

CmpE 524 Progress Report 2

Ozan Oytun Karakaya

April 2024

1 Introduction

In this introduction section, I will go over the problem and the approach I use in general in order to do a re-cap of the context stated in the first progress report. In this problem, the objective is to cover the maximum possible area with sensors stated with their sensing ranges within the given area with obstacles placed in it and create a WSN (Wireless Sensor Network) in that area for monitoring a particular phenomenon with sensors.

The area where the sensors will be deployed on is displayed as a rectangular area consisting of unit 1x1 squares (which is stated as cells in the problem solution, these 1x1 unit squares will be stated as cells in this document from now on) with the given lengths of height and width in terms of unit cells and it is projected on to Cartesian coordinate system where bottom-left (first) cell's center is on the origin. This monitoring area includes obstacles which are opaque to sensors' sensing ranges and they are displayed with black (as a visualization) cells in the area. It is important to note that sensors are not placeable onto this black cells (A visual representation of a sample problem is given below). Also sensors have to stand at a given minimum distance between each other to display interference problems about their sensing abilities as a constraint in the problem. Last but not least about problem formulation, a cell is accepted as covered if any sensor is able to reach to that cell's center point within its sensing range and all cells have their IDs as the formula below.

$$cell_ID = cell_y_coordinate \times area_width + cell_x_coordinate$$

I have created a solution finding optimal points for sensors by creating a manual solution which can be defined as a simple greedy algorithm using Python. I will state the heuristics and assumptions I have made in the solution in the section below.

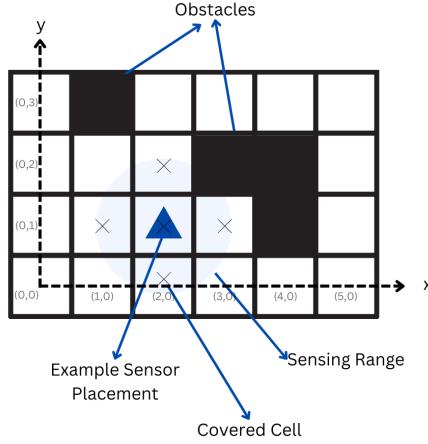


Figure 1: Example Visualization of The Problem

2 Heuristics and Assumptions

I will be mentioning about the assumptions and the heuristics that I have come up during the solution process under this section. Heuristics play a significant role in the structure of my problem, which are explained below.

1. Placement of Sensors into Cells

Since any cell will be accepted as covered if its center is sensed by a sensor, also the placement of sensors onto the centers of the cells can help to generate an optimal solution, especially considering the fact that sensing radius of the sensors will be given in terms of positive coefficient of unit length of 1 in coordinate system.

2. Early Placed Sensors' Distance

Since we are using a greedy algorithm with a minimum separation length constraint between sensors, chosen cells to place first sensors are picked distant to each other to cover more possible area later. Selection logic is explained under below sections.

Along with the heuristics, I would like to state some assumptions that are not made clear in the first progress report in the remaining part of this section.

1. Separation Length Between Sensors

Since separation length displays the interference problem about sensors' sensing abilities, separation length in the problem is stated with a positive coefficient of sensing radius in the interval [1.0, 2.0].

2. Black Cells' Corners

A sensor can sense a cell if line segment from the sensor to that cell's center does not split any black cell into two parts, thus corners of black cells are transparent to sensing.

3. Obstacles Consist of Black Cells

Obstacles are structured with sequentially merged black cells in one of four directions, up-down-left-right, for each length of the obstacle. Maximum length of an obstacle in terms of cells is limited to minimum of height and width value's half, floored to integer. Reason behind this is generating areas with distributed smaller obstacles compared to whole area; however, this assumption can be updated as inline with real area's conditions anytime.

3 Solution Formulation

I will breakdown the solution to its fundamental steps under this section.

1. Initialize the Grid

With the given width and height values, the grid is firstly initialized. As stated above, all cells have their sequentially given IDs and the cells that reside on an obstacle are also initialized here as black IDs. Black IDs can be manually given as wished or it can be generated by the program using uniform distribution for their locations, shapes and sizes by just specifying the number of obstacles (obstacles are referred as rocks in program).

2. Recognize Sensible IDs

Each cell is kept with an cell object in the program. Since black cell IDs are obtained at step above, sensible IDs from a cell's center obeying the constraints, if a sensor placed onto that cell, are calculated and put in a set. Later on, this set is updated for every sensor placement since program does not try to cover an already covered cell.

3. Creation of Cell Priority Queue

Here, the main priority queue for cells is created. The greedy algorithm I use, picks the top of this queue, most optimal cell according to the algorithm, at each iteration step. The comparison logic for cell objects puts the cell with greater number of sensible cells higher in the queue between two cell objects. If sensible cell number is equal between two cells, then the one with greater distance to the closest placed sensor is put higher in the priority queue, aligned with the second heuristic stated above.

4. Place Possible Maximum Number of Sensors

From now on, the algorithm, which will be explained the section below, iterates over the priority queue for choosing most optimal cells for placing a sensor. It places as many sensors as possible according to problem

constraints and its formulation. After this step finishes, the solution also finishes. The algorithm may not cover all of the non-black cells even it is possible with the allowed maximum number of sensors because of the fact that already placed first sensors does not allow any new sensor placement due to separation length constraint. The algorithm could be well improved by creating deviations from first placed sensors and creating space for adding new ones under the separation constraint; however, it was a challenge for me to even formulate this much of the solution.

4 Implemented Algorithm

Under this section, I will be explaining main algorithms that creates the solution with their pseudocodes and their code snippets in written in Python language. However, due to fact that it is much larger in pseudocode and snippet, I will be explaining verbally how sensible IDs are calculated from a cell's center.

1. Sensible ID Calculation

As it is stated in the above section, sensible IDs for each cell is firstly calculated if a sensor would be placed on each cell's center without considering black cell IDs, obstacles. Then, the IDs that can be blocked by black cells are extracted from the firstly calculated sensible ID set within the sensing range. The algorithm that determines which cells should be extracted uses slope intervals at black ID's distance within the sensing range of a sensor if it is placed on any cell's center. For each cell inside the sensing range, it is checked whether controlled cell's relative slope the current cell is in the interval of blocked regions under the distance. Visual explanation (Figure 2) displays the algorithm sufficiently below

2. Random Obstacle Generation

If it is not provided manually by providing the black cell IDs, an algorithm that utilizes random uniform distribution can generate obstacles distributed inside the area. The procedure creating these random obstacles as follows:

- Get the number of obstacles will be provided from the user
- For each obstacle, select random start cell ID
- Get random length for the obstacle
- For each unit length of 1, select a random direction between possible next cell, can vary where the current cell ID resides according to borders and corners
- Merge the selected cell ID, iterate until getting to random selected obstacle length
- Repeat this for every number of obstacles

Pseudocode for this algorithm is below with the name Algorithm 1.

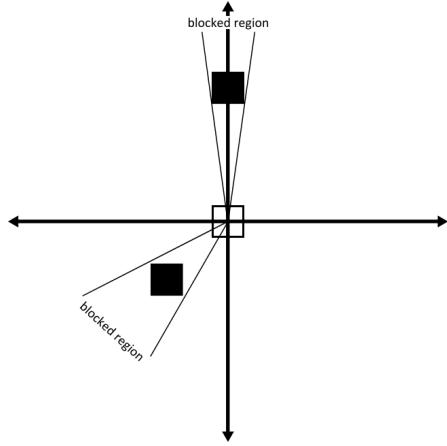


Figure 2: Blocked Regions for Each Cell Calculation

Algorithm 1 Random Obstacle Generation

```

1: random_black_ids ← {}
2: obstacle_number ← get_user_input()
3: for i in range(obstacle_number) do
4:   random_start_id ← get_random_starting_id()
5:   random_black_ids.add(random_start_id)
6:   obstacle_length ← get_random_length
7:   for length in range(obstacle_length-1) do
8:     random_black_ids.add(get_next_sequential_random_black_id())
      {Next: up-down-left-right}
      {Direction for next black cell ID depends on corners and borders of
       the area}
9:   end for
10: end for

```

3. Sensor Placement Algorithm As it is stated in above sections, a greedy approach is taken for sensor placements. All cells are ordered in a priority queue and the most optimal one according to ordering, which have the largest number of sensible IDs and then the most distant one to already placed sensors, is popped from the head of the queue. For all cells, it is checked whether it suits to place a sensor to that cell under the constraints, if not algorithm continues iterating by keeping the count of infeasible cells, if so it places a sensor there. If the infeasible cell count is more than the number of cells that is not black or sensor placed, algorithm stops. Also, algorithm can stop if all of the area is covered inside the main while loop. Pseudocode for this algorithm is below.

Algorithm 2 Sensor Placement Algorithm

```

1: temp_cell_list ← []
2: rejected_cell_count ← 0
3: while maximum_number_of_sensors > 0 and
   rejected_cell_count < Number of Available Cells do
4:   if Whole Area is Covered then
5:     break
6:   end if
7:   top_cell ← cell_queue.pop()
8:   if is_placeable(top_cell) then
9:     sensor_placed_ids.add(top.cell)
10:    covered_ids.set_union(top.cell.sensible_ids)
11:    Update All Cell's Sensible ID Sets
12:    Dequeue All Elements in cell_queue and Add to temp_cell_list
13:    Push All Elements in temp_cell_list to cell_queue {re-order
      the queue}
14:    temp_cell_list.empty_list()
15:    maximum_number_of_sensors ← maximum_number_of_sensors-1
16:   else
17:     rejected_cell_count ← rejected_cell_count+1
18:     continue
19:   end if
20: end while

```

The code snippets in Python language is below for the Algorithm 1 and Algorithm 2.

Algorithm for Random Obstacle (Rock) Generation : Algorithm 1

```

1  import random
2  import math
3
4  def uniform_select(a, b):
5      x = random.uniform(a, b+1)
6      if(x != b+1):
7          return math.floor(x)

```

```

8         else:
9             return math.floor(random.uniform(a, b))
10
11     def update_coordinates(x, y, movement):
12         x += movement[0]
13         y += movement[1]
14         return x, y
15
16     max_rock_length = math.floor(min(width, height)/2)
17     random_black_ids = set()
18     rock_number = int(input("Please specify the number of rocks:"))
19
20     for i in range(rock_number):
21         rock_length = uniform_select(1, max_rock_length)
22         first_black_id = uniform_select(0, width*height-1)
23         random_black_ids.add(first_black_id)
24         x,y = get_coordinate(first_black_id)
25         movement = [0,0]
26         for j in range(rock_length-1):
27             if(x == 0):
28                 if(y == 0):
29                     movement = [[1,0],[0,1]][uniform_select(0,1)]
30                 elif(y == height-1):
31                     movement = [[1,0],[0,-1]][uniform_select(0,1)]
32                 else:
33                     movement = [[1,0],[0,1],[0,-1]][uniform_select
34 (0,2)]
35                 elif(x == width-1):
36                     if(y == 0):
37                         movement = [[-1,0],[0,1]][uniform_select(0,1)]
38                     elif(y == height-1):
39                         movement = [[-1,0],[0,-1]][uniform_select(0,1)]
40                     else:
41                         movement = [[-1,0],[0,1],[0,-1]][uniform_select
42 (0,2)]
43                     else:
44                         if(y == 0):
45                             movement = [[1,0],[-1,0],[0,1]][uniform_select
46 (0,2)]
47                         elif(y == height-1):
48                             movement = [[1,0],[-1,0],[0,-1]][uniform_select
49 (0,2)]
50                         else:
51                             movement = [[1,0],[-1,0],[0,1],[0,-1]][
52 uniform_select(0,3)]
53         x, y = update_coordinates(x, y, movement)
54         random_black_ids.add(get_id(x, y))
55
56     random_black_ids = list(random_black_ids)

```

Algorithm for Sensor Placements : Algorithm 2

```

1     temp_cell_list = []
2     rejected_cell_count = 0
3     while(max_num_of_sensors > 0 and rejected_cell_count <
4           (width*height) - (len(black_ids) + len(sensor_placed_ids))):
5         if(len(covered_ids) == width*height - len(black_ids)):
6             break

```

```

7     top_cell = heapq.heappop(cell_queue)
8     if(is_placeable(top_cell.id)):
9         sensor_placed_ids.append(top_cell.id)
10        covered_ids = covered_ids.union(top_cell.sensible_ids)
11        update_sensible_id_lists()
12        # now dequeue and queue all cells that are not black
13        and not covered
14        while(cell_queue.__len__() > 0):
15            temp_cell_list.append(heapq.heappop(cell_queue))
16            for cell in temp_cell_list:
17                heapq.heappush(cell_queue, cell)
18            temp_cell_list.clear()
19            max_num_of_sensors -= 1
20        else:
21            rejected_cell_count += 1
22            continue

```

The auxiliary functions that I have used for both algorithms are not given in this documents, since it would create a non-readable format providing all functions here. Necessary functions' code snippets can be provided with source codes if wished.

5 Objective Function Evaluation

Here, the algorithm is a greedy algorithm constrained by maximum number of sensors and separation length between sensors. The objective function for this algorithm is to maximize the covered area with staying under this constraints. So it the function for evaluation the performance is simply:

$$Coverage = \frac{1}{N} \sum_{cell_id=0}^N f(cell_id)$$

$$N = area_width \times area_height - number_of_black_cell_ids$$

$$f(cell_id) = \begin{cases} 1 & \text{if cell.id is in covered.ids set} \\ 0 & \text{if not} \end{cases}$$

It is important to note that algorithm may not use the maximum number of sensors even it cannot cover all of the area due to its lack of ability to overcome the separation length constraint after placing some sensors. However, because of the fact that its objective function only focuses on the coverage area, the ratio of covered area is the only factor that is considered when evaluating the performance of the algorithm.

6 Performance Testing

In order to test the performance of this solution, I am planning to run a 2-to-the-k test with the listed parameters below.

- Obstacle Density of the Given Area (low number of obstacles vs high number of obstacles in same area)
- Separation Length of Sensor (low separation length constraint vs high separation length constraint)
- Sensing Radius of Sensors Compared to Border Lengths of Given Area (high ratio of sensing radius over minimum border length vs low ratio of sensing radius over minimum border length)

These parameters are the ones that I am planning to test for now, this list may be updated after revisions. After specifying the core parameters that defines the behavior of the solution the best, I will compare this solution with other solutions in the literature using these parameters. The solutions for coverage area in WSNs does not include the obstacle formation in general, they also include the problem of inter-communication between sensors; however, I will try to formulate and modify the inputs to that solutions so that the sensor deployment areas can behave similar to my deployment area with obstacles.