4.2 Q2

To find a minimum cost sensor placement plan that covers a 500mx500m surveillance area.Given the following conditions and resources:

1. Access to three types of omnidirectional sensors:
   - Network is tasked with monitoring 17 targets.
   - Type S1 with a detection range of 100m and cost 300 units.
   - Type S2 with a detection rage of 70m and cost 170 units.
   - Type S3 with a detection range of 30m and cost 65 units

4.2.1 Solution:

The task of devicing a function to come up with a minimum cost sensor placement plan that covers a 500mx500m surveillance area for 17 randomly placed target, with the above given resources and constraints was performed using the Python scripting language. The following section describe the implementation and results obtained.

1. **Sensor Model :** Each sensor was modelled by defining a Sensor class that has important parameters that define the nature of the sensor.

```
class Sensor:
    def __init__(self, width, height, sensorType, sensorRange, sensorCost,
        sensorId):
        self.sensor_type = sensorType
        self.sensor_id = sensorId
        self.sensor_cost = sensorCost
        self.pos_x = random.randint(0, 500)
        self.pos_y = random.randint(0, 500)
        self.range = sensorRange
        self.coverage = 0
        self.coverage_list = [] #The targets covered by the sensor
        self.sensor_present = True
        self.sensor_area = round((math.pi * ((sensorRange)**2)), 2)
        self.used_status = True
        pass
```

   The Sensor Type, Range, and Cost is defined using the below enums.

```
class SensorType(enum.Enum):
    Type_s1 = 1 # Cost 300
    Type_s2 = 2 # Cost 170
    Type_s3 = 3 # Cost 65
```

```
class SensorRange(enum.Enum):
    Type_s1 = 100
    Type_s2 = 70
    Type_s3 = 30
```

```
class SensorCost(enum.Enum):
    Type_s1 = 300
    Type_s2 = 170
    Type_s3 = 65
```

2. **Target Model :** Each target was modelled by defining a Target class that has important parameters that define the target.

```
class Target:
    def __init__(self, width, height, targetId):
        self.pos_x = random.randint(0, width)
        self.pos_y = random.randint(0, height)
        self.target_id = targetId
        self.coverage = 0
        self.coverage_list = [] # The number of sensors the cover the target
        pass
```

3. **The Layout:** The layout here is used to define the region in which the sensors are places and has functions that are used to evaluate and optimize the sensor placement according the given constraints.The constructors initializes with basic properties of the layout.

```
class Layout:
    def __init__(self, width, height, sensorCount, targetCount, kCoverage):
        self.width = width
        self.height = height
        self.k = kCoverage
        self.sensor_count = sensorCount
        self.target_count = targetCount
        pass
```

   (a) **Generating Targets:** 17 sensors are generated at random positions within the 500m x 500m layout.

```
def generate_targets(self):
        for a in range(0, self.target_count):
            target = Target(self.width, self.height, a)
            target_pos = [target.pos_x, target.pos_y]
            target_dict[a] = tuple(target_pos)
            target_dict_type[a] = target #Dictionary holding the 17 targets
            target_data.append(target)
```

   (b) **Generating Sensors:** Random number of sensors are choosen for each type and each of them are placed at random placement, the following parameters enable different modes of placement.

   - **sensorType:** To set the type of sensor to be placed randomly.
   - **combination:** Set to TRUE to allow random placement of sensor of each type.
   - **calcPlacement:** Set to TRUE to enable sensor placement at each target.Set above two parameters to false to enable sensor placement of type mentioned by the *sensorType* parameter.

```
def generate_sensors(self, sensorType, combination, calcPlacement, targetDict):
        if ((calcPlacement == False) & (combination == False)):
         .
        elif combination == True:
         .
        elif calcPlacement == True:
            .
        pass
```

(c) **Calculating Coverage:** The Coverage of the network is calculated according the function given. Setting the *finalCovList* as true will return the total list of targets covered in the network.

```
def calculate_coverage(self, sensorDict, targeDict, finalCovList):
        for tD in targeDict.items():
            for sD in sensorDict.items():
                if sD[1].used_status == True:
                    if sD[1].sensor_present == True:
                        status = self.calculate_distance(tD[1], sD[1])
                        if status:
                            tD[1].coverage_list.append(
                                sD[0]) if sD[0] not in tD[1].coverage_list else
                                    tD[1].coverage_list
                            sD[1].coverage_list.append(
                                tD[0]) if tD[0] not in sD[1].coverage_list else
                                    sD[1].coverage_list
                    else:
                        if (sD[0] in tD[1].coverage_list) & (tD[0] in
                            sD[1].coverage_list):
                            tD[1].coverage_list.remove(sD[0])
                            sD[1].coverage_list.remove(tD[0])
            tD[1].coverage = len(tD[1].coverage_list)
            sD[1].coverage = len(sD[1].coverage_list)
        if (finalCovList == True):
                .
            return final_covered
```

(d) **Calculating Cost:** The cost of the network to be optimized using the below function.

```
    def cost_function(self, sensorDict, targetDict):
        coverage_cost = 0
        for tD in targetDict.items():
            for a in tD[1].coverage_list:
                if sensorDict[a].sensor_present == True:
                    coverage_cost = coverage_cost + sensorDict[a].sensor_cost
        return coverage_cost
```

(e) **Simulated Annealing:** At each iteration of the Simulated Annealing algorithm we determine th random choice of the sensor that would be disabled to analyse the effect of the cost of he network, If the cost of the network is less than the previous state/-configuration we accept the change. However, if the change is not better we accept if the temperature is high this way by hill climbing we don't get stuck at a local minima.

This is done with some probability and eventually we will get to the global minima with carefull consideration of the values.
In the code given below:

- **init_temp:** Is the initial hot temperature when the optmization algorithm starts.
- **final_temp:** The final temperature at which the optmization settles to.
- **cost_trend:** Holds the trend of the cost variation.
- **alpha:** The learning parameter.

```
def optmize_sa(self, sensorDict, targetDict):
        init_temp = 90
        final_temp = 2
        no_of_iterations = 0
        alpha = 0.1
        current_temp = init_temp
        cost_trend = []
        atleast = 0
        while current_temp > final_temp:
            no_of_iterations += 1
            self.calculate_coverage(sensorDict, targetDict, False)
            prev_cost = self.cost_function(sensorDict, targetDict)
            random_sensor = random.choice(list(sensorDict.keys()))
            if sensorDict[random_sensor].used_status == True:
                sensorDict[random_sensor].sensor_present = False
                cov_list = self.calculate_coverage(
                    sensorDict, targetDict, True)
                if (len(cov_list)/self.target_count)*100 >= 99:
                    new_cost = self.cost_function(sensorDict, targetDict)
                    cost_difference = (prev_cost - new_cost)
                    coverage_condition = True
                    for a in targetDict.items():
                        if len(a[1].coverage_list) <= self.k:
                            pass
                        else:
                            coverage_condition = False
                    if coverage_condition:
                        if cost_difference > 0:
                            atleast += 1
                            print("No of Iter_"+str(no_of_iterations) +
                                  "----"+str(new_cost))
                            cost_trend.append(new_cost)
                        else:
                            if random.uniform(0, 1) < math.exp(-cost_difference /
                                current_temp):
                                sensorDict[random_sensor].sensor_present = True
                        current_temp -= alpha
                    else:
                        sensorDict[random_sensor].sensor_present = True
                        if ((no_of_iterations > 10000) & (atleast == 0)):
                            break
                else:
                    sensorDict[random_sensor].sensor_present = True
        if atleast == 0:
            print("Unable to optimize for coverage : "+str(self.k))
```

```
        else:
            saName = "After_SA"
            generate_graph(cost_trend, 'SA_Swap_Trend', 'SA Swap Trend', True)
            generate_map(sensorDict, targetDict, saName, True, 'SA Placement')
            print("Final Cost "+"----" +
                  str(self.cost_function(sensorDict, targetDict)))
        pass
```

(f) **Generating Graphs:**In this module we generate two types of graphs. Graph to generate the trend of the cost of the network for each iteration of the Simulated Annealing algorithm.

```
def generate_graph(costTrend, name, titleName, showGraph):
    plt.savefig(name, bbox_inches='tight', dpi=150)
    if showGraph == True:
        plt.show()
    pass
```

Map that shows the target and sensor in the Layout

```
def generate_map(sensorDict, targetDict, name, showMap, titleName):
    plt.savefig(name, bbox_inches='tight', dpi=150)
    if showMap == True:
        plt.show()
```

4. **Driver Code:** The following is the driver code

```
def main():
    random.seed(1)
    sensor_layout = Layout(500, 500, 600, 17, 5)
    sensor_layout.generate_targets()
    # sensor_layout.print_location()
    sensor_layout.print_targetDict()
    # For a Combination of sensors
    sensor_layout.generate_sensors(1, True, False, target_dict_type)
    sensor_layout.print_sensorDict(1, True, False)
    finalList = sensor_layout.calculate_coverage(
        placement_dict_type_comb, target_dict_type, True)
    print(finalList)
    generate_map(placement_dict_type_comb,
                 target_dict_type, 'Layout_First.png', True, 'Initial Random
                     Placement')
    cost_of_ntwrk = sensor_layout.cost_function(
        placement_dict_type_comb, target_dict_type)
    print(cost_of_ntwrk)
    sensor_layout.remove_unused(placement_dict_type_comb, target_dict_type)
    sensor_layout.optmize_sa(placement_dict_type_comb, target_dict_type)

if __name__ == '__main__':
    main()
```

4.2.2 Results:

Case 1:
Considering the following inputs to the Layout Model

- **Layout Width :** 500
- **Layout Height :** 500
- **Targets :** 17 Nos.
- **Coverage:** 2 Nos.
- **Sensors**
  - **Type S1 :** 8 Nos.
  - **Type S2 :** 17 Nos.
  - **Type S3 :** 89 Nos.
- **Optimization Parameters:**
  - **Initial Temperature:** 90
  - **Final Temperature:** 10
  - **Alpha:** 0.5

1. **Initial random placement of sensors**

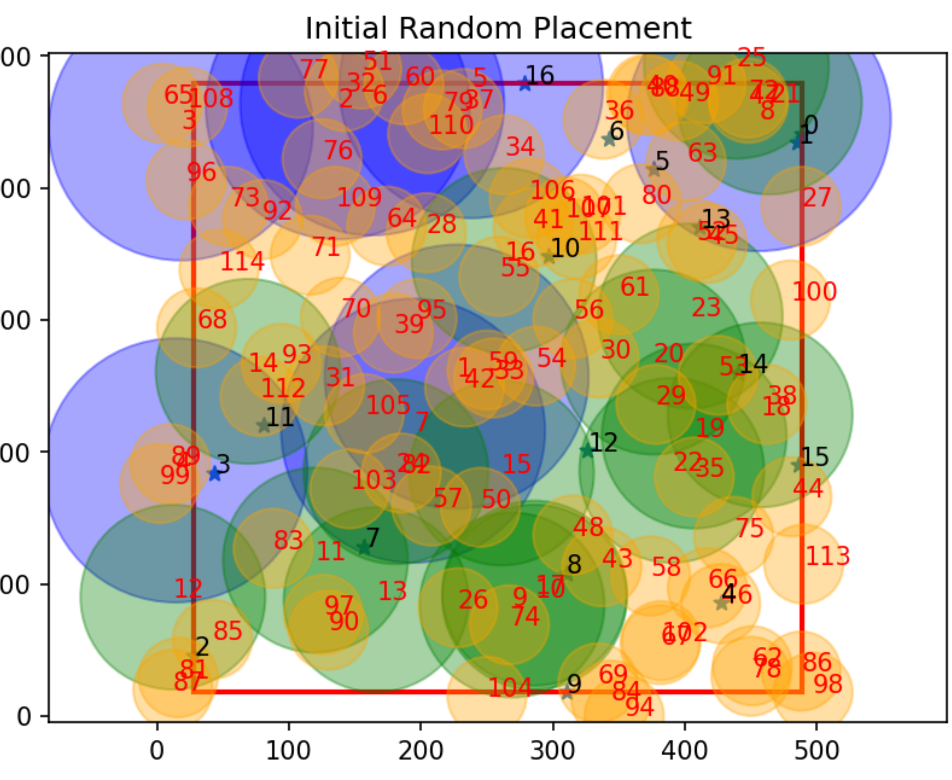

Figure 1: Initial Random placement

2. **Removing invalid sensors**

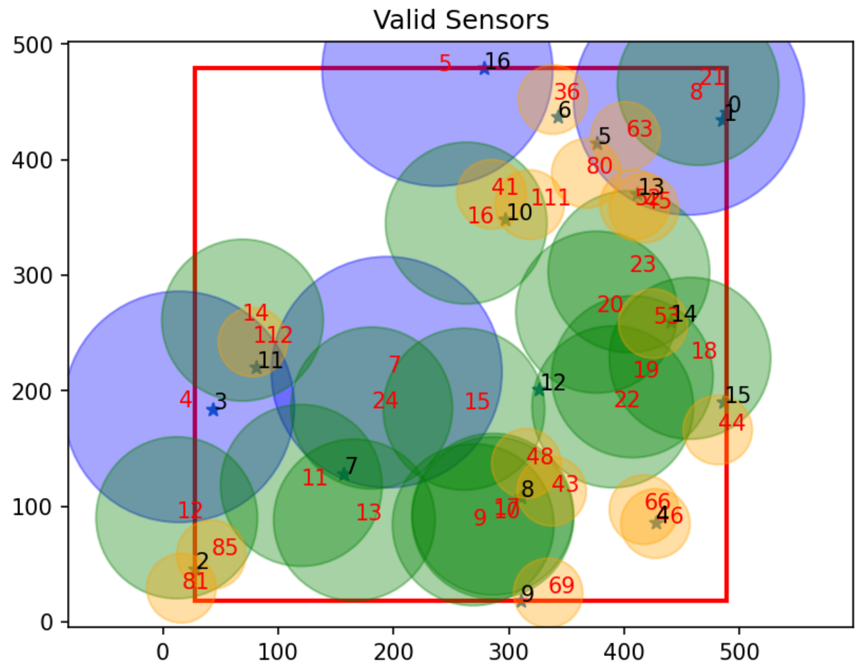

Figure 2: Removal of sensors with zero covered targets

3. **After simualted annealing**

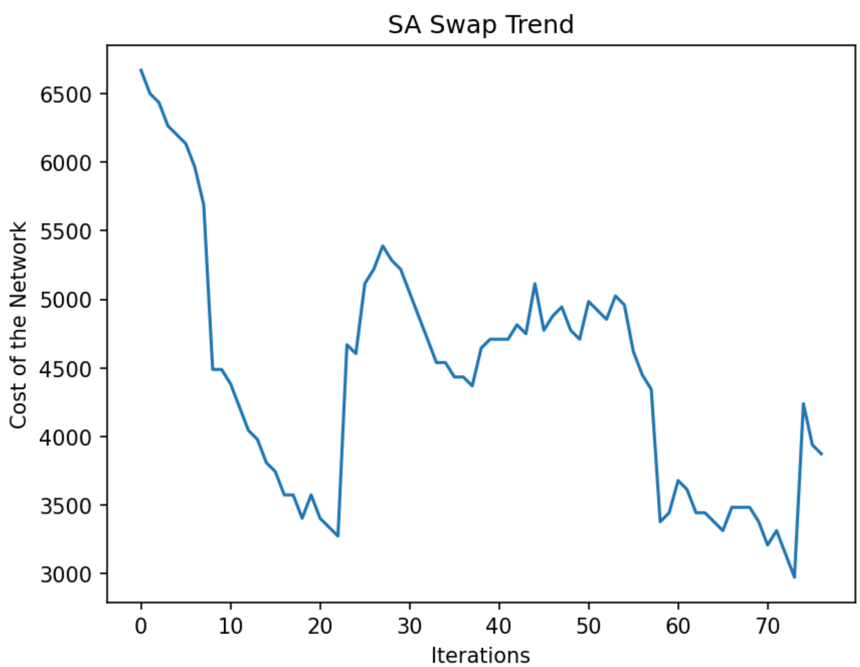

Figure 3: Trend of the network cost

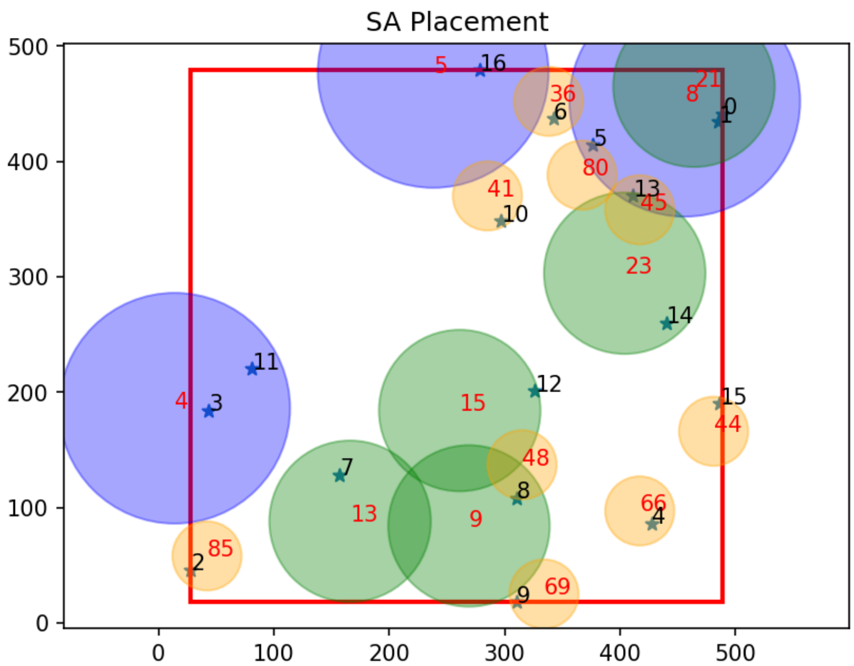

Figure 4: Final SA placement

Case 2:

- **Layout Width :** 500
- **Layout Height :** 500
- **Targets :** 17 Nos.
- **Coverage:** 2 Nos.
- **Sensors**
  - **Type S1 :** 2 Nos.
  - **Type S2 :** 2 Nos.
  - **Type S3 :** 500 Nos.
- **Optimization Parameters:**
  - **Initial Temperature:** 90
  - **Final Temperature:** 0.01
  - **Alpha:** 0.1

1. **Initial random placement of sensors**

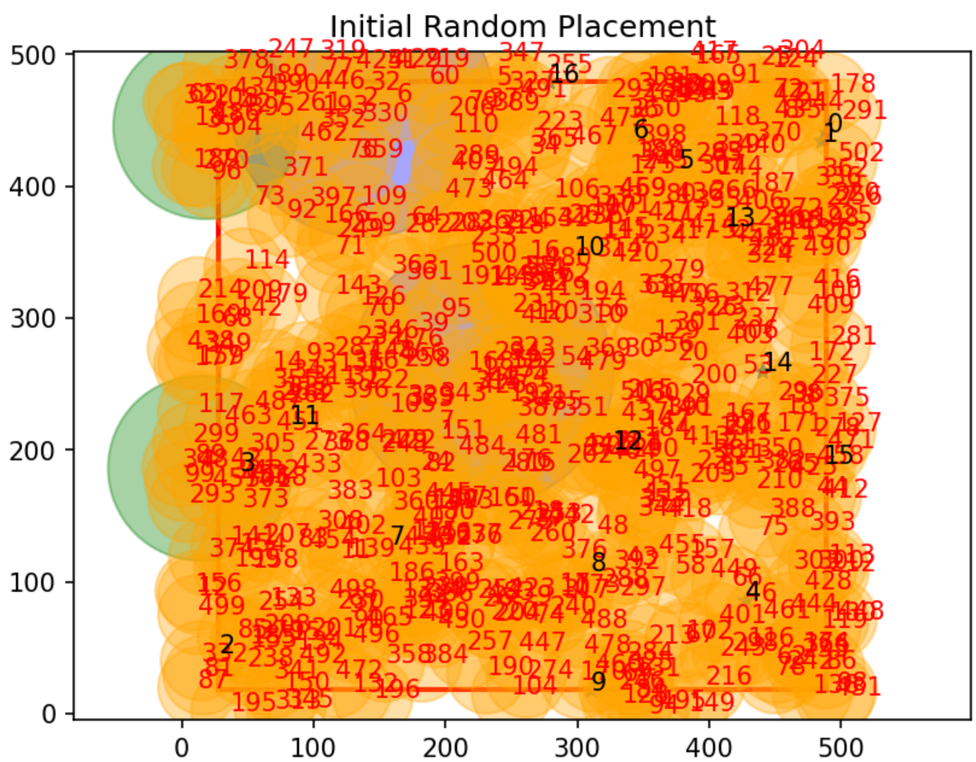

Figure 5: Initial Random placement

2. **Removing invalid sensors**

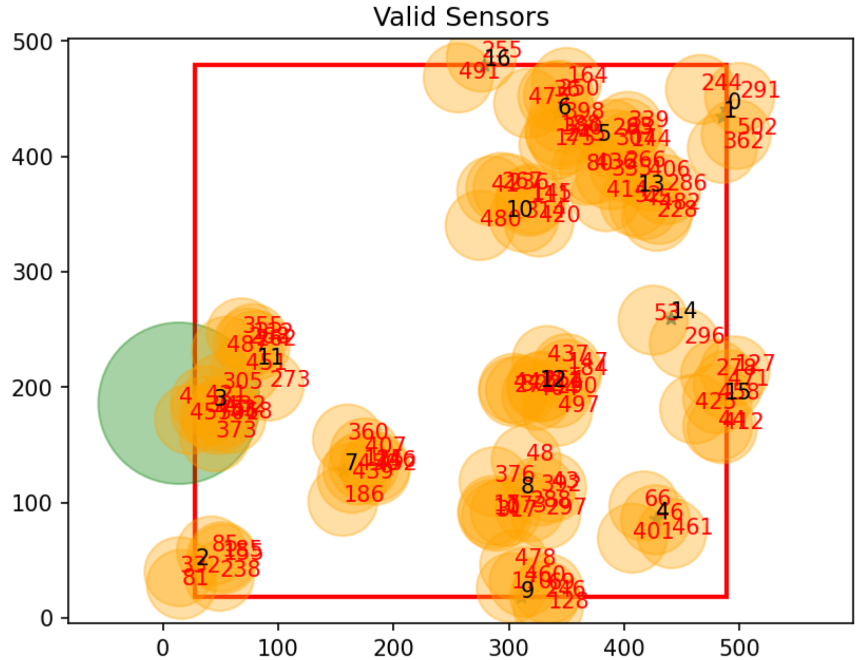

Figure 6: Removal of sensors with zero covered targets

3. **After simualted annealing**

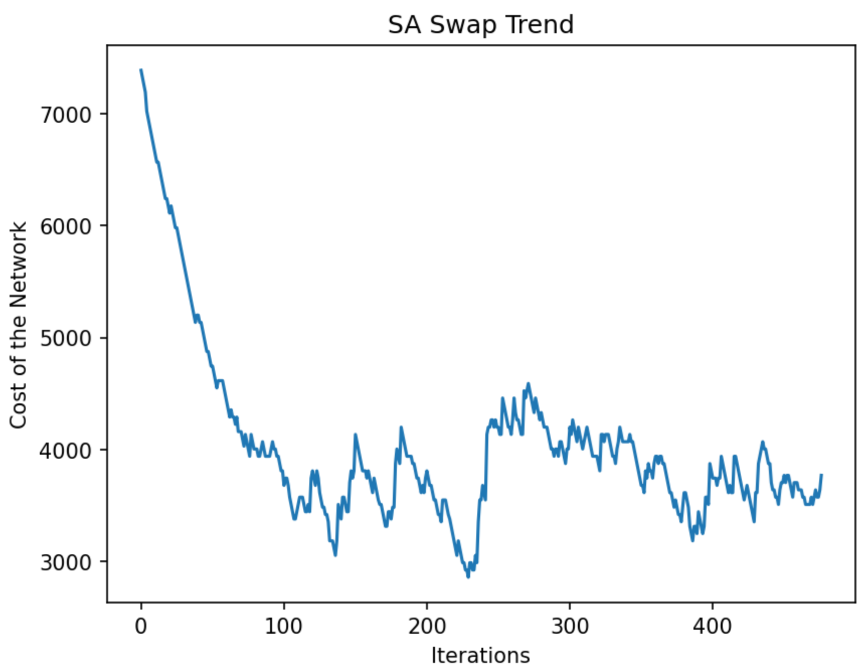

Figure 7: Trend of the network cost

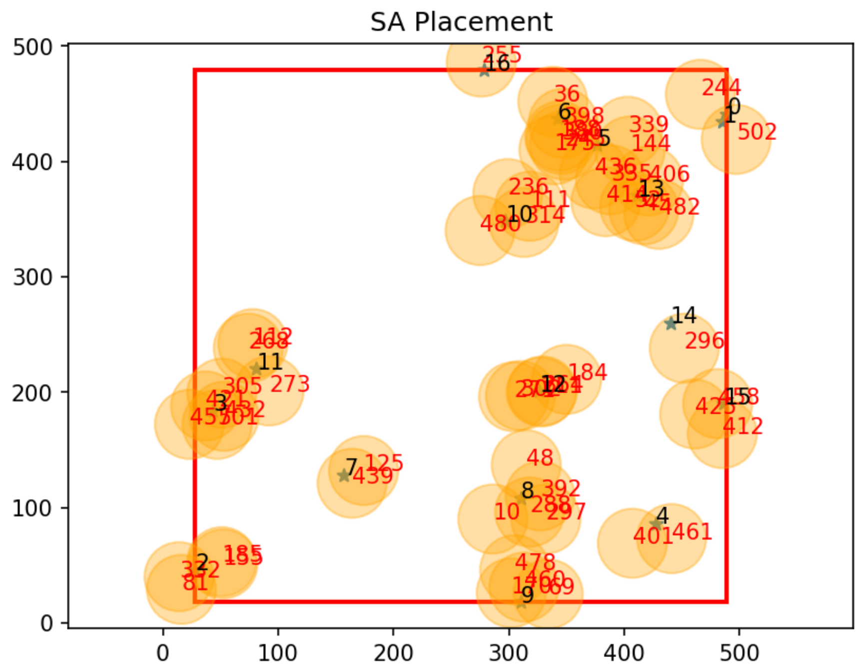

Figure 8: Final SA placement

4.2.3 Observations:

The Simulated Annealing optimization algorithm was run under two cases:

Case 1:

- Random number of sensor of each type
- Higher Final temperature and greater learning rate
- Lesser number of sensors

Case 2:

- Higher number of low cost sensors
- Lower Final temperature and smaller learning rate
- Larger number of sensors

Under case 1 the sensor placement took smaller time to be optimized to get maximum coverage of targets, and with case 2 the algorithm takes a lot of time but reaches a minimum cost but performs hill climbing frequently and settles a lesser cost. With carefull selection of learning and the temperature parameters the cost of the network can be minimized but at the cost of higher runtime.