Pontificia Universidad
JAVERIANA
Colombia

Entrega 3: Proyecto

Estructuras de Datos

Integrantes:
Luisa Vargas
Liseth Lozano
Manuela García

Acta de evaluación de la entrega anterior

Para este caso no se aplicaron cambios debido al acta de evaluación de entrega anterior.

TADs

TDA Robot:

Datos Mínimos:

-*coordenadaX*: Valor flotante que corresponde a la posición en el eje x del robot.
-*coordenadaY*: Valor flotante que corresponde a la posición en el eje y del robot.
-*orientación:* Valor flotante que corresponde hacia el grado en el plano que está mirando el robot.

Operaciones:

-*obtenerCoordenadaX*: Retorna el valor actual de la coordenada en X.
-*obtenerCoordenadaY:* Retorna el valor actual de la coordenada en Y.
-*obtenerOrientación:* Retorna el valor actual hacia cuantos grados está apuntado el robot.
-*fijarCoordenadaX (CoordenadaX):* Fija el nuevo valor para la coordenada X.
-*fijarCoordenadaY (CoordenadaY):* Fija el nuevo valor para la coordenada Y.
-*fijarOrientación (Orientación):* Fija el nuevo valor para la Orientación.
-*simularComandos(CoordenadaX,CoordenadaY):* Permite simular el resultado de los comandos de movimiento que se enviarán al robot.
-*guardarComandos(tipo_archivo,nombre_archivo):* Guarda en el archivo nombre_archivo la información solicitada de acuerdo al tipo de archivo: comandos almacena en el archivo la información de comandos de movimiento y de análisis que debe ejecutar el robot.

TDA Comando

Datos Mínimos

-*nombreComando*: Nombre del comando del robot.
-*descripción*: Descripción de lo que hace ese comando en específico.

Operaciones

-*obtenerNombreComando*: Retorna el nombre actual del comando.
-*obtenerDescripción:* Retorna la descripción del comando.
-*fijarNombreComando(nombreComando):* Fija el nuevo nombre del comando.
-*fijarDescripción(descripción):* Fija la nueva descripción del comando.
-*mostrarComandos*: Imprime todos los comandos del robot.
-*cargarComandos*: Se cargan los comandos (análisis y movimiento) desde un archivo en específico.

TDA Análisis:

Datos Mínimos:

-*tipoAnalisis*: Especifica el tipo de análisis que puede ser fotografiar, composición o perforar.
-*objeto*: Nombre del elemento que se quiere analizar.

-*comentario*: Agrega más información del elemento o análisis.
Operaciones:
-*obtenerTipoAnalisis*: Retorna el tipo de análisis actual.
-*obtenerObjeto*: Retorna el objeto actual del análisis.
-*obtenerComentario*: Retorna el comentario actual del análisis.
-*fijarTipoAnalisis(tipoAnalisis)*: Fija el nuevo tipo de análisis.
-*fijarObjeto(objeto)*: Fija el nuevo objeto del análisis.
-*fijarComentario(comentario):* Fija el nuevo comentario del análisis.
-*agregarAnalisis(tipoAnalisis, objeto, comentario)*: Agrega el comando de análisis descrito a la lista de comandos del robot.
-*cargarComandos(nombreArchivo)*: Carga al vector de análisis la lista de comandos tipo análisis encontrados en el archivo.

TDA Elemento:
Datos Mínimos:
-*tipo_comp:* Componente del elemento que puede ser uno entre roca, cráter, montículo o duna.
-*unidad_med*: Valor con la convención que se usó para su medición (centímetros, metros, ...).
-*tamaño*: Valor real que da cuenta de qué tan grande es el elemento
-*coordX:* Número real que da información de la posición en X del elemento en el eje coordenado.
-*coordY:* Número real que da información de la posición en Y del elemento en el eje coordenado.
Operaciones:
-*obtenerTipoComp*: Retorna el tipo de componente actual.
-*obtenerUnidadMed*: Retorna la unidad de medida actual..
-*obtenerTamaño*: Retorna el tamaño del componente.
-*obtenerCoordX*: Retorna la posición actual en el componente X.
-*obtenerCoordY*: Retorna la posición actual en el componente Y.
-*fijarTipoComp (tipoComp)*: Fija el tipo de componente actual.
-*fijarUnidadMed(unidadMed)*: Fija la unidad de medida..
-*fijarTamaño(tamaño)*: Fija el tamaño del componente.
-*fijarCoordX(coordX)*: Fija la coordenada en X.
-*fijarCoordY(coordY)*: Fija la coordenada en Y..
-*agregarElemento (tipoComp, tamaño, unidadMed, coordX, coordY):* Agrega el componente o elemento descrito a la lista de puntos de interés del robot

TDA Movimiento:
Datos Mínimos:
-tipo_movimiento: Corresponde al tipo de movimiento puede ser de dos tipos: avanzar o girar.
-magnitud: Corresponde al valor del movimiento.

-unidad_med: corresponde a la convención con la que se mide la magnitud del movimiento.
Operaciones:
-Obtenertipo_movimiento:Retorna el tipo de movimiento actual.
-Obtenermagnitud:Retorna la magnitud actual.
-Obtenerunidad_med:Retorna la unidad de medida actual.
-Fijartipo_movimiento(tipo_movimiento):Fija el nuevo tipo de movimiento.
-Fijarmagnitud(magnitud):Fija la nueva magnitud del movimiento.
-Fijarunidad_med(unidad_med):Fija la unidad de medida del movimiento.
-agregar_movimiento(tipo_mov, magnitud, unidad_med):Agrega el comando de movimiento descrito a la lista de comandos del robot.
-cargar_comandos(nombre_archivo):Carga al vector de movimientos la lista de comandos tipo movimiento encontrados en el archivo.

TDA Quadtree:
Datos Mínimos:
- root : Node*: Puntero al nodo raíz del quadtree.
- capacity int : Capacidad máxima de elementos que puede contener cada nodo antes de subdividirse.
- x_min float:Límite del espacio que abarca el quadtree en las coordenada x mínima.
- x_max float:Límite del espacio que abarca el quadtree en las coordenada x máxima.
- y_min float:Límite del espacio que abarca el quadtree en las coordenada y mínima
- y_max float: Límite del espacio que abarca el quadtree en las coordenada y máxima.
Operaciones:
+Quadtree(int capacity, float x_min, float x_max, float y_min, float y_max) : constructor : Constructor que inicializa el quadtree con la capacidad máxima de elementos por nodo y los límites del espacio.
+ ~Quadtree() : destructor : Destructor que libera la memoria asociada con el quadtree.
+insertarElemento(const Elemento& elemento) : void: Método público para insertar un elemento en el quadtree.
+enCuadrante(float coordX1, float coordX2, float coordY1, float coordY2) : vector<Elemento>: Método público para buscar elementos en un cuadrante específico, definido por las coordenadas x e y.
- crearNodo(float x, float y, const Elemento& value) : Node* :Crea un nuevo nodo con las coordenadas x e y y un valor asociado.
- dividirNodo(Node* node) : void :Crea un nuevo nodo con las coordenadas x e y y un valor asociado.: Divide un nodo en 4 nodos hijos cuando se alcanza la capacidad máxima de elementos en ese nodo.
-ubicar_elementos(nombre_archivo): Carga en memoria los datos de puntos de interés o elementos contenidos en el archivo.
- insertarElementoEnNodo(Node* node, const Elemento& elemento) : bool :Inserta un elemento en un nodo específico del quadtree. Si el nodo alcanza su capacidad máxima, se divide.
-buscarElementosEnCuadrante(Node* node, float coordX1, float coordX2, float coordY1, float coordY2, vector<Elemento>& elementosEnCuadrante) : void : Método privado que

busca elementos en un cuadrante específico a partir de un nodo dado, almacenando los elementos encontrados en un vector.

TDA Nodo:

Datos Mínimos:

-Node* nw :Puntero al nodo hijo noroeste
-Node* ne:Puntero al nodo hijo noreste
-Node* sw: Puntero al nodo hijo suroeste
-Node* se:Puntero al nodo hijo sureste
-std::vector<Elemento> elementos:Vector de elementos almacenados en el nodo

Operaciones:

+dividirNodo(): Esta operación se encarga de dividir el nodo en cuatro nodos hijos (noroeste, noreste, suroeste y sureste) cuando se alcanza la capacidad máxima de elementos en el nodo.
+insertarElementoEnNodo(const Elemento& elemento): Esta operación se encarga de insertar un elemento en el nodo actual. Si el nodo alcanza su capacidad máxima, se invoca la operación dividirNodo() para dividir el nodo y redistribuir los elementos en los nodos hijos correspondientes.

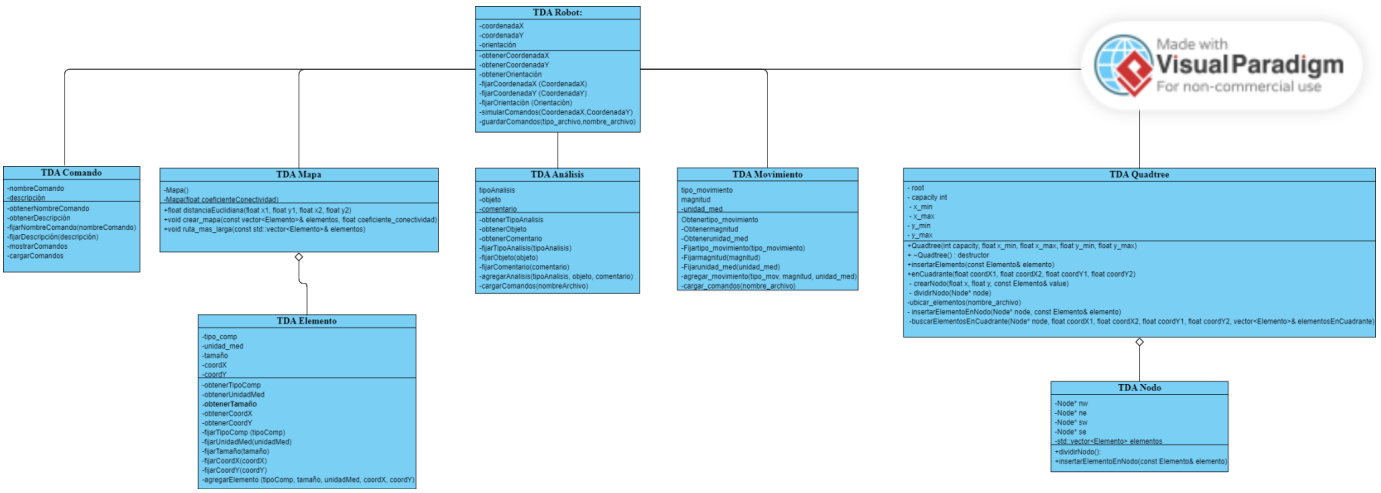
TDA Mapa:

Datos Mínimos:

-Mapa(): Constructor por defecto.
-Mapa(float coeficienteConectividad): Constructor que toma el coeficiente de conectividad como parámetro y lo asigna al atributo correspondiente.

Operaciones:

+float distanciaEuclidiana(float x1, float y1, float x2, float y2): Calcula la distancia euclidiana entre dos puntos (x1, y1) y (x2, y2).
+void crear_mapa(const vector<Elemento>& elementos, float coeficiente_conectividad): Crea un mapa de elementos basado en la cercanía entre ellos y el número permitido de vecinos. Esta función recibe un vector de elementos y el coeficiente de conectividad. Utiliza la fórmula de distancia euclidiana para calcular las distancias entre los elementos y los conecta con sus vecinos más cercanos hasta alcanzar la cantidad deseada.
+void ruta_mas_larga(const std::vector<Elemento>& elementos): Identifica los dos componentes más alejados entre sí en el mapa generado. Esta función recibe un vector de elementos y utiliza el algoritmo de Dijkstra para encontrar la ruta más larga entre los elementos. Luego, imprime la ruta más larga encontrada.



Clases implementadas

Para el proyecto se declararon las siguientes **clases:**

//Clase Análisis -> Dividida en analisis.h y analisis.cpp

```
class Analisis {
private:
    string tipo_analisis;
    string objeto;
    string comentario;

public:
    Analisis();
    Analisis(string tipo, string obj, string com);

    // Getters de la clase Analisis
    string getTipoAnalisis() const;
    string getObjeto() const;
    string getComentario() const;
            void agregar_analisis(string tipo_analisis, string objeto, string comentario,
vector<Analisis>& analisis);
};
```

//Clase Comando -> Dividida en comando.h y comando.cpp

```
class Comando {
private:
  string nombre_comando;
  string descripcion;
```

```
public:
  Comando() {}
  Comando(const string& nombre, const string& desc);

  //Getters de la clase Comando
  string getNombreComando() const;
  string getDescripcion() const;
  void mostrarComandos(vector<Comando>& comandos);
      void cargarComandos(string nombre_archivo,vector<Movimiento>& movimientos,
vector<Analisis>& analisis_v);
};
```

//Clase Elemento ->Dividida en elemento.h y elemento.cpp

```
class Elemento {
private:
    // Atributos de la clase Elemento
    string tipo_comp;
    string unidad_med;
    float tamano;
    float coordX;
    float coordY;

public:
    // Constructores de la clase Elemento
    Elemento() {}
    Elemento(string tipo_comp, string unidad_med, float tamano, float coordX, float coordY);

    // Getters de la clase Elemento
    std::string getTipoComp() const;
    std::string getUnidadMed() const;
    float getTamano() const;
    float getCoordX() const;
    float getCoordY() const;
    void cargarElementos(string nombre_archivo,vector<Elemento>& elementos);
     void agregarElemento(string tipo_comp, float tamano, string unidad_med, float coordX,
float coordY, vector <Elemento>& elementos);
};
```

//Clase Mapa -> Dividida en mapa.h y mapa.cpp

```
class Mapa {
```

```
private:
    Quadtree quadtree;
    vector<Elemento> elementos;
    float coeficienteConectividad;
    map<const Elemento*, map<const Elemento*, float>> mapa;

public:
    Mapa();
    Mapa(float coeficienteConectividad);
    float distanciaEuclidiana(float x1, float y1, float x2, float y2);
                    void conectarElementos(std::vector<Elemento>& elementos, float
distanciaMaximaParaConectar);
                  void crear_mapa(const std::vector<Elemento>& elementos, float
coeficiente_conectividad);
    void ruta_mas_larga(const std::vector<Elemento>& elementos);
};
```

//Clase Movimiento -> Dividida en movimiento.h y movimiento.cpp

```
class Movimiento {
private:
  string tipo_movimiento;
  float magnitud;
  string unidad_med;

public:
  Movimiento();
  Movimiento(string tipo, float mag, string unidad);

  // Getters
  string getTipoMovimiento() const;
  float getMagnitud() const;
  string getUnidadMed() const;
  bool esUnidadValidaParaMovimiento(string tipo_movimiento, string unidad_med);
     void agregar_movimiento(vector<Movimiento> &movimiento, string tipo_mov, float
magnitud, string unidad_med);
};
```

//Clase Node -> Dividida en node.h y node.cpp

```
class Node {
public:
```

```
    float x, y;   // Coordenadas del nodo
    std::string tipo_comp;    // Valor del nodo
    Node *nw, *ne, *sw, *se;  // Punteros a los cuatro hijos del nodo

    Node(float x, float y, std::string tipo_comp);
    ~Node();
};



//Clase Quadtree -> Dividida en quadtree.h y quadtree.cpp
class Quadtree {
  private:
       Node* root;
  public:
    Quadtree(); //constructor por defecto
    ~Quadtree(); //destructor por defecto
    void insert(float x, float y, string tipo_comp);
    void print();
    void insertar(Elemento &elemento);
    void printNode(Node *node);
    void ubicarElementos(vector<Elemento>& elementos, Quadtree &quadtree);
   vector<Elemento> enCuadrante(float coordX1, float coordX2, float coordY1, float
coordY2);
     void buscarElementosEnCuadrante(Node *node, float coordX1, float coordX2, float
coordY1, float coordY2, vector<Elemento> &elementosEnCuadrante);
};
//Clase Robot -> Dividida en robot.h y robot.cpp
class Robot {
private:
  float coordenadaX;
  float coordenadaY;
  float orientacion;

public:
  Robot(); // Declaración del constructor
    void simular_comando (float coordenadaX, float coordenadaY, vector <Movimiento>
movimiento);
};
```

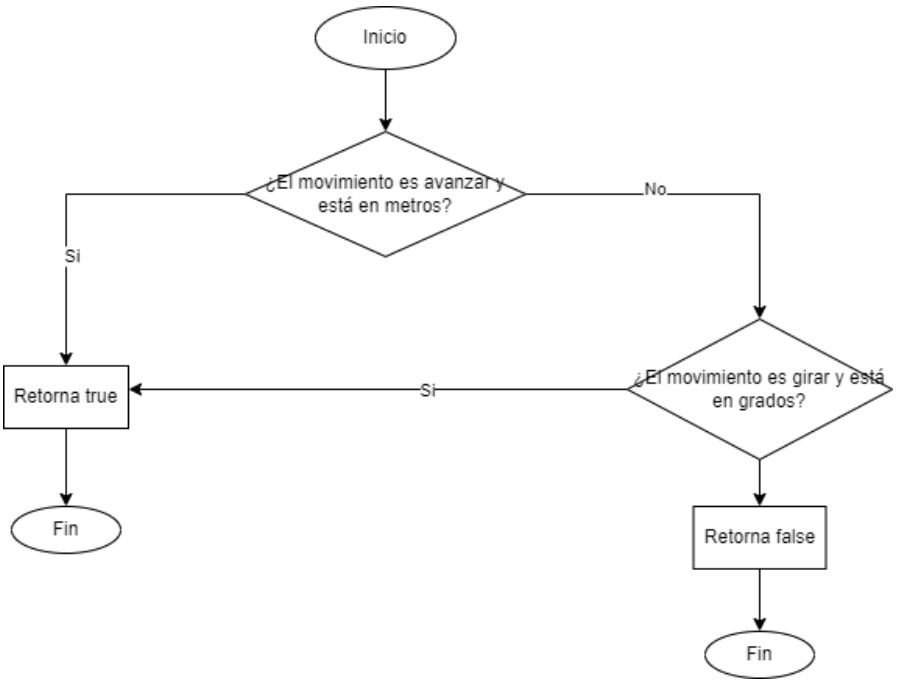
Funciones Implementadas

esUnidadValidaParaMovimiento

En esta funciòn se pasan por paràmetro dos strings (tipo_movimiento, string unidad_med) y se verifican las unidades de medida de la siguiente forma:

→Si es avanzar, se verifica que sean metros

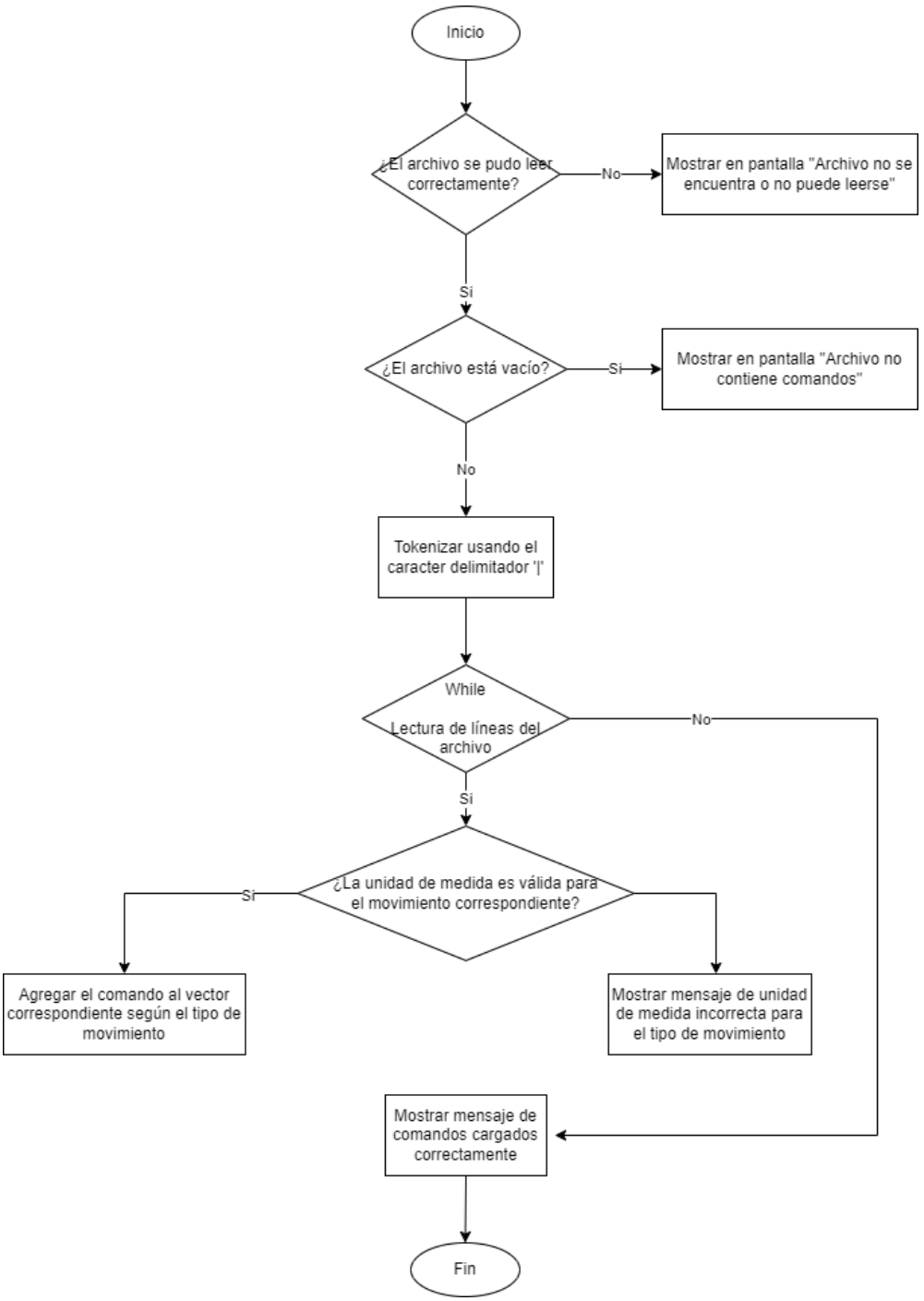
→Si es girar, se verifica que sean grados



cargarComandos

En esta función se pasan por parámetro los vectores de movimientos y análisis, junto con el nombre del archivo que se debe leer.

Se realiza la lectura y apertura del archivo y si todo sale bien, se procede a tokenizar cada línea del archivo usando el caracter delimitador '|' y la función getline, verificando también las unidades de medida (metros o grados dependiendo del desplazamiento) a través de la función **esUnidadValidaParaMovimiento** y posteriormente, se agrega al respectivo vector.



cargarElementos

En esta función se pasan por parámetro el vector de elementos, junto con el nombre del archivo que se debe leer.

Se realiza la lectura y apertura del archivo y si todo sale bien, se procede a tokenizar cada línea del archivo usando el caracter delimitador ‘|’ y la función find, verificando también la unidad de medida del elemento (metros) y posteriormente, se agrega al respectivo vector.

agregar_movimiento

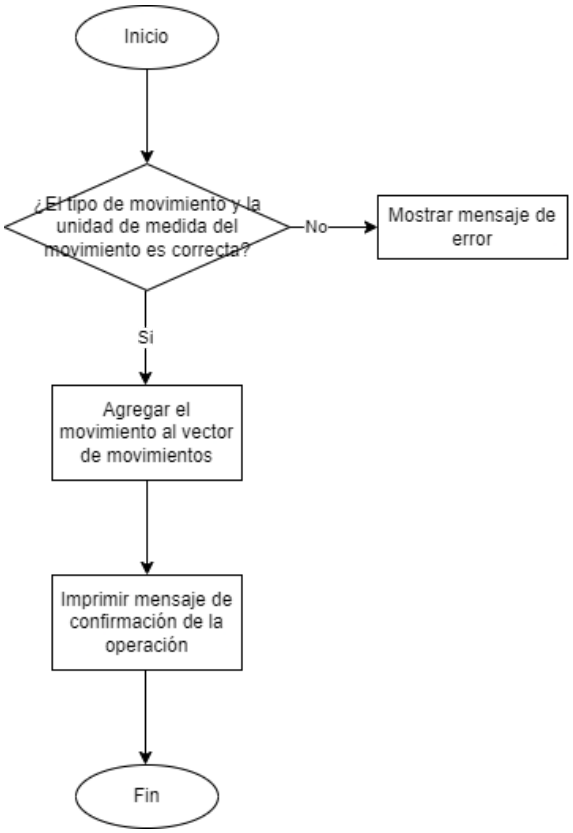
Esta función agrega un nuevo movimiento a un vector de movimientos, donde cada movimiento contiene tres elementos: tipo de movimiento, magnitud y unidad de medida.

Los cuatro parámetros de la función son:

- movimiento: un vector de movimientos existente, al que se agrega el nuevo movimiento.

- tipo_mov: una cadena de caracteres que indica el tipo de movimiento que se va a realizar. Solo se permiten los tipos de movimiento "avanzar" y "girar".
- magnitud: un número decimal que representa la magnitud del movimiento.
- unidad_med: una cadena de caracteres que indica la unidad de medida de la magnitud del movimiento solo se permite metros para el caso de avanzar y grados para el caso de girar.

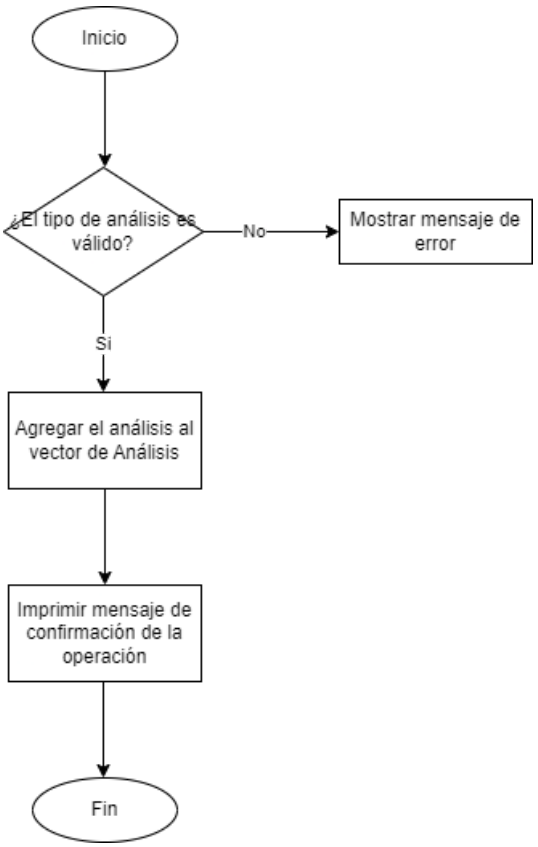
Antes de agregar el nuevo movimiento al vector de movimientos, la función verifica si la información del movimiento es adecuada. Si el tipo_mov es "avanzar" o "girar" y si es metros o grados, entonces la función creará un nuevo movimiento y lo agrega al final del vector de movimientos existente.
Si la información del movimiento no es adecuada, la función imprimirá un mensaje de error y no se agregará ningún movimiento al vector de movimientos.
En cualquier caso, la función imprimirá un mensaje indicando si el comando de movimiento ha sido agregado exitosamente o no.



agregar_analisis
Esta función agrega un nuevo análisis a un vector de análisis, donde cada análisis contiene tres elementos: tipo de análisis, objeto y comentario.
Los tres parámetros de la función son:

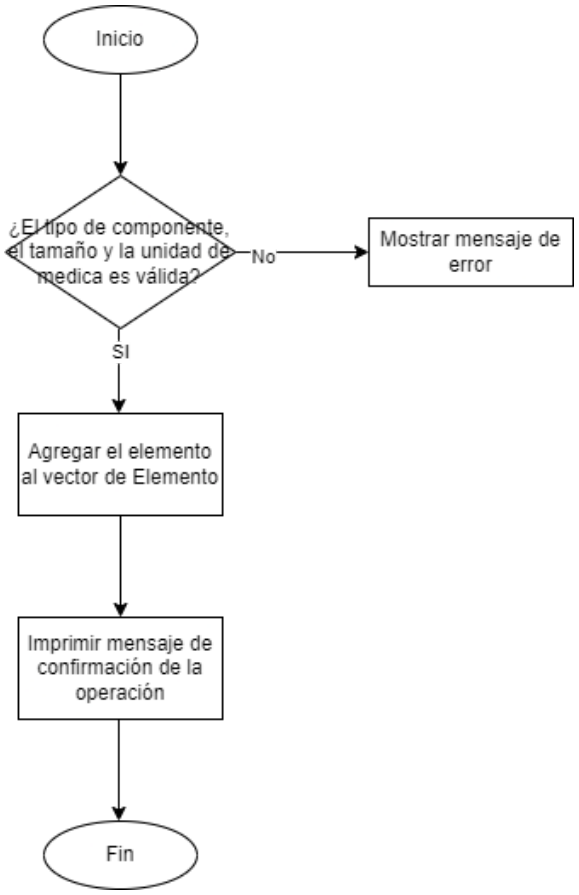
- tipo_analisis: una cadena de caracteres que indica el tipo de análisis que se va a realizar. Solo se permiten los tipos de análisis "fotografiar", "composicion" y "perforar".
- objeto: una cadena de caracteres que representa el objeto que se va a analizar.
- comentario: una cadena de caracteres que contiene algún comentario o descripción adicional sobre el análisis.

El último parámetro es un vector de análisis existente, al que se agrega el nuevo análisis.
Antes de agregar el nuevo análisis al vector de análisis, la función verifica si el tipo_analisis es válido, es decir, si es uno de los tipos de análisis permitidos. Si el tipo_analisis no es válido o el objeto está vacío, la función imprimirá un mensaje de error y saldrá de la función sin agregar el nuevo análisis.
Si el tipo_analisis es válido y el objeto no está vacío, la función creará un nuevo análisis y lo agregará al final del vector de análisis existente. Luego, la función imprimirá un mensaje de éxito.



agregarElemento
La función "agregarElemento" recibe como parámetros información sobre un elemento, como su tipo de componente, tamaño, unidad de medida y coordenadas de ubicación (X y Y), así como también un vector de elementos.

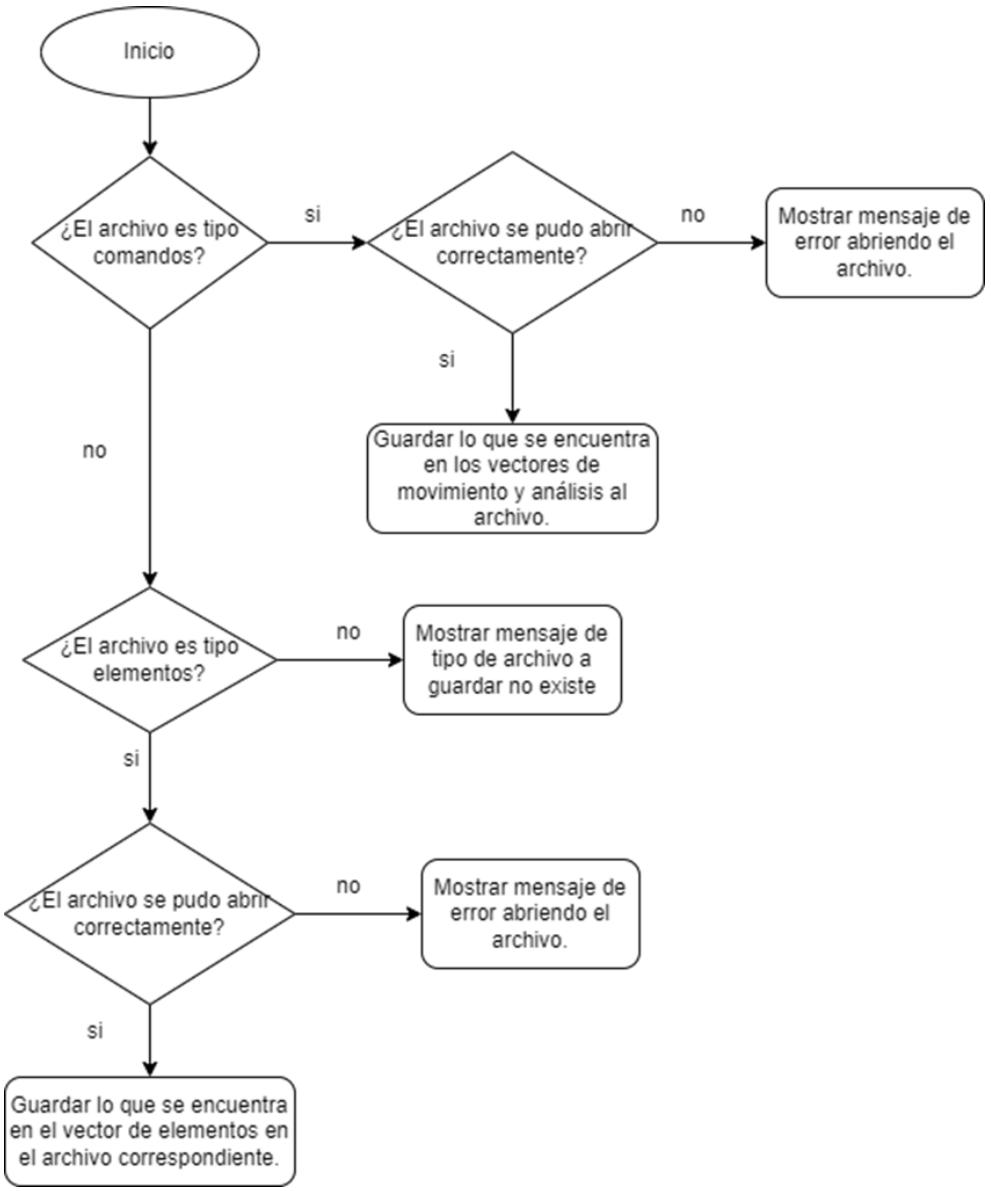
La función verifica que la información ingresada sea adecuada y, si es así, crea una nueva instancia de estructura de Elementos con la información ingresada y la agrega al final del vector de elementos. Si la información no es adecuada, la función simplemente emite un mensaje de error indicando que la información ingresada no corresponde a los datos pedidos. En general, la función se encarga de agregar un nuevo elemento al vector de elementos si se cumplen ciertas condiciones y emite un mensaje de éxito o fracaso dependiendo del caso.

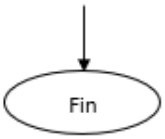


guardar

La función guardar pasa por parámetro el nombre del archivo, tipo de archivo (comandos o elementos) y los vectores de elementos, movimientos y análisis. Se realiza una verificación de que tipo de archivo se debe abrir, para hacer una apertura del mismo y si todo abre de

forma correcta se procede a guardar todos los elementos del correspondiente vector(es) en el archivo, para finalmente cerrarlo.



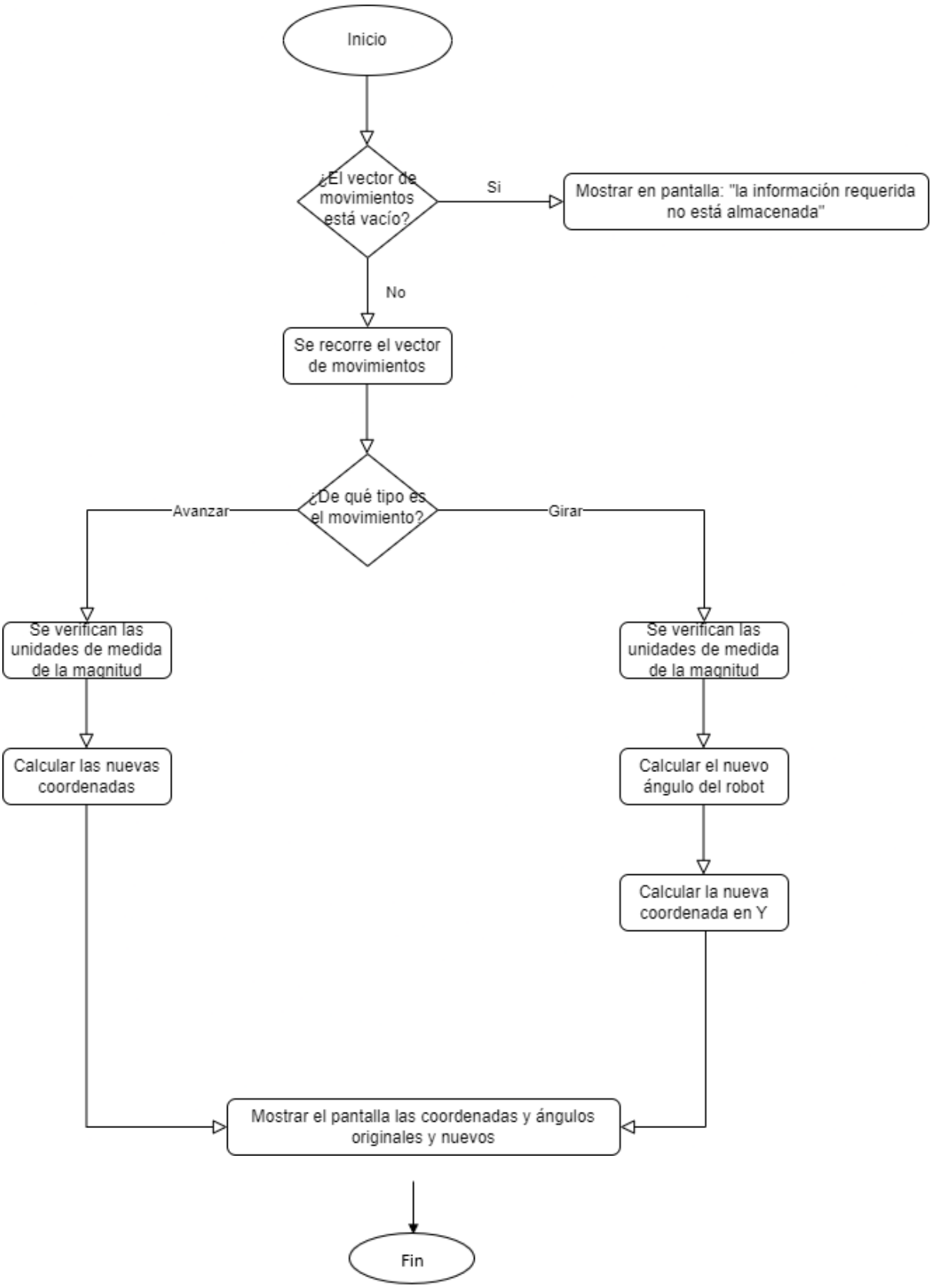


simularComandos

La función simular_comando recibe como parámetros tres variables: coordenadaX y coordenadaY, que representan la posición inicial del robot, y movimiento, que es un vector de la estructura Movimiento que contiene información sobre los movimientos que el robot debe realizar.

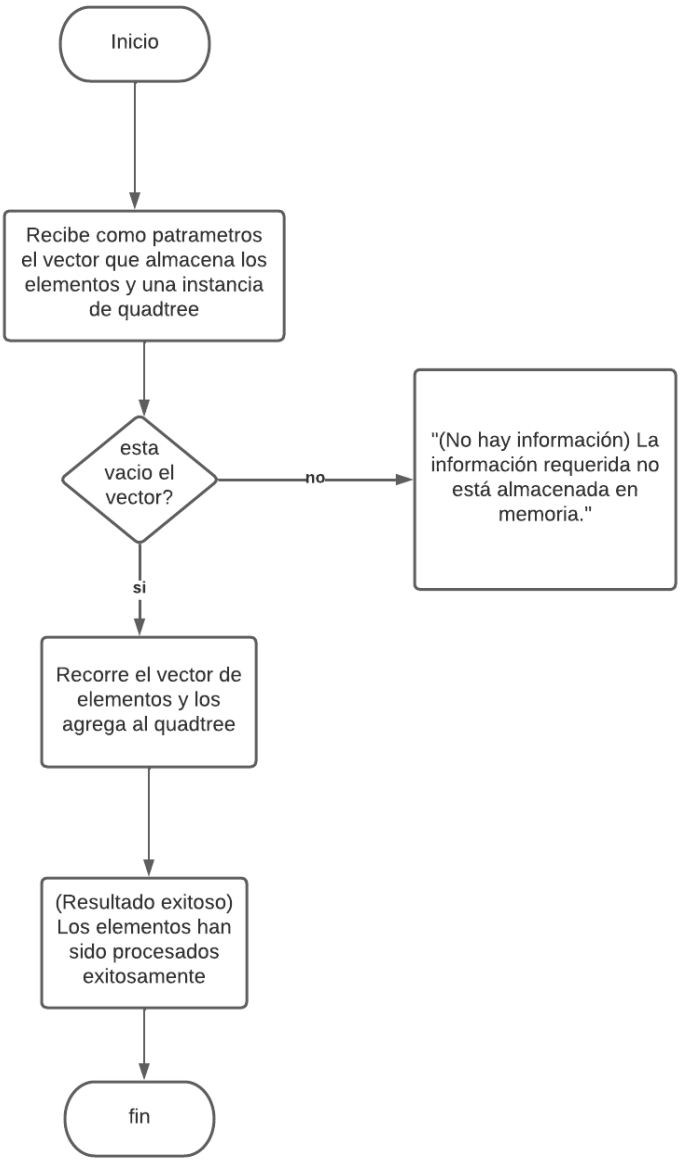
El ciclo for recorre cada elemento del vector movimiento, accediendo a cada elemento mediante el iterador it, dependiendo del tipo de movimiento se verifican las unidades de medida del movimiento.

Si el tipo_movimiento_actual es "avanzar" y la unidad_med_actual es "metros", se calcula la distancia que se debe recorrer y se actualiza la posición del robot en función de la distancia recorrida y el ángulo actual. Mientras que, si el tipo_movimiento_actual es "girar" y la unidad_med_actual es "grados", se calcula el ángulo del giro y se actualiza la posición del robot en función del ángulo y la coordenada en Y.


Inicio
¿El vector de movimientos está vacío?
Si
Mostrar en pantalla: "la información requerida no está almacenada"
No
Se recorre el vector de movimientos
¿De qué tipo es el movimiento?
Avanzar
Girar
Se verifican las unidades de medida de la magnitud
Calcular las nuevas coordenadas
Se verifican las unidades de medida de la magnitud
Calcular el nuevo ángulo del robot
Calcular la nueva coordenada en Y
Mostrar el pantalla las coordenadas y ángulos originales y nuevos
Fin

ubicarElementos

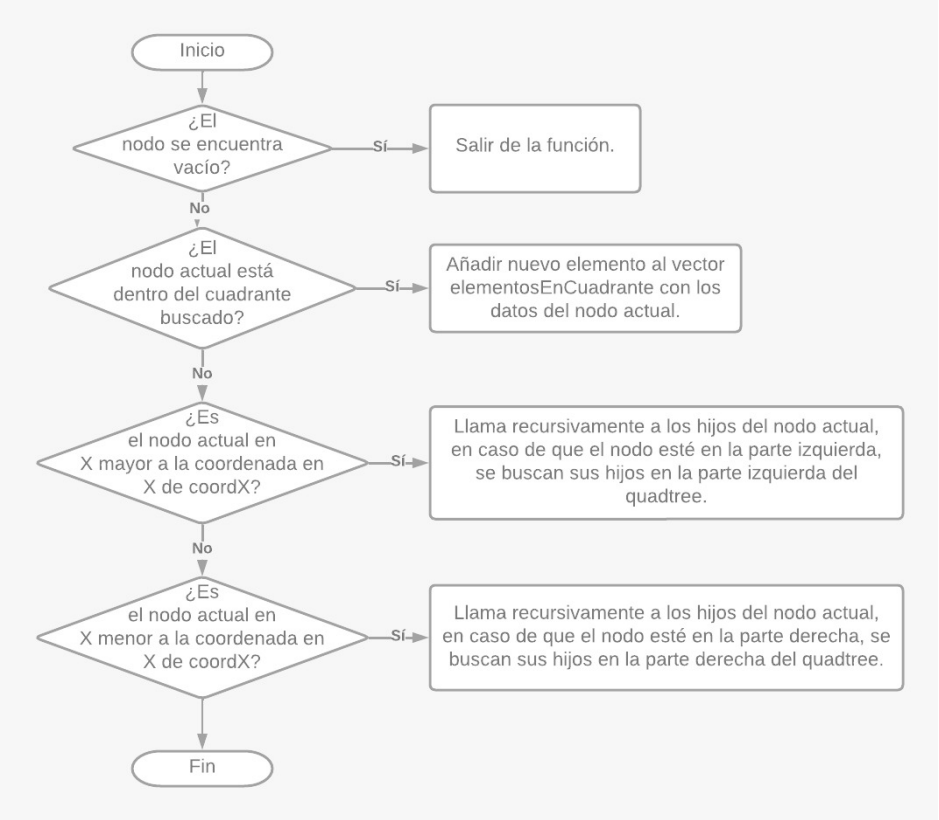
Esta función "ubicarElementos" recibe un vector de objetos Elemento y una instancia del árbol Quadtree. Primero, la función verifica si el vector de elementos está vacío y en caso contrario, agrega cada elemento al Quadtree mediante la función "insert". Después de agregar todos los elementos, la función imprime el árbol Quadtree y muestra un mensaje de éxito en la operación. En resumen, esta función agrega elementos al árbol Quadtree y lo imprime



buscarElementosEnCuadrante

Esta función "buscarElementosEnCuadrante" recibe un nodo del árbol Quadtree, las coordenadas de un cuadrante delimitado por dos puntos (coordX1, coordY1) y (coordX2, coordY2), y un vector de elementos.

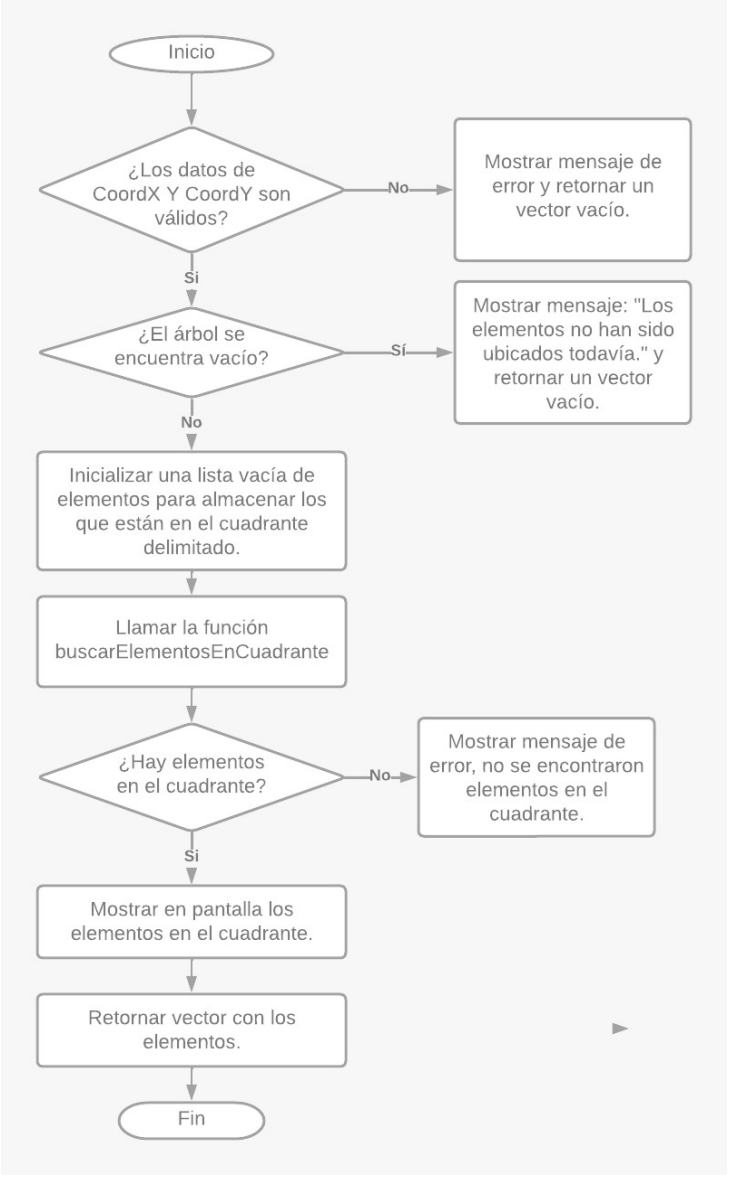
La función verifica si el nodo está dentro del cuadrante buscado, y si es así, añade un nuevo objeto Elemento al vector. La función también realiza llamadas recursivas a los hijos del nodo actual que están dentro del cuadrante buscado. En resumen, esta función busca y añade los elementos dentro de un cuadrante específico del Quadtree.



enCuadrante

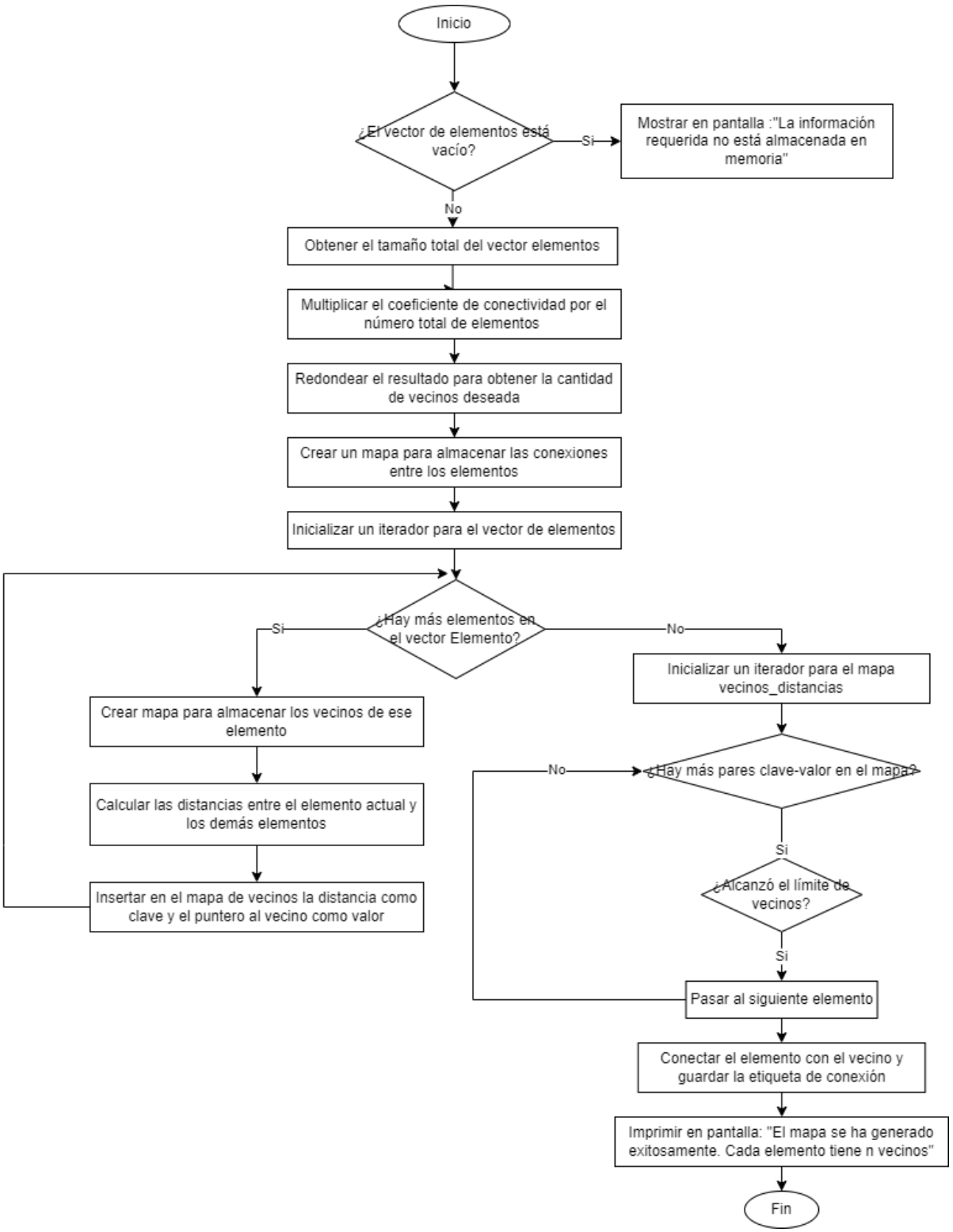
La función "enCuadrante" recibe como parámetros las coordenadas de un cuadrante delimitado por dos puntos (coordX1, coordY1) y (coordX2, coordY2).

Si los datos son válidos y el árbol de búsqueda no está vacío, la función busca los elementos almacenados en dicho cuadrante y los almacena en un vector. Si se encontraron elementos, los muestra en pantalla junto con su información relevante. En caso contrario, muestra un mensaje indicando que no se encontraron elementos. La función retorna el vector con los elementos encontrados o un vector vacío en caso de errores.



crear_mapa

La función crear_mapa recibe un vector de Elemento y un coeficiente de conectividad. Esta función crea un mapa de conexiones entre los elementos, asegurando que cada elemento tenga un número específico de vecinos de acuerdo con la fórmula otorgada (número de elementos * coeficiente de conectividad) Para lograr esto, se calculan las distancias entre cada par de elementos, se seleccionan los vecinos más cercanos y se establecen las conexiones correspondientes en el mapa. Al finalizar, se muestra un mensaje indicando el éxito de la generación del mapa y la cantidad de vecinos que tiene cada elemento.


Inicio
¿El vector de elementos está vacío?
Si
Mostrar en pantalla :"La información requerida no está almacenada en memoria"
No
Obtener el tamaño total del vector elementos
Multiplicar el coeficiente de conectividad por el número total de elementos
Redondear el resultado para obtener la cantidad de vecinos deseada
Crear un mapa para almacenar las conexiones entre los elementos
Inicializar un iterador para el vector de elementos
¿Hay más elementos en el vector Elemento?
Si
No
Crear mapa para almacenar los vecinos de ese elemento
Calcular las distancias entre el elemento actual y los demás elementos
Insertar en el mapa de vecinos la distancia como clave y el puntero al vecino como valor
Inicializar un iterador para el mapa vecinos_distancias
¿Hay más pares clave-valor en el mapa?
No
Si
¿Alcanzó el límite de vecinos?
Si
Pasar al siguiente elemento
Conectar el elemento con el vecino y guardar la etiqueta de conexión
Imprimir en pantalla: "El mapa se ha generado exitosamente. Cada elemento tiene n vecinos"
Fin

Plan de pruebas para ruta_mas_larga:

Caso de prueba 1: Mapa vacío
Datos de prueba:
Vector de elementos: Vacío
Resultado esperado:
Se muestra el mensaje: "(No hay información) El mapa no ha sido generado todavía."

Caso de prueba 2: Mapa con un solo elemento
Datos de prueba:
Vector de elementos: [Elemento1]
Resultado esperado:
Se muestra el mensaje: "(No hay información) El mapa no ha sido generado todavía."

Caso de prueba 3: Mapa con múltiples elementos
Datos de prueba:
Vector de elementos: [Elemento1, Elemento2, Elemento3, Elemento4, Elemento5]
Configuración de las conexiones:
Elemento1 está conectado a Elemento2 y Elemento3.
Elemento2 está conectado a Elemento1 y Elemento4.
Elemento3 está conectado a Elemento1 y Elemento5.
Elemento4 está conectado a Elemento2.
Elemento5 está conectado a Elemento3.
Resultado esperado:
Se muestra el mensaje: "(Resultado exitoso) Los puntos de interés más alejados entre sí son Elemento4 y Elemento5. La ruta que los conecta tiene una longitud total de X y pasa por los siguientes elementos: Elemento2 Elemento1 Elemento3 Elemento5"

Caso de prueba 4: Mapa con elementos desconectados
Datos de prueba:
Vector de elementos: [Elemento1, Elemento2, Elemento3, Elemento4, Elemento5]
Configuración de las conexiones:
Elemento1 está conectado a Elemento2.
Elemento2 está conectado a Elemento1.
Elemento3 está desconectado.
Elemento4 está desconectado.
Elemento5 está desconectado.
Resultado esperado:
Se muestra el mensaje: "(No hay información) El mapa no ha sido generado todavía."