EC ENGR 134, Fall 2021: Kellen Cheng (905155544) and Yu-Wei Vincent Yeh (005123289)
Project 1: Sensor Network Synthesis
Due: 4 November, 2021 at 11:59 PM

1 Problem Statement

The study of wireless sensor networks has drawn great interests within disciplines such as area monitoring, environmental detection, and space defense. Through an optimal placement of sensors, environmental obstructions of the signal could be mitigated while maintaining good connectivity and coverage - the connectivity and coverage that arise from this optimal placement are the critical figures of merits to this type of study. Thus, the goal of this project is to place the minimum number of sensors, where each sensor covers a 9x9 area, on a map of size 300x300, while simultaneously optimizing the network's connectivity and coverage. In this project, we created an efficient sensor placement algorithm that not only achieved a coverage that outperformed random placement methods by 35-40%, but also possessed a far superior connectivity.

2 Algorithm Description

2.1 Graph Class:

(a) **init**: The main data types stored in the Graph class include: "graph", a weighted adjacency matrix; "nodes", a dictionary mapping a node number (from 0 to number of nodes) to an actual coordinate; "reverse-nodes," a dictionary mapping coordinates to the node number.

(b) **add-node-edge and add-edge**: The add-node-edge function adds a node to the graph, with an optional parameter to add a weight corresponding to some edge. The add-edge function adds an edge with some weight between existing nodes. These functions add nodes and edges by setting the ij and ji elements in the "graph" weighted adjacency matrix to some weight, then updating the "nodes" and "reverse-nodes" dictionary.

(c) **max-span**: The max-span function aims to find the maximum spanning tree in the complete graph that we constructed. It utilizes Prim's algorithm, which first initializes a tree with one node, then repeatedly finds edges of maximum weight to grow the tree one node at a time. Since the time complexity of the algorithm is rather costly, on the order of $O(n^2)$, and calculating the pair-wise cut also takes $O(n^2)$, a new tree was only reconstructed every 20 iterations, so as to preserve runtime. The function returns a dictionary, mapping tuples to the weight of the edge - the first element in the tuple is the parent node, and the second element is the child node (both parent and child are represented by their node numbers, instead of the actual coordinates).

(d) **min-cut**: The min-cut function takes three inputs: source, drain, and tree, and returns the minimum cut from the source to the drain based on the tree input. Since the graph structure is a tree, we first find out the paths that the source and drain need to traverse to the root node (using the tree), then we find the first common node in which both the source and the drain have to traverse. Afterwards, we find the edge with minimum weight from the source to the the common node, and then to the drain, which is the minimum cut of the pair of nodes. This function was used to find the worst minimum cut and the average minimum cut, and was used in the worst-mult-cut function to find the minimum cut from a certain subset to the root node.

(e) **worst-mult-cut**: The worst-mult-cut function aims to find the normalized cut from the subsets in the tree, then take the minimum value. To do so, observe that each subset in the tree basically contains some sort of parent node with some children - basically, only the cut from the parent node to the source matters, since that's the only path for the subset to communicate with the root. With this in mind, we simply obtain the unnormalized cut for each node, then divide the cut by the number of children each node has, including itself. We find the number of children by finding every node's path to the sink, and then count the occurrences of the node in all paths, since every child of a parent node has to go through the parent to reach the sink. Finally, we divide the minimum cut values by the set size, then return the minimum value found.

(f) **Miscellaneous**: Other tertiary functions in the Graph class include the "get" and "set" functions for the variables in the graph class, including the nodes dictionary and the internal tree (that was stored only for sanity check purposes).

2.2 Utility Functions:

(g) **check-obstruction**: The check-obstruction function takes two coordinates as inputs, then returns whether there is an obstruction between the two locations. We assume the coordinates represent the middle of some block, so if two sensors lie on the same row or column, checking whether there is an obstructed coordinate would suffice. If the two sensors are on different rows and columns, we fit a line that passes through both coordinates, and depending on the slope, we either traverse through the row or column coordinates. If the slope is less than 45 degrees, we traverse through the columns, and vice versa.

(h) **comm-rate**: The comm-rate function is returns the communication rate calculated based on whether two coordinates are obstructed, according to the check-obstruction function.

2.3 Node-Adding Functions:

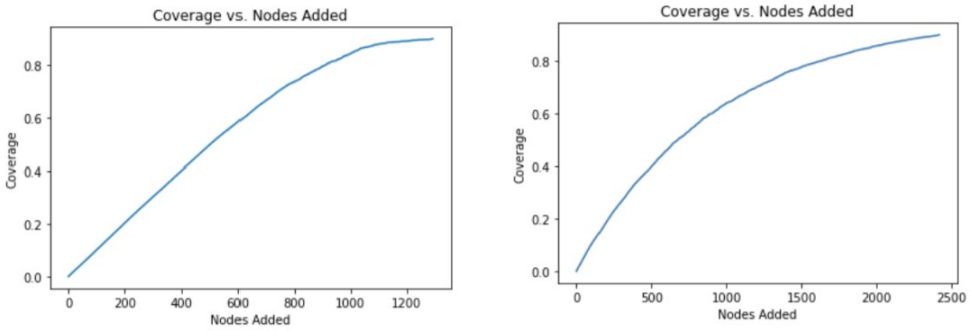
(i) **add-nodes**: The add-nodes function takes four arguments: G, n-nodes, map, and sensor-map. It first repeatedly runs add-one-node for n-nodes times, then calculates the worst case minimum cut, average minimum cut, and the minimum multiple unicast cut, by utilizing the algorithms specified above. Since some of these algorithms already take above $O(n^2)$ complexity to run, these processes became extremely costly. Thus, each value was calculated every 20 iterations to preserve runtime.

(j) **add-one-node**: The add-one-node function takes three arguments: G, map, and sensor-map. It employs a waterfall thresholding technique to first search for the most optimal placement locations, utilizing a set of three different thresholds. As always, a new node is always tied next to an "anchor" node, so that the algorithm will not place a subsequent node on the opposite end of the map. Within this anchor, it will search a predefined square of surrounding coordinates, choosing the location that meets the coverage threshold. It will fall down to the next threshold if no option is located, until it reaches completely random placement at the fourth threshold. To avoid connectivity issues, all nodes are placed within a range of this anchor node, so that a node is guaranteed to be within a certain distance to another node in the network.

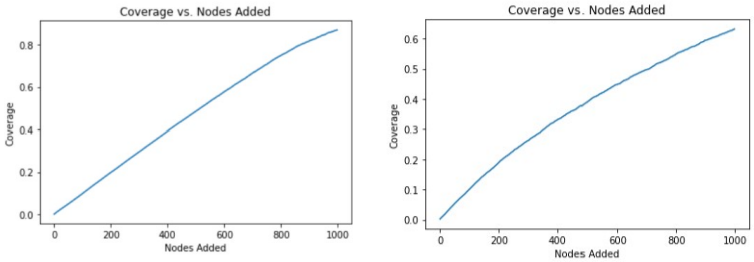
(k) **update-graph**: The update-graph function takes three arguments, G, loc, and map. This function first adds a node to the graph G based on the location specified, then it uses add-edge to add every single edge from loc to all previously added nodes based on the weights calculated via comm-rate (since the graph we are constructing is a complete graph and the spanning tree is only used for calculating connectivity purposes).

3 Figures and Results

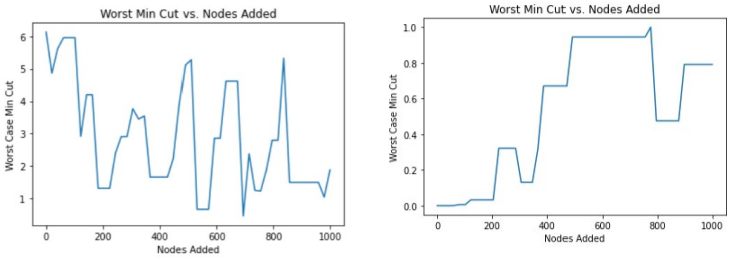
(a) **Minimum Requirement for 90% Coverage**: Our algorithm required 1293 nodes to reach 90% coverage, while the random placement required 2419 nodes to reach 90% coverage. Below are the figures required for 90% for our algorithm (left) and random placement (right):



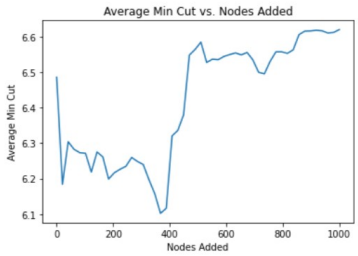
(b) **Coverage for 1000 Sensors**: Our algorithm demonstrated a reasonable increase in coverage over the uniform random placement, with a coverage of 86.917% at 1000 nodes, versus 63.175% at 1000 nodes. Below are the figures for our algorithm (left) and random placement (right):

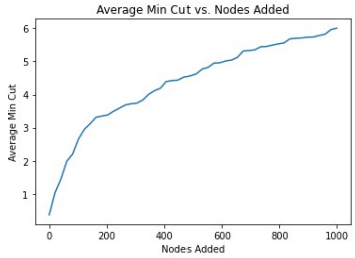


(c) **Worst Case Minimum Cut Comparison**: Below are the subplot figures illustrating the worst case minimum cut performance for our algorithm (left) and random placement (right):



(d) **Average Minimum Cut Comparison**: Below are the subplot figures illustrating the average minimum cut performance for our algorithm (left) and random placement (right):





(e) **Minimum Multiple Unicast Cut Comparison**: Below are the subplot figures illustrating the minimum multiple unicast cut performane for our algorithm (left) and random placement (right): ***Note**: Multiple Unicast Cut was abbreviated as Multicast Cut so that the plot titles wouldn't be too long