UT8 – PD3

**Parte 1: Lenguaje Natural**

Para la solución de este problema se desarrolla un algoritmo que encuentre la forma más eficiente de conectar todas las casas y la subestación de energía eléctrica con cables, minimizando la cantidad de cable necesaria. Para lograr esto, se utiliza un grafo no dirigido y ponderado, donde cada vértice representa una casa o la subestación, y cada arista representa la conexión entre dos vértices con un costo asociado (la distancia entre ellos).

El algoritmo itera sobre el grafo, seleccionando las aristas con menor costo que conecten los vértices, hasta que todos los vértices estén conectados. El resultado es un árbol de expansión mínima (MST) que representa la forma más eficiente de conectar todas las casas y la subestación.

**Parte 2: Estructuras de datos**

Para resolver el problema son necesarias las siguientes estructuras de datos:

* Grafo: una estructura de datos que represente el grafo no dirigido y ponderado, compuesto por:
  + Vértices (casas y subestación de energía eléctrica): un conjunto de nodos identificados por un ID único.
  + Aristas: un conjunto de pares de vértices conectados, con un peso o costo asociado (la distancia entre las casas y la subestación).
* Árbol de expansión mínima (MST): una estructura de datos que represente el subgrafo resultante, compuesto por:
  + Vértices: los mismos vértices del grafo original.
  + Aristas: un subconjunto de aristas del grafo original, seleccionadas para formar el MST.

**Parte 3: Seudocódigo**

// Inicializar el grafo con los vértices y aristas

Grafo grafo = new Grafo();

grafo.agregarVertice("Subestación");

for each casa in casas {

grafo.agregarVertice(casa);

grafo.agregarArista(casa, "Subestación", distancia(casa, "Subestación"));

}

for each par de casas {

grafo.agregarArista(par.casa1, par.casa2, distancia(par.casa1, par.casa2));

}

// Encontrar el Árbol de Expansión Mínima (MST)

MST mst = new MST();

mst = encontrarMST(grafo);

// Mostrar el resultado

for each arista in mst.getAristas() {

println("Conectar " + arista.getVertice1() + " con " + arista.getVertice2() + " con un costo de " + arista.getPeso());

}

// Función para encontrar el MST

MST encontrarMST(Grafo grafo) {

MST mst = new MST();

// Seleccionar el vértice con menor costo (en este caso, la subestación)

Vertice inicial = grafo.getVerticeConMenorCosto();

mst.agregarVertice(inicial);

// Iterar hasta que todos los vértices estén conectados

while (!mst.estanTodosConectados()) {

// Seleccionar la arista con menor costo que conecte un vértice del MST con uno fuera del MST

Arista arista = grafo.getAristaConMenorCosto(mst.getVertices());

mst.agregarArista(arista);

}

return mst;

}

**Parte 4: Análisis del orden del tiempo de ejecución**

El orden del tiempo de ejecución de la aplicación depende del algoritmo utilizado para encontrar el MST. En este caso, se puede utilizar el algoritmo de Kruskal o el algoritmo de Prim.

Algoritmo de Kruskal: el orden del tiempo de ejecución es O(E log E), donde E es el número de aristas del grafo.

Algoritmo de Prim: el orden del tiempo de ejecución es O(E + V log V), donde E es el número de aristas y V es el número de vértices del grafo.