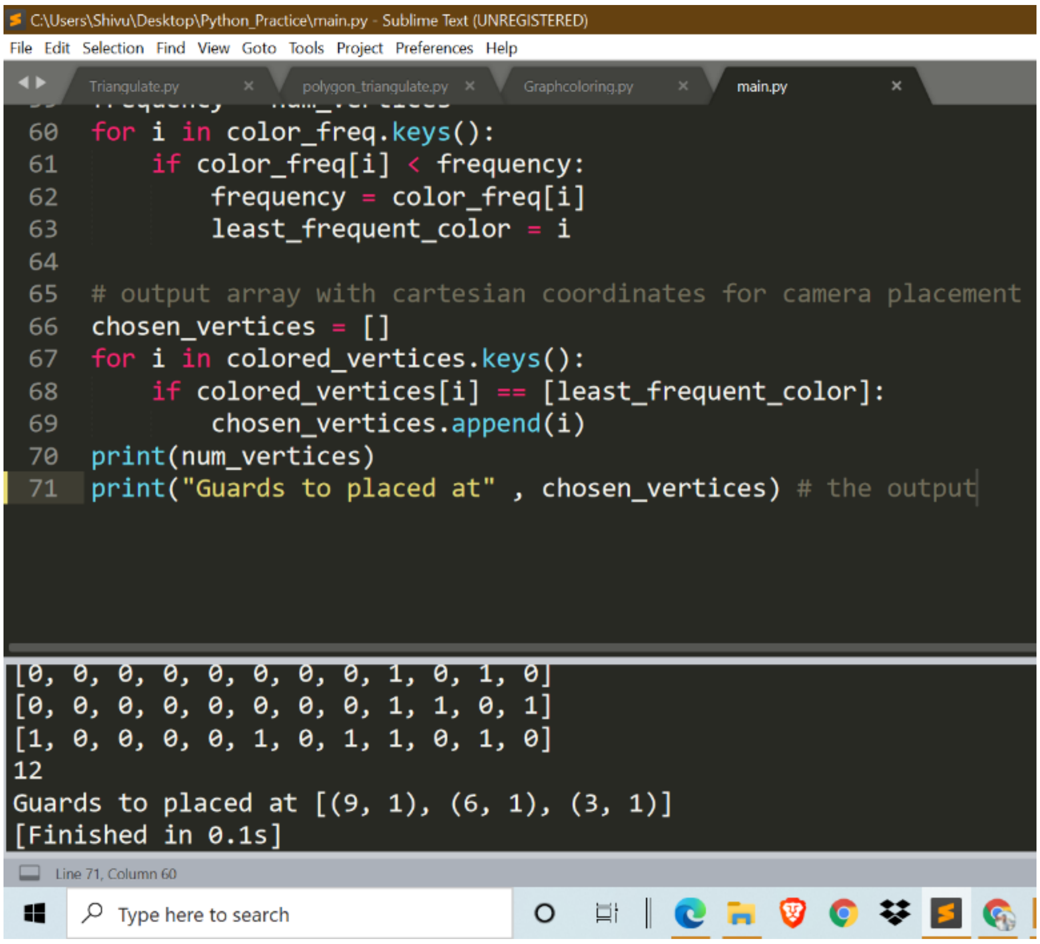
For an input of polygon: = [(0,0), (1,0), (1,1), (2,4), (3,1), (4,1), (5,4), (6,1), (7,1), (8,4), (9,1), (10,1), (10,0)] # n = 5

The guards will be placed: [(9, 1), (6, 1), (3, 1)]

O (n log n) running time achieved

- Reflection

The art gallery problem was successfully solved in O (n log n) time which is an optimum time period's


C:\Users\Shivu\Desktop\Python_Practice\main.py - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
Triangulate.py
polygon_triangulate.py
Graphcoloring.py
main.py
Line 71, Column 60
Type here to search

```
for i in color_freq.keys():
    if color_freq[i] < frequency:
        frequency = color_freq[i]
        least_frequent_color = i

# output array with cartesian coordinates for camera placement
chosen_vertices = []
for i in colored_vertices.keys():
    if colored_vertices[i] == [least_frequent_color]:
        chosen_vertices.append(i)
print(num_vertices)
print("Guards to placed at" , chosen_vertices) # the output
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1]
[1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0]
12
Guards to placed at [(9, 1), (6, 1), (3, 1)]
[Finished in 0.1s]
```

Figure 5: Results