**Documentation for BigWeather Assignment**

1. **Description of data structures argumentation**

**-Edge class**

To create the adjacency list using the bidirectional edges and to compute the MST through them by using them in the Prim’s Algorithm in Priority Queues. The code is more compact and clean.

**-Adjacency List**

Since dealing with graphs, it is more efficient in storage and makes it easier to iterate through neighbours

**-Priority Queue using min-heap**

To always find the next best edge in an MST, we use need to use Prim’s Algorithm with a Priority Queue using a min-heap because it satisfies the greedy algorithm requirements and it will always give us the smallest cost edge. If a new edge between at least one unvisited dyno is added and is has a smaller weight than the other edges, using the min-heap properties, it will go by proirity to the front and be ready to be used (by being popped) at a very efficient time (O(lon n)).

**-NetworkX graph**

This is a fundamental tool for representing graphs. It consists of vertices and edges that connect pairs. We used it as first of all it is very helpful for visualizing the graph by the input file, also it has pre-built minimum spanning tree functions so it is way more helpful. NetworkX also works well with matplotlib so we can create the visualization easier. In our code G is the graph meanwhile T is MST.

**-Lists**:

**-real\_nodes**: Used to store a filtered list of node IDs, specifically excluding the abstract central bucket node (0). This is primarily for visual clarity, allowing the plot to focus on the dynos.

**-node\_colors**: Stores a sequence of color strings that directly correspond to the real\_nodes. This allows for distinct visual differentiation of dynos based on their connection type in the MST (Red if there is a hosted bucket and blue if there is not)

**-mst\_edges\_to\_draw**: Holds a list of edges from the MST T, that represent actual bonds between dynos (excluding edges connected to node 0). This ensures that only needed connections are displayed on visualization.

**-Dictionary(pos)**:

A collection of key-value pairs, where each node ID is a key and its corresponding 2D coordinate for plotting is the value.

- **Set**:

Used to efficiently store the IDs of dyno nodes that are *directly* connected to the central bucket node 0 within the calculated MST T.

1. **Pseudo-code for the algorithm**

FUNCTION main:

//Edge class

Class Edge:

Init Edge(startnode, endnode, weight):

this. v = startnode; this.z = endode; this.weight = weight

//Adding edges in adjacency list

Create adjacency list of list of edges A

for (num of bonds)

read line(num). create list of characters L

u = L[0] v=[1]

A.append(Edge(u,v, bondcost)

A.append(Edge(v,u, bondcost)

For (num of dynos + 1)

A.append(Edge(0,num, bucketcost)

A.append(Edge(0,num, bucketcost)

//Prim’s Algorithm

for all u ∈V do

cost(u) = inf; prev(u) =nil

end for

pick source node 0; cost(0) = 0

H=make-queue(V)

while |H|>0 do

v = heapop(H)

for each (v ,z) ∈E do

if cost(z) >w (v ,z) then

cost(z) = w (v ,z); mst\_cost += w(v, z)

heapop(H,z)

end if

end for

end while

return the tree represented by prev

end function

Read n, m, bc, bdc from "data.txt"

Initialize Graph G

// Add edges between dynos

FOR m times:

Read u, v

Add edge (u, v) to G with weight bdc

// Add edges from central node (0) to all dynos

FOR i from 1 to n:

Add edge (0, i) to G with weight bc

Calculate Minimum Spanning Tree T of G

Calculate mst\_weight = sum of weights of edges in T

Calculate total\_msts = CALL count\_all\_msts(G, mst\_weight)

Print "Total MSTs with same minimum cost:", total\_msts

// Visualization

real\_nodes = all nodes in G except 0

pos = Calculate spring layout for G's subgraph containing real\_nodes

bucket\_nodes = Set of nodes directly connected to 0 in T

node\_colors = List of colors ('red' if node in bucket\_nodes, else 'skyblue') for real\_nodes

Draw nodes (real\_nodes) with node\_colors using pos

Draw labels for real\_nodes

mst\_edges\_to\_draw = Edges in T that do not involve node 0

Draw edges (mst\_edges\_to\_draw) in green using pos

Create custom legend handles

Add legend to plot

Set plot title

Remove axes

Show plot

END FUNCTION

1. **Discussion or proof arguing about the correctness of your algorithms**

**Correctness of Prim's Algorithm:** Prim's algorithm relies on always choosing the globally minimum-weight edge that expands the current MST. A min-heap ensures that this property is maintained throughout the algorithm's execution, leading to the correct MST and MST cost.

**-Finding MST (2nd time):** We use networkx.minimum\_spanning\_tree(). This function uses algorithms like Kruskal's or Prim's. These algorithms are proven to find the absolute lowest-cost way to connect all parts of a graph.

-**The count\_all\_msts function is a brute-force check**. It correctly finds all *other* possible MSTs with the *exact same minimum cost* by:

* Looking at **every single combination** of edges
* It verifies if edges are connected and if total cost of edges match the MST cost.

1. **Time complexity**

**Initializing bucketcost, bondcost num of dynos and bidirectional edges:** O(N)

**Adjacency List of edges:**

Since first we add arrays based on the number of nodes: O(V)

Then, adding bond number of edges means time complexity is O(N)

So overall it’s O(V+E)

**Prim’s Algorithm for finding MST**

O(E logV) where where V=dynos+1 and E=bonds+dynos

**Graph build:**  
for \_ in range(m): O(m)  
for i in range(1, n + 1): O(n)  
Adding edges is O(1)

Total time complexity is O(m+n) which is =O(E) as m+n is total edges.

**MST:**  
The networkx implementation performs at O(E log N). N is number of nodes.

Summing MST and counting : The total time complexity is the number of combinations multiplied by the cost of processing each combination.

* This results in approximately O (C(E,N−1)\* N) (the highest time complexity in algorithm)

**Visualization: O(N^2)****So the total time complexity of the algorithm (without bonus consideration parts) is: O(E logV)**

**So the total time complexity of the algorithm (with bonus consideration parts) is:** **O (C(E,N−1)\* N)**