**PROJECT MID-TERM REPORT**

**CSE 551 FOUNDATIONS OF ALGORITHMS**

**KALLURI SRINIVASULA REDDY (skallur6@asu.edu)**

**PRAVEEN RAO NITTOOR (pnittoor@asu.edu)**

**AVINASH REDDY SAMALA (asamala2@asu.edu)**

**AMIT SINGH (asing227@asu.edu)**

**CONTENTS:**

**INTRODUCTION**

**PROJECT FORMULATION**

**PROBLEM SOLUTION**

1. **Generating the completed graph from the terminal points**
2. **Budget Constraint Relay Node Placement – Minimum Number of Connected Components**
3. **Budget Constraint Relay Node Placement – Maximizing the Largest Connected Component**

**PROGRESS**

**CONCLUSION**

**INTRODUCTION:**

In wireless sensor networks, Relay node placement problem have been studied widely in the last few years because of its importance. The bottom line of this problem is to place fewest number of possible relay nodes in deployment area to ensure connectivity in a sensor network formed by sensor nodes and relay nodes.

In this project we are implementing the paper "Budget constrained relay node placement problem for maximal connectedness" with goal of designing sensor networks along with relay nodes to maximize the connectedness with a fixed budget constraint. The outcome of this work is to look for positions of relay nodes which can increase the connectivity formed by sensor and relay nodes. Two metrics are provided to measure the connectedness in a disconnected graph. The first metric is the number of connected components, lower the number of connected components is an indicator of a higher degree of connectedness of the graph. The second metric is the number of nodes in the largest connected component – larger size of the largest connected component is an indicator of higher degree of connectivity.

The mathematical abstraction of relay node placement problem corresponds to Geometric Steiner Tree Problem and the terms Steiner points and terminal points corresponds to locations of relay nodes and sensor nodes. Let’s consider a problem scenario mentioned in this paper. First consider a set of twenty-three sensor nodes. The set is divided into three clusters with first one consists of ten sensor nodes and second one consists of eight sensor nodes and similarly third one consists of five sensor nodes. Let’s consider a scenario where the maximum inter cluster distance is less than twice the communication range. In this case we need two relay nodes to make the graph fully connected. But due to budget constraint if we have only one relay node. According to the first metric of connectedness connecting second cluster and third cluster using relay node will give the optimal solution. This is true as there are exactly two connected components which is the best that can be achieved with only one relay node. But the largest connected component is having thirteen nodes and the above solution is not optimal as per the Second metric of connectedness. If we connect the first and second cluster, it forms a component with eighteen sensor nodes. This solution is optimal according to second metric of connectedness. This solution also satisfies the first metric of connectedness.

**PROBLEM FORMULATION:**

The goal of the project, that is, to heighten the “connectedness” of a wireless sensor network using limited number of relay nodes can be formulated in two different ways, that is, two different problems.

The parameters involved in solving these problems are given by:

a) A set of sensor nodes located at S = {s­1, s2, . . . . . ., sn} in the Euclidean plane,

b) the range of effective communication R of sensor and relay nodes, and

c) limitation on number of relay nodes that can be used, denoted by budget Bi.

With the above parameters, a graph’ G = (V, E) is constructed such that for each si ϵ S, a node vi ϵ V is constructed and nodes vi, vj forms an edge ei,j ϵ E if the distance between si and sj is at most R. Due to this constraint, the constructed graph, G is disconnected. To make the additional graph (having both sensor and relay nodes), G’ = (V’, E’) connected, the relay nodes are deployed. Let Bi be the number of relay nodes located at N = {n1, n2, ........, nBi} such that for each ni ϵ N, a node vi ϵ V’ - V is constructed and nodes vi and vj ϵ V’ forms an edge if the distance between ni  and corresponding sj is at most R. This goal can be easily achieved if the budget Bi is unlimited. On other side, this goal cannot be achieved if the budget Bi  is less than the minimum required relay nodes to make the graph, G connected.

The goal of creating the graph G’ = (V’, E’) with as much connectedness as possible, can be achieved by

(i) deploying the relay nodes in a way that minimizes the number of connected components of

G = (V’, E’), referred to as “Budget Constrained Relay node Placement with Minimum Number of Connected Components” (BCRP-MNCC) or

(ii) deploying the relay nodes in a way that maximizes the size of the largest connected components of

G’ = (V’, E’), referred to as “Budget Constrained Relay node Placement for Maximizing the Largest Connected Component” (BCRP-MLCC).

**PROBLEM SOLUTION:**

**1. Generating the completed graph from the terminal points**

Given the data set, we will be parsing the X, Y coordinates of each terminal node given in the Euclidian plane to construct a node in the graph. Once the node is generated, we will add the node into the Hashset data structure.

When a new node is parsed and constructed, it will check all the existing nodes in the Hash set and will add an edge to each of the already exiting nodes in the set. The weight of the edge will be calculated by the Euclidian distance formula that is √ (X2-X1)2 + (Y2-Y1)2 / R where R is the communication range between two Sensor nodes.

To generate the graph, we will be using the **GraphStream** open source library. This library already has inbuilt functions given as follow.

* node addition
* node removal
* edge addition
* edge removal
* graph/node/edge attribute addition
* graph/node/edge attribute change
* graph/node/edge attribute removal

To include the library, we will add the gs-core-X.Y.jar and gs-algo-X-Y.jar in the class path of our Java project.

We can create the graph by the following way:

***Graph graph = new SingleGraph();***

***graph.addNode(<NodeIfo>);***

***graph.addEdge(<SourceNodeInfo>, <DestinationNodeIndo> <weight>);***

**Note:** The **NodeInfo** is class which holds the information of the Node such as Name, X coordinate, Y coordinate. After parsing all the terminal nodes from the dataset and adding all the edges we will generate a complete graph where each node is connected to all other nodes. This Step to generate the complete graph given from the data set is going to be same for both the algorithms to solve either BCRP-MNCC and BCRP-MLCC.

* 1. **Structure of the input Data set**

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~START~~~~~~~~~~~~~~~~~~~~~~~~

1

12

10

5

1,2

3,4

-4,-4

3,-5

-3,2

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~NEXT~~~~~~~~~~~~~~~~~~~~~~~~~

1

10

10

8

3,8

2,2

-4,-4

-1,3

-3,2

7,11

-6,8

5,9

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~~~~~~~~~

The first line in the input tells the start of the file. Then a test case follows

First Line of the test case is the Range of the sensor nodes.(1)

Second line is the Budget 1 constraint for MNCC problem used in algorithm 4(12)

Third line is Budget 2 constraint for MLCC problem used in algorithm 5(10)

Fourth line is the number of sensor nodes (5)

Following 5 lines contain the coordinates of the sensor nodes

After that next line would either contains END or NEXT. If it contains END that means, there are no more test case to run. If it contains NEXT that means, there is test case to run.

We will parse the file using Buffer Reader and create the graph.

**public** **class** Graph **implements** Serializable{

**private** **static** **final** **long** ***serialVersionUID*** = -828725917652137089L;

**public** **int** numOfNodes = 0 ;

**public** Map<Integer, Node> g = **null**;

**public** **double**[][] weightMatrix = **null**;

**int** range = 0;

**public** Graph(**int** numOfNodes, **int** range) {

**this**.numOfNodes = numOfNodes;

**this**.range = range;

g = **new** HashMap<>();

weightMatrix = **new** **double**[numOfNodes][numOfNodes];

}

**public** **void** addNode(**int** id, **int** x, **int** y) {

Node src = **new** Node(id, x, y);

g.put(id, src);

Collection<Node> nodes = g.values();

**for**(Node dest : nodes ) {

**if**(!src.equals(dest)) {

addEdge(src, dest);

addWeight(src, dest);

addEdge(dest, src);

addWeight(dest, src);

}

}

}

**public** **void** addNode(Node node) {

**if**(!g.containsKey(node.getId())) {

g.put(node.id, node);

}

}

**public** **void** addEdge(Node src, Node dest) {

**if**(g.containsKey(src.getId()) && g.containsKey(dest.getId())) {

src.getNeighbours().addLast(dest);

}

}

**public** **void** addWeight(Node src, Node dest) {

**if**(g.containsKey(src.getId()) && g.containsKey(dest.getId())) {

**int** x1 = src.getxCordinate();

**int** x2 = dest.getxCordinate();

**int** y1 = src.getyCordinate();

**int** y2 = dest.getyCordinate();

weightMatrix[src.getId()][dest.getId()] = SensorUtil.*calculateEuclidianDistance*(x1, x2, y1, y2, range);

}

}

**public** **void** addWeight(Node src, Node dest, **double** weight) {

**if**(g.containsKey(src.getId()) && g.containsKey(dest.getId())) {

weightMatrix[src.getId()][dest.getId()] = weight;

}

}

**public** **void** removeEdge(Node src, Node dest) {

**if**(g.containsKey(src.getId()) && g.containsKey(dest.getId())) {

src.getNeighbours().remove(dest);

}

}

**public** Node getNodeById(**int** n) {

**if**(g.containsKey(n)) {

**return** g.get(n);

}

**return** **null**;

}

**public** Node getNodeByCordinates(**int** x, **int** y) {

Collection<Node> nodes = g.values();

**for**(Node next : nodes ) {

**if**(next.getxCordinate() == x && next.getyCordinate() == y) {

**return** next;

}

}

**return** **null**;

}

**public** **double** calculateLength() {

**double** len = 0;

**for**(**int** i=0; i<weightMatrix.length; i++) {

**for**(**int** j=0; j<weightMatrix[i].length; j++) {

len += weightMatrix[i][j];

}

}

**return** len/2;

}

**public** Edge maxWeightEdge() {

**double** max = 0;

Edge e = **new** Edge(-1,-1);

**for**(**int** i=0; i<weightMatrix.length; i++) {

**for**(**int** j=0; j<weightMatrix[i].length; j++) {

**if**(max< weightMatrix[i][j]) {

max = weightMatrix[i][j];

e.src = i;

e.dest = j;

}

}

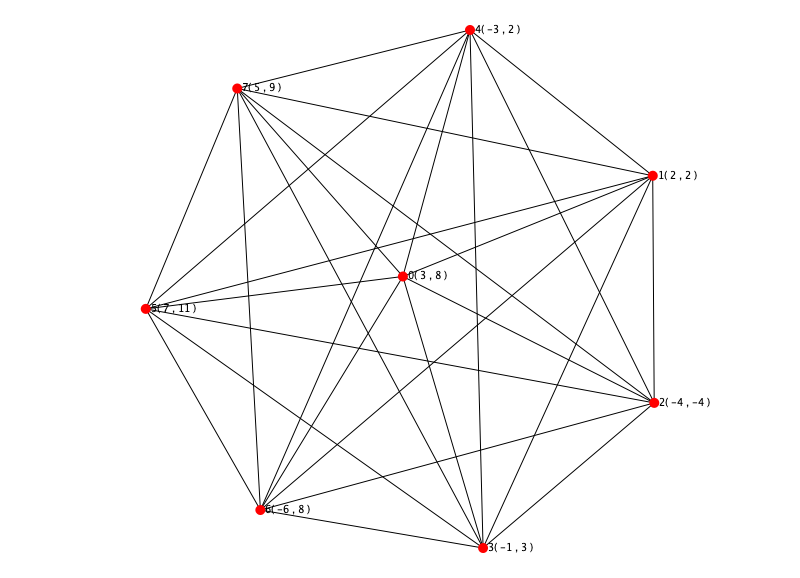
}

**return** e;

}

}

After creating the graph, it will like following



**2. Budget Constraint Relay Node Placement – Minimum Number of Connected Components:**

**Step 1:**

We have generated the complete graph from the terminal points explained in the previous Step. Next Step is to generate the Minimum Spanning tree from the given complete graph.

To generate the Minimum Spanning tree, we will be using the Prim’s Algorithm given as follows

* Associate with each vertex v of the graph a number C[v] (the cheapest cost of a connection to v) and an edge E[v] (the edge providing that cheapest connection). To initialize these values, set all values of C[v] to +∞ (or to any number larger than the maximum edge weight) and set each E[v] to a special “[flag value](https://en.wikipedia.org/wiki/Flag_value)” indicating that there is no edge connecting v to earlier vertices.
* Initialize an empty forest F and a set Q of vertices that have not yet been included in F (initially, all vertices).
* Repeat the following steps until Q is empty:
* Find and remove a vertex v from Q having the minimum possible value of C[v]
  1. Add v to F and, if E[v] is not the special flag value, also add E[v] to F
  2. Loop over the edges vw connecting v to other vertices w. For each such edge, if w still belongs to Q and vw has smaller weight than C[w], perform the following steps:

Set C[w] to the cost of edge vw

Set E[w] to point to edge vw.

* Return F

As described above, the starting vertex for the algorithm will be chosen arbitrarily, because the first iteration of the main loop of the algorithm will have a set of vertices in Q that all have equal weights, and the algorithm will automatically start a new tree in F when it completes a spanning tree of each connected component of the input graph.

**public** **static** Graph createSpanningTree(Graph graph, **int** start){

Map<Integer, DistanceInfo> map = **new** HashMap<>();

**for**(**int** i=0; i<graph.numOfNodes; i++) {

**if**(i==start) {

map.put(i, **new** DistanceInfo(i, 0, i));

}

**else** {

map.put(i, **new** DistanceInfo(i, Integer.***MAX\_VALUE***, -1));

}

}

Set<Integer> spanningTree = **new** HashSet<>();

Set<Integer> visited = **new** HashSet<>();

Queue<DistanceInfo> q = **new** PriorityQueue<>();

q.add(map.get(start));

visited.add(start);

**while**(!q.isEmpty()) {

DistanceInfo df = q.poll();

spanningTree.add(df.node);

**int** parentNodeId = df.node;

Node parent = graph.g.get(parentNodeId);

LinkedList<Node> neighbours = parent.getNeighbours();

**for**(Node neighbour : neighbours) {

**if**(!spanningTree.contains(neighbour.id)) {

**if**(visited.contains(neighbour.id)) {

**if**( map.get(neighbour.id).distance > graph.weightMatrix[parent.id][neighbour.id]) {

map.get(neighbour.id).distance = graph.weightMatrix[parent.id][neighbour.id];

map.get(neighbour.id).lastNode = parentNodeId;

}

}

**else** {

map.get(neighbour.id).distance = graph.weightMatrix[parent.id][neighbour.id];

map.get(neighbour.id).lastNode = parentNodeId;

q.add(map.get(neighbour.id));

visited.add(neighbour.id);

}

}

}

}

Graph spanTree = **new** Graph(graph.numOfNodes, graph.range);

Set<Map.Entry<Integer, DistanceInfo>> set = map.entrySet();

Iterator<Map.Entry<Integer, DistanceInfo>> itr = set.iterator();

**while**(itr.hasNext()) {

Map.Entry<Integer, DistanceInfo> entry = itr.next();

DistanceInfo df = entry.getValue();

**if**(df.distance != Integer.***MAX\_VALUE*** && df.distance != 0){

**int** srcId = df.lastNode;

**int** destID = df.node;

**double** edgeLen = df.distance;

Node srcNodeST = spanTree.g.get(srcId) ;

**if**(srcNodeST== **null**) {

Node srcNode = graph.getNodeById(srcId);

srcNodeST= **new** Node(srcNode.id, srcNode.xCordinate, srcNode.yCordinate);

}

Node destNodeST = spanTree.g.get(destID);

**if**(destNodeST == **null**) {

Node destNode = graph.getNodeById(destID);

destNodeST= **new** Node(destNode.id, destNode.xCordinate, destNode.yCordinate);

}

spanTree.addNode(srcNodeST);

spanTree.addNode(destNodeST);

spanTree.addEdge(srcNodeST, destNodeST);

spanTree.addEdge(destNodeST, srcNodeST);

spanTree.addWeight(srcNodeST, destNodeST, edgeLen);

spanTree.addWeight(destNodeST, srcNodeST, edgeLen);

}

}

**return** spanTree;

}

}

Once the graph gets generated it will look like below

A picture containing skiing, sky

Description automatically generated

**Step 2:**

Calculate the total length of the MST if the length is less than or equal to the Budge then return the MST else

While (Total \_length of MST > B):

Remove the edge with the highest weight from the MST

**Step 3:**

Return the different components in form of forest.

If All the edges are removed, then we will return all the terminal nodes.

Code for the following algorithm looks like

**public** **static** Graph generateMNCCGraphs(Graph mst, **double** budget){

**while**(mst.calculateLength() > budget) {

Edge e = mst.maxWeightEdge();

mst.removeEdge(mst.getNodeById(e.src), mst.getNodeById(e.dest));

mst.removeEdge( mst.getNodeById(e.dest), mst.getNodeById(e.src));

mst.addWeight(mst.getNodeById(e.src), mst.getNodeById(e.dest), 0);

mst.addWeight( mst.getNodeById(e.dest), mst.getNodeById(e.src), 0);

}

**return** mst;

}

Resultant graph from this is shown below

A close up of a map

Description automatically generated

**2. Budget Constraint Relay Node Placement – Maximizing the Largest Connected Component:**

We need to generate the complete graph from the terminal nodes given in the Euclidian Plane, the step to generate the graph is discussed above.

Once the graph is generated, this algorithm requires us create a K-MST from the complete graph. The algorithm to generate the K-MST is discussed below

**Brute-Force Approach:**

We have N nodes in the graph, we need to generate the K-MST out of these N nodes. We can select all combinations K-nodes from these N-nodes and generate the K-MST out of those K-nodes. There will be **nCk**combinations possible for these k nodes from N nodes.

For each combination, we will generate a MST and check if theses nodes generate a connected MST by checking the edges in the original completed graph, if not then discard the combination.

Check the length of all he MST’s and choose the MST with the lowest length.

This is naïve approach to generate the K-MST from the N-node completed graph with very high complexity.

**Optimized Approach:**

To optimize the generation of K-MST from the complete graph, there is better approach available in the paper “**Approximate k-msts and k-steiner trees via the primal-dual method and lagrangean relaxation”** by F. A. Chudak, T. Roughgarden, and D. P. Williamson

We are still going through the paper to understand the generation of K-MST and will be using the approach given in the paper in our project while implementing the Algorithm 5.

To generate K-MST from Given MST, we will find the difference of the nodes (n-k)

N = Total nodes in graph

K = Number of nodes in K\_MST

Nodes to be removed = n-k

To remove the node, we will use the logic such that we will find all the nodes which are terminal that is has no neighbors and will delete the neighbor with maximum edge weight.

Repeat the steps n-k times.

**public** **static** Graph generateKMST(Graph mst, **int** k) {

List<Integer> terminalNodes = **new** ArrayList<>();

Map<Integer, Node> g = mst.g;

Set<Map.Entry<Integer, Node>> set = g.entrySet();

Iterator<Map.Entry<Integer, Node>> itr = set.iterator();

**while**(itr.hasNext()) {

Map.Entry<Integer, Node> entry = itr.next();

**if**(entry.getValue().neighbours==**null** || entry.getValue().neighbours.size()==0) {

terminalNodes.add(entry.getKey());

}

}

**for**(Integer i : terminalNodes) {

g.remove(i);

}

**int** n = g.size();

**int** numOfNodesToBeRemoved = n-k;

**for** (**int** i=0; i< numOfNodesToBeRemoved; i++) {

Edge e = *getTerminalNodeEdgeWithMaximumWeightEdge*(mst);

mst.removeEdge(mst.getNodeById(e.src), mst.getNodeById(e.dest));

mst.removeEdge( mst.getNodeById(e.dest), mst.getNodeById(e.src));

mst.addWeight(mst.getNodeById(e.src), mst.getNodeById(e.dest), 0);

mst.addWeight( mst.getNodeById(e.dest), mst.getNodeById(e.src), 0);

}

terminalNodes.clear();

Map<Integer, Node> g1 = mst.g;

Set<Map.Entry<Integer, Node>> set1 = g1.entrySet();

Iterator<Map.Entry<Integer, Node>> itr1 = set1.iterator();

**while**(itr1.hasNext()) {

Map.Entry<Integer, Node> entry = itr1.next();

**if**(entry.getValue().neighbours==**null** || entry.getValue().neighbours.size()==0) {

terminalNodes.add(entry.getKey());

}

}

**for**(Integer i : terminalNodes) {

g.remove(i);

}

**return** mst;

}

**public** **static** Edge getTerminalNodeEdgeWithMaximumWeightEdge(Graph graph) {

Map<Double, Edge> map = **new** TreeMap<>(Collections.*reverseOrder*());

Set<Integer> terminalNodes = **new** HashSet<>();

Map<Integer, Node> g = graph.g;

Set<Map.Entry<Integer, Node>> set = g.entrySet();

Iterator<Map.Entry<Integer, Node>> itr = set.iterator();

**while**(itr.hasNext()) {

Map.Entry<Integer, Node> entry = itr.next();

**if**(entry.getValue().neighbours==**null** || entry.getValue().neighbours.size()==1) {

terminalNodes.add(entry.getKey());

}

}

**double**[][] weightMatrix = graph.weightMatrix;

**for**(**int** i=0; i<weightMatrix.length; i++) {

**for**(**int** j=0; j<weightMatrix[i].length; j++) {

**if**( weightMatrix[i][j] != 0 && terminalNodes.contains(j)) {

map.put(weightMatrix[i][j], **new** Edge(i,j));

}

}

}

Map<Double, Edge> mapInOrder = **new** LinkedHashMap<>(map);

Map.Entry<Double, Edge> entry = mapInOrder.entrySet().iterator().next();

**return** entry.getValue();

}

**Steps to implement BCRNP-MLCC problem:**

Step 1:

Run a for loop for variable K (number of nodes) starting from N to 2:

For k from N to 2:

N = total number of terminal nodes

Step2:

* Generate K-MST using the algorithm discussed above.
* The weight of each edge between the two nodes is already calculated while generating K-MST and it is calculated by the Euclidian distance formula that is √ (X2-X1)2 + (Y2-Y1)2 / R where R is the communication range between two Sensor nodes.

Step3:

If the total length of MST is less than the budget, return the K-MST as the solution else decrement the value of k from n to n-1 and repeat the Step 2.

If length of k-MST < B:

Return K-MST as a solution

Else

K = K-1

Where B is the allocated budget

Step 4:

If the K = 1

Return any terminal node as solution

**public** **static** Graph generateMLCCGraph(Graph mst, **double** budget){

**int** n = mst.g.size();

**for**(**int** k = n; k>1; k--) {

mst = KMST.*generateKMST*(mst, k);

**if**(mst.calculateLength() <= budget) {

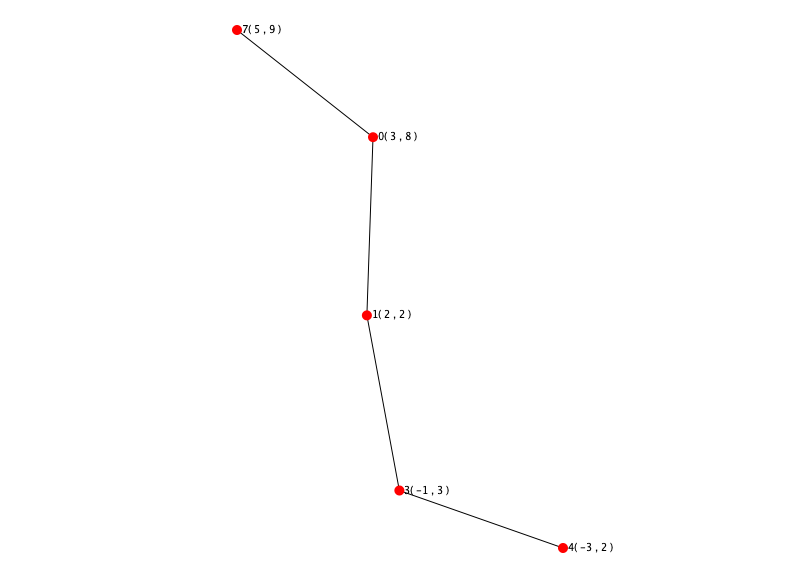
**return** mst;

}

}

**return** mst;

}



The fundamental difference between Algorithm 4 and 5 is:

**Algorithm 4**: we are removing Most weighted Edges and our component count increases by 1 after every edge removal thus resulting in a forest.

**Algorithm 5:** we are removing vertices one by one but always have only one component throughout the algorithm and our result will be only one component.

**PROGRESS:**

**CONCLUSION:**