CmpE 524 Progress Final Progress Report

Ozan Oytun Karakaya

May 2024

1 Introduction

In this project, we have solved an Optimal Coverage Area problem in the context of Wireless Sensor Networks. Any area that will be deployed with sensors having circular identical ranges will be an input for our problem with opaque obstacles that is not transparent any sensing over them are placed on it. These obstacles can be placed randomly or configured manually as the case requires. Along with obstacles, a minimum separation length between placed sensors applies another constraint on the problem. Under this project, we have developed a greedy algorithm that tries to find most optimal places for sensors by placing them one by one, leading us to the maximum coverage possible on the area. More about the problem itself is covered in the other progress reports attached with this report. You can find the diagram describing the problem below.

In this report, the design created for problem and solution will be given first. Then, we will proceed with explaining the implementation details and performance evaluation. Lastly, future work that can be done will be mentioned shortly.

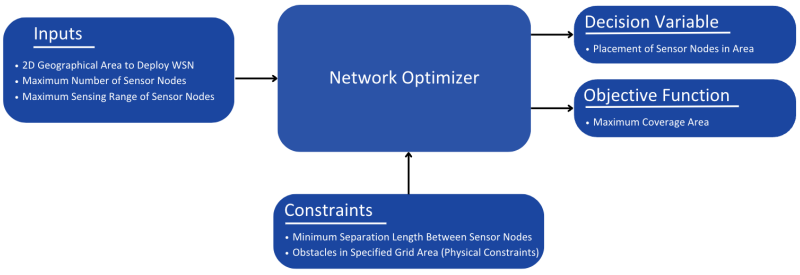

Figure 1: Problem Definition Diagram

2 Design

Under this chapter, formulation of the problem and the solution mechanism we have developed will be explained in detail.

2.1 Problem Formulation

Under this subsection, main lines of the problem formulation is summarized. Please refer to the Progress Report 2 for further details.

2.1.1 2D Input Area

Any 2D rectangular shaped area can be given as an input to the program. Input area is formulated as a grid where it consists of 1x1 unit squares, which will be referred as cells, representing $1m^2$. All cells have integer ID numbers starting from 0 where the first one's, Cell_ID:0, center is placed onto origin of the coordinate plane. ID's of the cells are calculated with the formula below:

$$cell\_ID = cell\_y\_coordinate \times area\_width + cell\_x\_coordinate$$

2.1.2 Obstacles

Obstacles in the grid are displayed with cells called black_ids inside the program. Obstacles consist of merged black_id cells, or just 1 cell, which are opaque to sensing abilities of the sensors meaning that a sensor can't sense any point behind of an obstacle even if that point is in the sensing range of the sensor. Please note that, corners of the obstacles are transparent to sensing.
These obstacles can be randomly generated throughout the area according to the rock_density variable, please refer to user manual attached, or can be provided manually by assigning values to black_ids list. Random Obstacle Generation Algorithm can be found in the Progress Report 2, it won't be mentioned here.

2.1.3 Sensing Mechanism

As it was stated in the Progress Report 2, an heuristic is used to determine where to put sensors, which is simply putting them into the centers of the cells. This heuristic is derived from the assumption we have made about the problem which is a cell is accepted as covered if any sensor can sense it's center while obeying the constraints. Thus, for each cell, another cells in the sensing range are calculated whether they are sensible or not by the constraints (blocking of obstacles).
This calculation is made as getting relative position for each cell in the sensing range, getting the relative slope of the line segment between the centers and comparing it to slope intervals that are blocked by obstacles. There are 8 slope intervals kept as a list with indices:

- 0: 1^{st} Quarter, 1: 2^{nd} Quarter, 2: 3^{rd} Quarter, 3: 4^{th} Quarter

- 4: Positive X-axis, 5: Positive Y-axis, 6: Negative X-axis, 7: Negative Y-axis

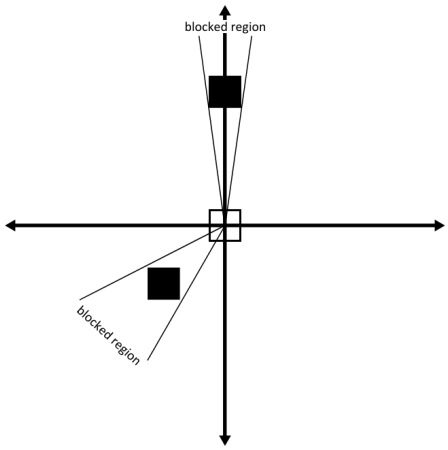

Figure 2: Blocked Regions for Each Cell Calculation

You can refer to the source code attached for implementing the necessary functions for this issue. Related functions are:

- `get_relative_slope_interval(id1, id2)`
- `sensing_range(id)`
- `find_blocking_intervals(id, black_id_list)`
- `is_sensible_in_range(id1, id2, block_interval_list)`

2.1.4 Coverage

Objective function, coverage, is calculated simply in the program just by iterating over all cells with their IDs and check whether they are covered or not. The formula used for calculating coverage:

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

$$N = area\_width \times area\_height - number\_of\_black\_cell\_ids$$

$$f(cell\_id) = \begin{cases} 1 & \text{if cell\_id is in covered\_ids set} \\ 0 & \text{if not} \end{cases}$$

2.2 Solution Mechanism

2.2.1 Queueing Logic for Cells

As it is stated in the 3.1.3, how many cells can be covered if a sensor is placed onto the center of each cell is calculated. Using this number, cells are put in a priority queue where the greedy algorithm utilizes by taking the head of the queue for each sensor placement if it is feasible. In this queue, if two cells have equal number of sensible cells, then distances to the closest placed sensor is used for sorting them. Comparison logic for cells which is used in constructing the priority queue is displayed with a flowchart below.

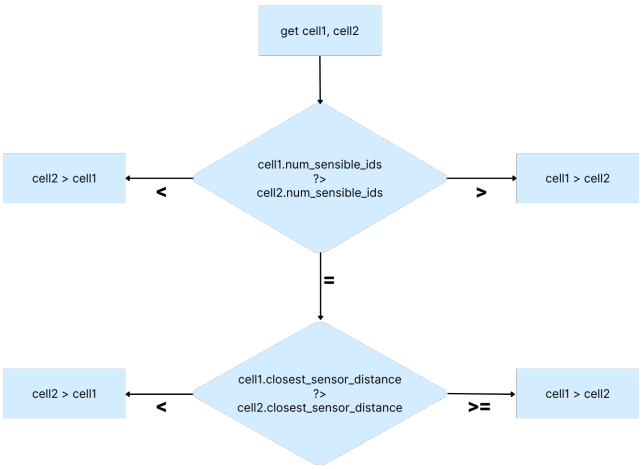

Figure 3: Cell Priority Queue Logic

2.2.2 Greedy Algorithm for Sensor Placements

We have used a greedy approach for sensor placements using the cell priority queue created. Basically, `max_num_of_sensors` variable limits the algorithm where the algorithm pops the head of the queue, tries the placement and rejects it if it is not feasible for each sensor placement trial. There is also a variable for counting the rejection number which used to understand whether the limit is reached due to constraints even if number of placed sensors is below the maximum number. Algorithm can also stop before reaching the maximum sensor number if whole area is covered, however, this is not the case mostly.
Flowchart describing the behavior of the algorithm is below, moreover, you can find more information about the algorithm in the source code attached and the Progress Report 2.

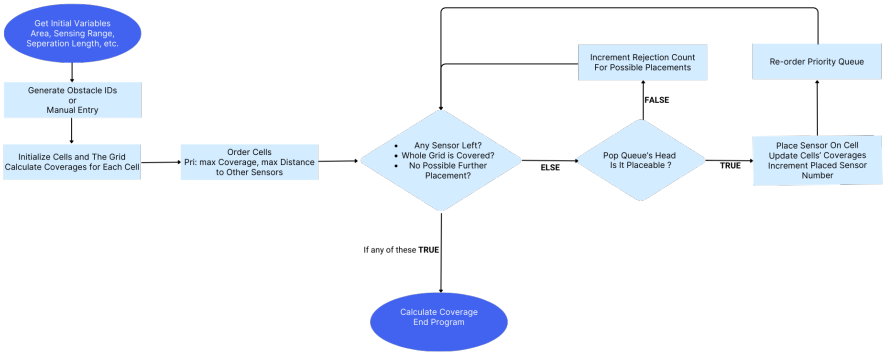

Figure 4: Greedy Algorithm for Sensor Placements

3 Implementation

We have used Python 3.12 for the implementation of the algorithm. For any run, modifying the beginning of the program where initialization part for the inputs and constraints is sufficient where it is explained in further detail in the attached user manual. Algorithm for each run depends on the size and the complexity of the given input area, however, since algorithm utilizes the Python data structures largely, runs including areas even up to 10^4 with %20 obstacle density takes less than a minute. Steps of the implemented solution is explained shortly below.

1. Imports & Assignments: First part of the program includes 3 imports which are `math` library, `random` library and the `heapq` library. Then, assignments of program variables are made except for `black_ids` which are handled later.

2. Auxiliary Functions: There are lots of functions defined inside the program according to design that is made in the section above. Most of them is used in the cell initialization part. Only `is_placeable(id)` and `update_sensible_id_lists()` functions are used in the main algorithm.

3. Initialization of Cells: Cells in the program are defined as Cell class objects since there are lots of data being held for each cell in the program, thus, these data are kept as fields in the cell objects. This is the part of the program where it is executed in %90 of all execution time since calculations for sensing are made in this part.

4. Obstacle Initialization: Density variable determining the number of obstacles is set in the first part which is used within in the Random Obstacle Generation part, here. Under random generation part, users can manually assign black_ids list as case-specific runs.

5. Main Algorithm: In this part, main algorithm is run with a while loop. It modifies the global variables such as `covered_ids` sets during the execution.

6. Results: After the algorithm finishes, results are printed to the console.

It is recommended to go over this section with analyzing the source code in parallel which is commented out properly according to the steps above.

4 Performance Evaluation

Under this section, a deep analysis conducted will be explained in the first subsection, then a comparison made with implemented random algorithm will be explained in the second subsection.

4.1 Performance Analysis

In order to understand the performance of the algorithm, first process required to be completed was to determine which parameters affects the objective function's value in a significant manner. Thus, we have run $\mathbf{2^k}$ Testing with the parameters that can be important for the execution.

- Rectangular **vs** Square Area: Tested with 50x200 & 100x100 Width & Height Values
- Low **vs** High Obstacle Density: Tested with %5 & %50 Obstacle Density Values - Input File for `black_ids` are attached with the name of `Obstacle Density Comparison Black IDs.txt` file.
- Low **vs** High Minimum Separation Length: Tested with `1*(sensing range)` & `2*(sensing range)`
- Low **vs** High Maximum Number of Sensors: Tested with 32 & 64 maximum number of sensors

It is important to note that, sensing range value for sensors are kept constant since we are testing with different values for maximum number of sensor. The reason behind this is the fact that sensing range and maximum number of sensors parameters are not independent for a constant area value.

Please note that, low number corresponds to the minimum number of sensors where the input area has no obstacles in it. High number is the double of the low number through an heuristic with the logic: for each cell pair; if there is 1 obstacle between then there should be 2 sensors to cover all cells between. Low and High values are for maximum number of sensors are calculated with the formula below.

$$Maximum\,Number\,of\,Sensors\,(Low) = \left\lceil \frac{Area}{\pi \times r^2} \right\rceil$$
$$Maximum\,Number\,of\,Sensors\,(High) = 2 \times \left\lceil \frac{Area}{\pi \times r^2} \right\rceil$$

Sensing range for sensors is constant on whole experiments equal to length of 10 units, can be thought as 10 meters as well.
You can see the results of the $\mathbf{2^k}$ Testing below.

Sensing Range (r) = 10

Height	Width	Obstacle Density	Minimum Separation Length (r)	Maximum Number of Sensors	Number of Placed Sensors	Coverage	Run Time (sec)
100	100	5%	1	32	32	85.80%	19.5
100	100	5%	1	64	64	99.76%	24
100	100	5%	2	32	21	61.34%	18.3
100	100	5%	2	64	21	61.34%	18.3
100	100	50%	1	32	32	83.86%	19.2
100	100	50%	1	64	64	99.08%	23.5
100	100	50%	2	32	21	59.40%	18.2
100	100	50%	2	64	21	59.40%	18.2
50	200	5%	1	32	32	83.45%	66.9
50	200	5%	1	64	64	99.71%	58.5
50	200	5%	2	32	23	64.65%	58.4
50	200	5%	2	64	23	64.65%	56.9
50	200	50%	1	32	32	82.40%	58.4
50	200	50%	1	64	64	98.56%	65.8
50	200	50%	2	32	22	60.54%	56.7
50	200	50%	2	64	22	60.54%	56.8

Figure 5: $\mathbf{2^k}$ Testing

Here, it is noted that first two parameters, rectangular shape of the area and the obstacle density, does not affect the behavior algorithm and the results in a significant way compared to `minimum separation length` constraint and the `maximum number of sensors` input parameter. Thus, these two parameters are analyzed in more detail in further testing.
An 100x100 input area with %20 Obstacle Density is used when analyzing these two parameters in detail. Input file for this obstacle configuration is provided along with this report with the name of `20 Percent Obstacle Density Configuration.txt`. Sensing range (r) is again kept constant with 10 units and the maximum number of sensors is set to 90 for making this parameter independent in the context of this experiment. For testing the minimum separation length constraint's effect, we have tested the algorithm with same inputs by incrementing the minimum separation length from **1*r** to **2*r** by 0.1 at each step.
For testing the maximum number of sensors parameter, same input setup is used with the exception of keeping the minimum separation length constant at **1*r**. Maximum number of sensors parameter is incremented one by one at each run. You can see the results of both experiments in the figure below.

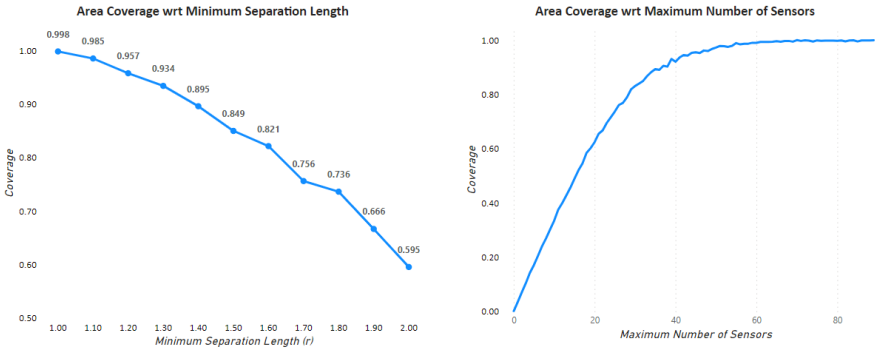

Figure 6: Detailed Testing with Significant Parameters

We can conclude that the results of these experiments showed an expected behavior. It can be simply analyzed that minimum separation length constraint have a large effect on the objective function's value. Moreover, we can see that decrease in the objective function value is larger in the steps closer to **2*r** than the steps closer to **1*r**.
As for maximum number of sensors parameter, objective function displays an horizontal asymptotic behavior as expected since the most covering cells are placed with sensors in the earlier and the later ones have slighter effect on the increase in the objective function value.

4.2 Solution Comparison

Our algorithm and the randomly implemented algorithms are compared under this subsection. Input setup we have used for this purpose is close to the setup we have used above, given below.

- 100x100 Input Area
- Obstacle Density: %20, `black_ids` list is given in the input file `20 Percent Obstacle Density Configuration.txt`
- Sensing Range (r): 10
- Minimum Separation Length: **1*r**
- Maximum Number of Sensors: 50

We have implemented two random solution algorithms. First one is an algorithm that is more primitive than the second one, where its purpose was to display the fact that generating feasible solutions according to constraints was a hard task to complete. It simply generates random cell IDs where the sensors are placed

and checks whether the solution obeys to the constraints or not. After running this algorithm *10 million times and getting no feasible results out of it*, we have created the second algorithm where it uses our functions for checking whether the randomly suggested ID is feasible or not. Flowcharts and Python codes for these two random solution algorithms are attached below.

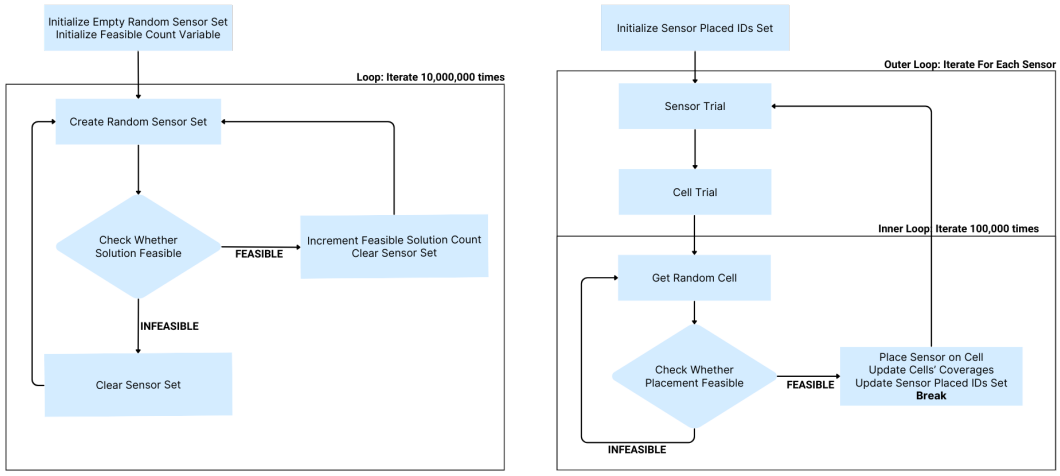

Figure 7: Random Solution Algorithms

First Random Solution Algorithm for Feasibility Testing

```
# Algorithm for checking the feasibility of random solution in
    10000000 trials
random_sensors = set()
infeasible_placement_count = 0
for i in range(10000000):
    while(len(random_sensors) < max_num_of_sensors):
        random_id = random.randint(0, width*height-1)
        random_sensors.add(random_id)
    sensor_placed_ids = list(random_sensors)
    for sensor_id in sensor_placed_ids:
        if(is_placeable(sensor_id)):
            continue
        else:
            infeasible_placement_count += 1
            break
    random_sensors.clear()
    sensor_placed_ids.clear()
print("Feasible placement rate of random sensor placements: %",
    100*(1-infeasible_placement_count/10000000))
```

Second Random Solution Algorithm for Solution Comparison

```
# Algorithm for checking random feasible placements in 100000 trial for each cell
feasible_placement_count = 0
for i in range(max_num_of_sensors):
    for trial in range(100000):
        random_id = random.randint(0, width*height-1)
        if(is_placeable(random_id)):
            sensor_placed_ids.append(random_id)
            x,y = get_coordinate(random_id)
            covered_ids = covered_ids.union(grid[y][x].sensible_ids)
            update_sensible_id_lists()
            feasible_placement_count += 1
            break
print("Number of placed sensors: ", feasible_placement_count)
print("Coverage: ", calculate_coverage(), "%")
```

Since the second random solution algorithm is on steroids with some feasibility functions of ours, it generated sufficiently better results to compare the objective function values with our algorithm. We have tested our algorithm along with this second algorithm, with the defined input setup in the beginning of this subsection. Figure for comparing the objective function values is given below.

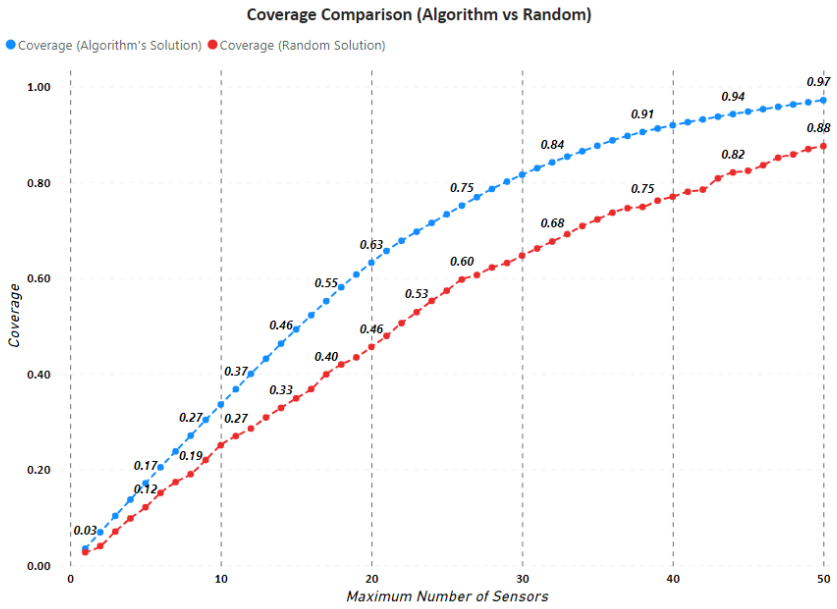

Figure 8: Comparison with Random Solution Algorithm

We can conclude that, our algorithm performs %25 better in the mid-section of maximum number of sensor parameter. In the left and right ends, since we do not expect so much difference, for the sensor numbers higher than 50, since feasibility constraints limits the options for cell selection.

5 Future Work

This greedy algorithm generates quick, effective results for sensor deployment decision in the context of Wireless Sensor Networks. It can be embedded into some hardware where agricultural studies or even maybe in the field of military forces. However, we are aware that this algorithm creates a good initial solution to this problem rather than generating the best optimal solution. We think that this algorithm can be well-updated with triggering random deviations from this initial solution phase through iterations, since minimum separation length constraint limits the algorithm's performance severely especially around the values closer to **2*r**. With the help of this deviations, more sensors can be placed inside the area while keeping the sensor number under the maximum sensor number constraint. In addition, we think our solution to implementing feasibility constraints is a light and effective process which can be still utilized.

6 Conclusion

In this report, we have deeply analyzed and studied our problem and the solution we have come with to this problem in the context of Optimal Coverage Area in Wireless Sensor Networks. After explaining the design of the problem in detail first, we have explained our solution along with its logic. Lastly, we have gone through the process of performance evaluation for our algorithm. It is recommended to read this report along with the previous Progress Reports in order to have the understanding of full context with its development through the process.