

PROYECTO

Entrega_3

GRUPO 1

Sophia Aristizábal

(Ingeniería de Sistemas y economía)

Juan Felipe Bríñez

(Ingeniería sistemas)

Iván Darío Orozco Ibáñez

(Ingeniería de Sistemas e Ingeniería Electrónica)

Facultad de Ingeniería, Pontificia Universidad Javeriana

SISTEMAS 4800-1 (2891): Estructuras de Datos (Teo-Prac)

Andrea del Pilar Rueda Olarte

21 de Febrero del 2023



Pontificia Universidad
JAVERIANA
Bogotá

Pontificia Universidad Javeriana
Estructura de Datos

Índice

Desarrollo	5
1. Acta de Evaluación segunda entrega	5
Comentario #1	5
Comentario #2	6
Comentario #3	8
Comentario #4	9
2. Acta de Evaluación tercera entrega	11
Comentario #1	11
3. TADs	12
TAD Grafo	12
TAD ArbolUbicarElementos	14
TAD NodoUbicarElementos	15
TAD Sistema	17
TAD Comando	17
TAD Desplazamiento	19
TAD Punto_de_interes	19
TAD Analisis	20
4. Diagrama de TADS.	21
5. Funciones/comandos:	23
cargar_comandos nombre_archivo	23
Diagrama del funcionamiento del comando	24
cargar_elementos nombre_archivo	27
Diagrama del funcionamiento del comando	28
agregar_movimiento tipo_mov magnitud unidad_med	30
Diagrama del funcionamiento del comando	31
agregar_analisis tipo_analisis objeto comentario	31
Diagrama del funcionamiento del comando	32
agregar_elemento tipo_comp tamaño unidad_med coordX coordY	33
Diagrama del funcionamiento del comando	34
guardar tipo_archivo nombre_archivo	35
Diagrama del funcionamiento del comando	36

PROYECTO

simular_comandos coordX coordY	37
Diagrama del funcionamiento del comando	38
Salir	40
Diagrama del funcionamiento del comando	40
Ayuda	41
Diagrama del funcionamiento del comando	43
Ubicar_elementos	44
Diagrama del funcionamiento del comando	46
en_cuadrante coordX1 coordX2 coordY1 coordY2	48
Diagrama del funcionamiento del comando	49
crear_mapa coeficiente_conectividad	50
Diagrama del funcionamiento del comando	52
ruta_mas_larga	54
Diagrama del funcionamiento del comando	55
6. Plan de pruebas comando simular comandos	56
Caso 1: Sin desplazamientos (no existen ni un solo movimiento dentro de la multi-lista de comandos)	57
Caso 2: Se ingresan tipos de datos incorrectos.	57
Caso 3: Cuadrado (donde todos los valores son enteramente positivos)	58
Caso 4: Rombo (Donde todos los valores son enteramente negativos)	59
caso 5: Se ingresan datos combinados (positivos y negativos) desde una posición cero	60
7. Plan de pruebas comando en_cuadrante	62
Caso 1: Se ingresan datos incompletos.	62
Caso 2: Se ingresan datos con un formato incorrecto.	63
Caso 3: Se ingresan datos sin haber ubicado elementos.	63
Caso 4: Verificación de coordenadas.	64
Caso 5: Permite números reales.	65
Caso 6: Ubica todos los elementos dentro del rango especificado.	65
Caso 7: No encuentra elementos dentro del rango especificado.	66
8. Plan de pruebas comando ruta_mas_larga	69
Caso 1: No se encuentra un mapa creado.	69
Caso 2: El formato del comando es incorrecto.	70
Caso 3: Los elementos han sido ubicados y correctamente conectados.	70
(Prueba 1)	71
(Prueba 2)	73



PROYECTO

(Prueba 3)

75



Pontificia Universidad
JAVERIANA
Bogotá

Pontificia Universidad Javeriana

Estructura de Datos

Desarrollo

1. Acta de Evaluación segunda entrega

Comentario #1

Observación realizada:

“El TAD Dato_de_interes creo que tal vez podría llamarse mejor Punto_de_interes”.

Corrección realizada: Se corrigió cambiando el nombre de los archivos vinculados al TAD (el .h y .cxx) de **Dato_de_interes** a **Punto_de_interes**. Así mismo, en el main se cambiaron todas las ocurrencias de **Dato_de_interes** por nombre **Punto_de_interes**. Las siguientes imágenes muestran los cambios comentados:

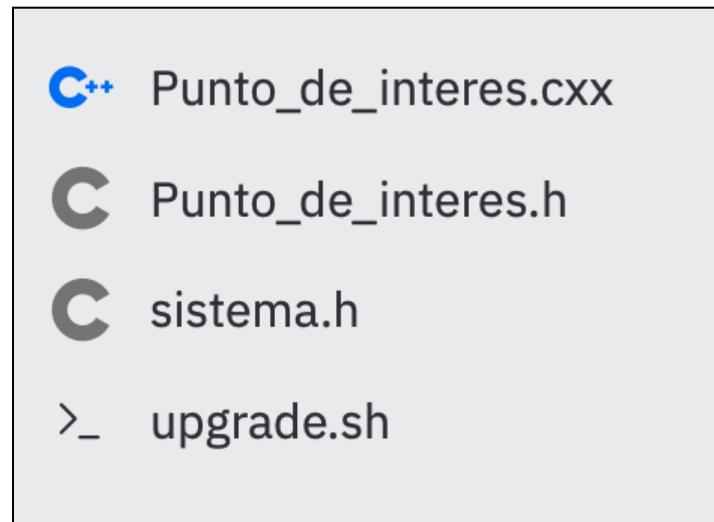


Fig 1. Cambio de nombre de los archivos.

```
int lecturaArchivoElementos(ifstream &flujo_arch_curiosity,
                           list<Punto_de_interes> &puntos_de_interes) {
    int cant_comandos = 0;
    string comando, guardar_input;

    // se elimina el contenido para sobreescribir todo
    puntos_de_interes.clear();

    Punto_de_interes dato;
```

PROYECTO

Fig 2. Ejemplo del cambio de nombre en el main

Comentario #2

Observación realizada:

“Comando cargar_comandos:

- Tal vez la validación sobre si los comandos tienen formato o elementos válidos debería hacerse a este nivel, no al momento de hacer la simulación.”

Corrección realizada: Se resolvió quitando la validación de las unidades de medida dentro del comando **simular_comandos** y en cambio se añadió dentro del comando **cargar_comandos**. Se puede evidenciar como en la *Fig 3.* (una muestra que hace parte del código de **simular_comandos**) ya no se realiza la validación del formato del comando:

```

auxConversion = itC->getCDesplazamiento().getMagnitud() / 100;

}
// si es m
else if (itC->getCDesplazamiento().getUnidadMedida() ==
    unidades[1]) {

    auxConversion = itC->getCDesplazamiento().getMagnitud();

}
// si es km
else if (itC->getCDesplazamiento().getUnidadMedida() ==
    unidades[4]) {
    auxConversion = itC->getCDesplazamiento().getMagnitud() * 1000;
}
// si es dm
else if (itC->getCDesplazamiento().getUnidadMedida() ==
    unidades[5]) {
    auxConversion = itC->getCDesplazamiento().getMagnitud() / 10;

}

arreglo_coordenadas[0] += auxConversion * cos(almacenadorGrados);
arreglo_coordenadas[1] += auxConversion * sin(almacenadorGrados);
}
}
cout << "\nLa simulación de los comandos, a partir de la posición (" 
<< setprecision(2) << fixed << stod(*(arreglo_input + 1)) << ", "
<< setprecision(2) << fixed << stod(*(arreglo_input + 2))
<< "), deja al robot en la nueva posición (" 
<< arreglo_coordenadas[0] << ", " << arreglo_coordenadas[1]
<< ")" . " << endl;
```

Fig 3. Reubicación de validación.

Como se puede observar, solo se busca saber cuál es la unidad de medida del comando para hacer la conversión a metros, para luego de realizar el proceso se imprime únicamente la respuesta de la simulación. Así mismo, dentro del comando **cargar_comandos** se añadió una validación donde comparaba la tercera palabra del comando, si este pertenecía a las palabras establecidas (de



PROYECTO

unidad de medida) se terminaba de crear el comando de desplazamiento y se asigna verdadero a un nuevo Booleano llamado **formatoCorrecto**, que establece si el formato del comando ingresado es el correcto para guardarlo, de lo contrario no es creado. Esta explicación puede observarse en la siguiente figura.

```

getline(X, guardar_input, ' ');
if(guardar_input=="cm" || guardar_input=="m" || guardar_input=="grd" || guardar_input=="rad" ||
guardar_input=="km" || guardar_input=="dm")
{
    // la ltima palabra es la unidad de medida
    desplazamiento.setUnidadMedida(guardar_input);

    // al final esto se guarda en el tad comandos
    comand.setCDesplazamiento(desplazamiento);
    // verdadero es para desplazamiento
    comand.setTipo(true);

    formatoCorrecto=true;
}

```

Fig 4. Validación unidades de medida

Por su parte, este booleano servirá precisamente para determinar si el comando debe ser ingresado dentro de la lista de comandos o descartarlo. La siguiente imagen evidencia esta evaluación, donde sí formato **formatoCorrecto**, se añade el comando y se cuenta en la cantidad de comandos guardados:

```

if (formatoCorrecto) {
    cant_comandos++;
    // se guarda el comando creado en la lista de comandos
    comandos.push_back(comand);
}

```

Fig 5. Validación booleano para inserción de comando

Por último, la validación qué se añadió al comando de agregar movimiento consistió únicamente en añadir un **for** anidado. En este se mirará si la tercera palabra del comando pertenece a alguna de las palabras preestablecidas para la unidad de medida. La siguiente figura muestra en azul el for anidado añadido y arriba se puede apreciar el arreglo de las palabras permitidas para la unidad de medida:

PROYECTO

```

string tipo_magn_arreglo[6] = {"cm", "m", "grd", "rad", "km", "dm"};

bool correcto = false;

// este exactamente debe tener 4 palabras
if (numPalabras == 4) {

    // se evaluara si tipo_mov es correcto
    for (int i = 0; i < 2 && !correcto; i++) {
        if (*(arreglo_input + 1) == tipo_magn_arreglo[i]) {
            // por ahora el comando es correcto
            // evaluar si la magnitud es correcta
            for (int j = 0; j < 6 && !correcto; j++) {
                if (*(arreglo_input + 2) == tipo_magn_arreglo[j]) {
                    correcto = true;

                    // se evalua si la unidad de medida puede ser un float
                    // si no lo se puede castear el string a float es incorrecto
                    try {
                        stod(*(arreglo_input + 2));
                    } catch (const std::invalid_argument &ex) {
                        correcto = false;
                    }
                }
            }
        }
    }
}

```

Fig 6. Ciclo para determinar si el parámetro hace parte de las palabras válidas.

Comentario #3

Observación realizada:

“Comando guardar:

- No hace falta almacenar en el archivo de comandos el indicador 0 o 1 para saber si el movimiento o análisis, puesto que eso no se está usando al momento de la lectura desde el archivo (de hecho, si el archivo los tiene no se puede leer).”

Corrección realizada: En este caso debido a que ocurrencia acontece al inicio de cada línea con los caracteres de de 0 ó 1 en los archivos con los comandos y elementos guardados. Primero nos dirigimos al comando **guardar**, siendo más precisos dentro del flujo de salida **archSalida**, luego en el momento que realiza la escritura dentro del archivo se descarta el primer carácter más el espacio (que hace alusión a la eliminación del problema anteriormente mencionado). Para tener una noción más clara observamos el cambio en *Fig 7*:



PROYECTO

```

702         for (list<Punto_de_interes>::iterator itD =
703             puntos_de_interes.begin();
704             itD != puntos_de_interes.end(); itD++) {
705                 archSalida << itD->getTipo_elemento() << " " << itD->getTamano()
706                     << " " << itD->getUnidad_medida() << " "
707                     << itD->getCoordenada_x() << " "
708                     << itD->getCoordenada_y() << endl;
709     }

```

Fig 7. Cambio en la escritura del archivo.

Comentario #4

Observación realizada:

“- Tampoco hace falta agregar automáticamente la extensión, dejemos que el usuario pueda indicar que extension quiere usar”

Corrección realizada: Para que el usuario tenga la libertad de recoger la extensión para su archivo en primera lugar nos dirigimos al comando **guardar**, siendo más precisos en la definición del nombre del archivo mediante la variable **nuevoArch**. Luego de eso cambiamos la palabra formada a partir de los argumentos del comando, quitándole al final de esta el “.txt”, para así terminar solucionando este problema.Para tener una noción más clara del cambio observamos la Fig 8:

```

676     // Definición del nombre de archivo:
677     string nuevoArch{*(arreglo_input + 1) + *(arreglo_input + 2)};

```

Fig 8. Cambio en la asignación de la variable nuevoArch.

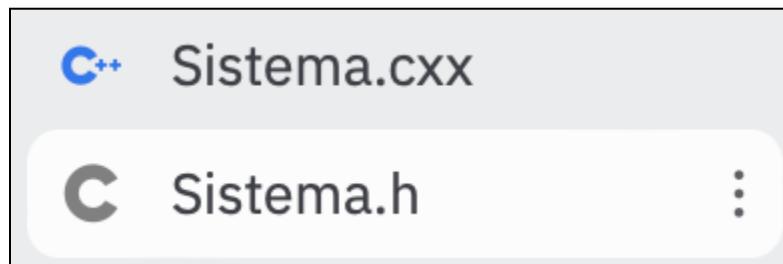
Comentario #5

Observación realizada:

“El main no es un TAD, es un espacio que se utiliza para hacer la interfaz con el usuario. Si se necesita un TAD para albergar cosas relacionadas con el sistema que no sean interfaz, se puede definir un TAD Sistema que sirva de intermediario entre el main y el sistema.”

Corrección realizada: En primera instancia, removimos al main de ser un TAD y lo utilizamos únicamente para ser una interfaz. Es por ello que definimos un TAD Sistema, en el que pusimos las operaciones realizadas en el main que hacían parte del sistema y no de la interfaz, que en este caso fueron las operaciones con la lista de puntos de interés y la lista de comandos. Dichos cambios pueden verse a continuación:

PROYECTO

*Fig 9. Definición del nuevo TAD Sistema.*

```
#include "Sistema.h"

//ingresar elemento a la lista de punto de interes
void Sistema::insertarPuntoInteres(Punto_de_interes puntoInteres)
{
    this->puntos_de_interes.push_back(puntoInteres);
}

//ingresar comando a la lista de comandos
void Sistema::insertarComando(Comando comando)
{
    this->comandos.push_back(comando);
}

//obtener lista de comando
std::list<Comando> Sistema::obtenerComando()
{
    return this->comandos;
}

//obtener lista de punto de interes
std::list<Punto_de_interes> Sistema::obtenerPuntoInteres()
{
    return this->puntos_de_interes;
}

void Sistema::borrarlistaPuntoInteres()
{
    return this->puntos_de_interes.clear();
}
```

Fig 10. Operaciones del TAD Sistema.

PROYECTO

```

if (formatoCorrecto) {
    cant_comandos++;
    // se guarda el comando creado en la lista de comandos
    //INGRESAR SISTEMA
    //comandos.push_back(comand);

    curiosity.insertarComando(comand);
}
}
}

```

Fig 11. Ejemplo de como quedo estas operaciones en el main.

2. Acta de Evaluación tercera entrega

Comentario #1

Observación realizada:

“- El comando guardar elementos parece haberse desconfigurado, pues la información no queda bien en el archivo.”

Corrección realizada: Es importante recordar que en C++ hay varios modos disponibles para abrir y escribir archivos, los modos de apertura de archivos se especifican al abrir un archivo utilizando la función ‘`std::ofstream`’ que se utiliza para abrir archivos en modo de escritura, o ‘`std::fstream`’ el cual se utiliza para abrir archivos en modo de lectura y escritura. Para este caso en particular nos interesa utilizar ‘`std::ofstream`’, del cual se utilizan los métodos más comunes como:

‘`std::ofstream::out`’: Abre un archivo en modo escritura, de forma que si el archivo no existe, este lo crea. Si es el caso contrario su contenido se descarta y se comienza a escribir desde cero.

‘`std::ofstream::trunc`’: Abre un archivo en modo de escritura, de forma que si el archivo existe, su contenido se descarta y se comienza a escribir desde cero. Si en el caso contrario el archivo no existe, se crea uno nuevo.

Teniendo en cuenta las definiciones anteriores, al combinar ‘`std::ofstream::out`’ y ‘`std::ofstream::trunc`’ mediante un ‘|’, el resultado va ser el mismo que especificar únicamente el modo ‘`std::ofstream::out`’, ya que ‘`std::ofstream::trunc`’ ya lo trae implícitamente, y en algunos casos crea conflictos en la escritura y sobreescritura de archivos como se vio en la anterior entrega, así que simplemente se dejó ‘`std::ofstream::out`’ como se puede ver en la *Fig 12*:



PROYECTO

```

678     // Creamos el archivo (si ya existe lo sobreescribe);
679     ofstream archSalida;
680     archSalida.open(nuevoArch, ios::out);

```

Fig 12. Cambio en el modo de escritura del archivo en la función ‘guardar’ .

3. TADs

A continuación muestra la descripción de los TAD utilizados para la elaboración de los comandos del componente 1, 2 y 3:

TAD Grafo

Datos mínimos:

- **vertices**, vector de tipo **template T**, representa el conjunto de vértices del grafo.
- **grafo**, vector de vectores de tipo **template U**, representa la matriz de pesos y adyacencias entre vértices
- **esDirigido**, dato de tipo booleano, define si el grafo es dirigido o no
- **infinito**, dato de tipo **template U**, representa el infinito dentro del grafo, el cual será el valor más grande que U puede tener.

Operaciones:

- + **Grafo()**, crea un **Grafo** con atributos vacíos.
- + **Grafo(vertices, aristas, tipoGrafo)**, crea un **Grafo** fijando los vértices, aristas y el tipo del grafo
- + **~Grafo()**, elimina el actual **Grafo**
- + **esVacio()**, retorna por medio de un Booleano si el grafo tiene información o no (tiene vértices o no)
- + **setEsDirigido(tipoGrafo)**, cambia el valor de **esDirigido** por **tipoGrafo**
- + **getEsDirigido()**, retorna si el grafo es dirigido o no por medio de un booleano
- + **getVertices()**, retorna el vector de vértices del grafo
- + **setAristas(aristas)**, fija la matriz de pesos/adyacencia por la matriz **aristas**
- + **getAristas()**, retorna la matriz de pesos de grafo



PROYECTO

- + **insertarVertice(*vertice*)**, inserta un nuevo vértice al vector de vértices y crea una nueva fila y columna del vértice en la matriz de pesos/aristas. Retorna verdadero si este se pudo ingresar.
- + **insertarArista(*VerticeOrigen, VerticeDestino,peso*)**, inserta una nueva arista con base al **VerticeOrigen**, **VerticeDestino** y el **peso** entre la conexión
- + **cantidadVertices()**, retorna la cantidad de vértices qué tiene el grafo
- + **cantidadAristas()**, retorna la cantidad de aristas o conexiones que tiene el grafo
- + **BuscarPos(*valor*)**, busca la posición de un vértice en el vector de vértices. Retorna la posicion
- + **BuscarVertice(*indice*)**, Busca si el vértice existe dentro del vector de vértices y retorna verdadero si existe
- + **BuscarArista(*VerticeOrigen, VerticeDestino*)**, busca si entre **VerticeOrigen** y **VerticeDestino** existe una conexión directa. Retorna verdadero si existe.
- + **BuscarAristaPorPosicion(*Origen, Destino*)**, en este caso se realiza una búsqueda pasando a través de todas las aristas del grafo en ‘**grafo**’ teniendo en cuenta la posición del vértice origen ‘**Origen**’ y la posición del vértice destino ‘**Destino**’. Retorna verdadero si encuentra un valor diferente a 0 (existe arista), y falso en el caso contrario.
- + **recorridoPlano()**, recorre la lista de vértices ‘**vertices**’ recopilando cada elemento con el que se va topando (sin duplicados).
- + **recorridoProfundidad(*verticeInicio, maxPeso, caminoActual*)**, busca si el vértice ‘**verticeInicio**’ se encuentra dentro de la lista de vértices existentes. En caso de ser verdadero se llama a ‘**dfs(indiceVertice, visitados, pesoActual, PesoPreliminar, maxPeso, caminoActual, caminoPreliminar)**’ para hacer el recorrido correspondiente.
- + **dfs(*indiceVertice, visitados, pesoActual, PesoPreliminar, maxPeso, caminoActual, caminoPreliminar*)**, recursivamente recorre desde el vértice con el índice especificado ‘**indiceVertice**’ todos sus vecinos y dentro de cada vecino repite el proceso hasta que no se pueda avanzar más. Esta evaluación de cada camino se enfoca en escoger el camino con la distancia más grande que encuentre. Si se encuentra el camino más largo, retorna el camino que atravesó de un punto hasta hallar su nodo con distancia más lejana y por ende también retorna el costo o distancia que le costó llegar hasta el otro lado.

PROYECTO

- + **encontrarCaminoMasLargo(puntosMasAlejados, pesoTotal,**
caminoMasLargo), evalua todos los caminos de cada vértice ‘**vertices**’ en el grafo, esto con objetivo de obtener el camino del cual se encuentre la distancia (en término de conexiones) más larga entre dos vértices del grafo.

TAD ArbolUbicarElementos

Datos mínimos:

- **raiz**, apuntador a **NodoUbicarElementos**, representa la raíz del árbol.

Operaciones:

- + **ArbolUbicarElementos()**, crea un **ArbolUbicarElementos** con atributos vacíos (sin raíz).
- + **ArbolUbicarElementos(punto_de_interes)**, crea un **ArbolUbicarElementos** con una raíz por defecto con el valor de **punto_de_interes**.
- + **~ArbolUbicarElementos()**, elimina el actual **ArbolUbicarElementos**.
- + **punto_de_interesRaiz()**, retorna el **punto** actual de la raíz.
- + **obtenerRaiz()**,
- + **fijarRaiz(punto_de_interes)**, cambia el valor de la **raíz** por **punto_de_interes**.
- + **fijarRaiz(nodo)**, cambia la **raíz** por **nodo**.
- + **esVacio()**, retorna el estado actual de la raíz dado verdadero si existe la raíz, falso si no.
- + **insertar(punto_de_interes, puntos_sin_ingresar)**, inserta un nuevo nodo tipo **punto_de_interes** en el árbol y de no ser posible, se guarda en la lista de **puntos_sin_ingresar**.
- + **preOrden()**, muestra el valor de los nodos mientras realiza un recorrido pre-orden sobre el **ArbolUbicarElementos**.
- + **inOrden()**, muestra el valor de los nodos mientras realiza un recorrido in-orden sobre el **ArbolUbicarElementos**.
- + **posOrden()**, muestra el valor de los nodos mientras realiza un recorrido pos-orden sobre el **ArbolUbicarElementos**.

PROYECTO

- + **nivelOrden()**, muestra el valor de los nodos mientras realiza un recorrido nivel-orden sobre el **ArbolUbicarElementos**.
- + **evaluarCuadrante(x_min, x_max, y_min, y_max, Elementos_en_cuadrante)**, recorre el **ArbolUbicarElementos** con el objetivo de buscar aquellos elementos que hagan parte de los límites especificados.

TAD NodoUbicarElementos

Datos mínimos:

- **punto**, tipo **Punto_de_interes**, representa el valor actual del nodo.
- **hijoSupIzq**, apuntador a un **NodoUbicarElementos**, representa el hijo superior izquierdo del nodo.
- **hijoSupDer**, apuntador a un **NodoUbicarElementos**, representa el hijo superior derecho del nodo.
- **hijoInfDer**, apuntador a un **NodoUbicarElementos**, representa el hijo inferior derecho del nodo.
- **hijoInfIzq**, apuntador a un **NodoUbicarElementos**, representa el hijo inferior izquierdo del nodo.

Operaciones:

- **NodoUbicarElementos()**, crea un nodo con atributos vacíos (sin un valor actual del nodo).
- **NodoUbicarElementos(punto_de_interes)**, crea un nodo con el valor de **punto_de_interes** como **punto** por defecto del nodo actual.
- **~NodoUbicarElementos()**, elimina la rama que conecta con el nodo actual.
- **obtenerPunto_de_interes()**, retorna el valor actual que tiene **punto** dentro del nodo.
- **fijarPunto_de_interes(punto_de_interes)**, cambia el valor del **punto** por **punto_de_interes**.
- **obtenerHijoSupIzq()**, retorna un **NodoUbicarElementos** con el hijo izquierdo que tiene **hijoSupIzq**.

PROYECTO

- **obtenerHijoSupDer()**, retorna un **NodoUbicarElementos** con el hijo izquierdo que tiene **hijoSupDer**.
- **obtenerHijoInfDer()**, retorna un **NodoUbicarElementos** con el hijo izquierdo que tiene **hijoInfDer**.
- **obtenerHijoInflIzq()**, retorna un **NodoUbicarElementos** con el hijo izquierdo que tiene **hijoSupIzq**.
- **fijarHijoSupIzq(punto_de_interes)**, le crea un hijo superior izquierdo al nodo actual. Se le asigna por defecto el valor del **punto** por **punto_de_interes**.
- **fijarHijoSupDer(punto_de_interes)**, le crea un hijo superior derecho al nodo actual. Se le asigna por defecto el valor del **punto** por **punto_de_interes**.
- **fijarHijoInfDer(punto_de_interes)**, le crea un hijo inferior derecho al nodo actual. Se le asigna por defecto el valor del **punto** por **punto_de_interes**.
- **fijarHijoInflIzq(punto_de_interes)**, le crea un hijo inferior izquierdo al nodo actual. Se le asigna por defecto el valor del **punto** por **punto_de_interes**.
- **esHoja()**, retorna el estado actual del nodo con base al número de hijos. Se considera hoja si no tiene ningún hijo, de tener 1 o 2 hijos no se considera como hoja.
- **insertar(punto_de_interes, nodo, puntos_sin_ingresar)**, agregar un hijo al **nodo** actual con el valor de **punto_de_interes**, ademas de tambien evaluando en todo momento con **puntos_sin_ingresar** para verificar que no existan nodos duplicados.
- **preOrden(nodo)**, muestra el valor de los nodos mientras realiza un recorrido pre-orden sobre el **nodo**.
- **inOrden(nodo)**, muestra el valor de los nodos mientras realiza un recorrido in-orden sobre el **nodo**.
- **posOrden(nodo)**, muestra el valor de los nodos mientras realiza un recorrido pos-orden sobre el **nodo**.
- **nivelOrden()**, muestra el valor de los nodos mientras realiza un recorrido nivel-orden sobre el **nodo**.
- **evaluarCuadrante(nodo, x_min, x_max, y_min, y_max, Elementos_en_cuadrante)**, realiza un recorrido sobre el **nodo** pasando por todos los hijos del árbol, simultáneamente ejecuta **evaluarCuadrante**.

PROYECTO

- **evaluarNodo(nodo, x_min, x_max, y_min, y_max, Elementos_en_cuadrante)**, comprueba que el **nodo** esté dentro de los intervalos del cuadrante. De ser el caso correcto, lo agrega en **Elementos_en_cuadrante**.

TAD Sistema

Datos mínimos:

- **punto_de_interes**, lista de **Punto_de_Interes**, representa la lista de los puntos de interés del sistema.
- **comandos**, lista de **Comando**, representa la lista de comandos del sistema.

Operaciones:

- **Sistema()**, crea un nuevo sistema vacío.
- **insertarPuntoInteres(puntoInteres)**, inserta el punto de interés dado en la lista de puntos de interés del sistema.
- **insertarComando(comando)**, inserta el comando dado en la lista de comandos del sistema.
- **obtenerComando()**, retorna la lista de comandos del sistema.
- **obtenerPuntoInteres()**, retorna la lista de puntos de interés del sistema.
- **borrarlistaPuntoInteres()**, limpia la lista de puntos de interés del sistema.
- **borrarlistaComando()**, limpia la lista de comandos del sistema.

TAD Comando

Datos mínimos:

- **Tipo**, tipo booleano, determina si el comando es de tipo desplazamiento (si es verdadero) o de tipo análisis (si es falso).
- **CDesplazamiento**, tipo Desplazamiento, guarda los componentes necesarios del comando desplazamiento.
- **CAnalisis**, tipo Análisis, guarda los componentes necesarios del comando desplazamiento.

PROYECTO

Operaciones:

- + **Comando()**, crea un nuevo comando vacío.
- + **Comando(t, des)**, crea un nuevo comando dado qué el comando que se está ingresando es de tipo desplazamiento.
- + **Comando(tipo, ana)**, crea un nuevo comando dado que el comando que se está ingresando es de tipo análisis.
- + **Comando(tip, desp, ana)**, crea un nuevo comando.
- + **setTipo(t)**, fija el tipo de comando a un valor nuevo; verdadero si se quiere cambiar a tipo desplazamiento y falso si se quiere cambiar a tipo análisis.
- + **setCDesplazamiento(desp)**, Fija el comando de desplazamiento a un valor nuevo
- + **setCAnalisis(analisis)**, fija el comando de análisis a un valor nuevo
- + **getTipo()**, retorna el tipo del comando, teniendo como referencia: verdadero si es comando de desplazamiento y falso si es comando de análisis
- + **getCDesplazamiento()**, Retorna el contenido de desplazamiento. Siendo este un dato del tipo Desplazamiento
- + **getCAnalisis()**, Retorna el contenido de análisis. Siendo este un dato del tipo Análisis.



PROYECTO

TAD Desplazamiento

Datos mínimos:

- + **TipoMovimiento**, cadena de caracteres, identifica el tipo de movimiento qué debe realizar el robot. Este puede ser avanzar o girar.
- + **magnitud**, flotante, corresponde a la cantidad de unidades qué debe avanzar o girar el robot.
- + **unidadMedida**, cadena de caracteres, identifica la convención de la magnitud; puede ser cm, grados, etc.

Operaciones:

- **Desplazamiento()**, crea un nuevo Desplazamiento vacío
- **Desplazamiento(TipoMov, mag, unidadMed)**, crea un nuevo desplazamiento con la información solicitada
- **setTipoMovimiento(tipom)**, fija el tipo de movimiento a un valor nuevo
- **setMagnitud(magn)**, fija la magnitud a un valor nuevo
- **setUnidadMedida(und)**, fija la unidad de medida a un valor nuevo
- **getTipoMovimiento()**, retorna el contenido del tipo de movimiento
- **getMagnitud()**, retorna el contenido de la magnitud
- **getUnidadMedida()**, retorna el contenido de la unidad de medida

TAD Punto_de_interes

Datos mínimos:

- + **tipo_componente**, cadena de caracteres, representa los diferentes tipos de elementos que pueden ser reconocidos por el robot (roca, cráter, montículo, duna).
- + **tamano**, número real, representa al valor de la dimensión del elemento.
- + **unidad_medida**, cadena de caracteres, representa la unidad de medida con la cual se realizó la medición del tamaño del elemento.
- + **coordenada_x**, número real, representa la posición del elemento sobre el eje x en el plano cartesiano.

PROYECTO

- + **coordenada_y**, número real, representa la posición del elemento sobre el eje y en el plano cartesiano.

Operaciones :

- **Dato_de_interes(tipo_elemento, tamano, unidad_medida, coordenada_x, coordenada_y)**, crea un nuevo elemento teniendo en cuenta la información solicitada.
- **eliminarDato_de_interes()**, elimina un elemento existente.
- **setTipo_elemento(tipo_elemento)**, fija el tipo de elemento.
- **setTamano(tamano)**, fija el tamaño del elemento.
- **setUnidad_medida(unidad_medida)**, fija la unidad de medida del elemento.
- **setCoordenada_x(coordenada_x)**, fija la posición sobre el eje x en el plano cartesiano del elemento.
- **setCoordenada_y(coordenada_y)**, fija la posición sobre el eje x en el plano cartesiano del elemento.
- **getTipo_elemento()**, obtiene el tipo de elemento.
- **getTamano()**, obtiene el tamaño del elemento.
- **getUnidad_medida()**, obtiene la unidad de medida del elemento.
- **getCoordenada_x()**, obtiene la posición sobre el eje x en el plano cartesiano del elemento.
- **getCoordenada_y()**, obtiene la posición sobre el eje x en el plano cartesiano del elemento.
- **printPunto()**, retorna una cadena de caracteres con la información del elemento en cuestión

TAD Analysis

Datos mínimos:

- + **tipo_analisis**, cadena de caracteres, representa los diferentes tipos de análisis que puede realizar el robot (fotografiar, composición, perforar).
- + **objeto**, cadena de caracteres, nombre del elemento sobre el cual se hace el análisis y el comentario (el comentario es opcional).



PROYECTO

- + **comentario**, cadena de caracteres, valor opcional que permite agregar información sobre el análisis a realizar o el elemento que se analizará (el comentario se debe hacer en comillas simples).

Operaciones :

- **Analisis(tipo_analisis, objeto, comentario)**, crea un nuevo análisis teniendo en cuenta la información solicitada.
- **eliminar_Analisis(tipo_analisis, objeto, comentario)**, elimina un análisis existente.
- **setTipo_analisis(tipo_analisis)**, fija el tipo de análisis.
- **setObjeto(objeto)**, fija el nombre del elemento como objeto de análisis
- **setComentario(comentario)**, fija un comentario que brinde información del análisis o del elemento.
- **getTipo_analisis()**, obtiene el tipo de análisis.
- **getObjeto()**, obtiene el nombre del elemento como objeto de análisis.
- **get Comentario()**, obtiene el comentario realizado al objeto de análisis.

4. Diagrama de TADS.

Con base a la definición de los TADs, se realizó el diagrama de relación entre estos. Como se puede evidenciar, Comando puede estar compuesto de ‘Desplazamiento’ y/o ‘Analisis’ y ‘Sistema’ está compuesto por una lista de ‘Comandos’ y otra lista de ‘Punto_de_interes’. Así mismo, el ‘nodoUbicarElementos’ se compone por un dato de tipo ‘Punto_de_interes’ y el ‘arbolUbicarElementos’ se compone por estos nodos. Finalmente ‘Grafo’ se compone por ‘Punto_de_interes’, los cuales serán sus vértices:

PROYECTO

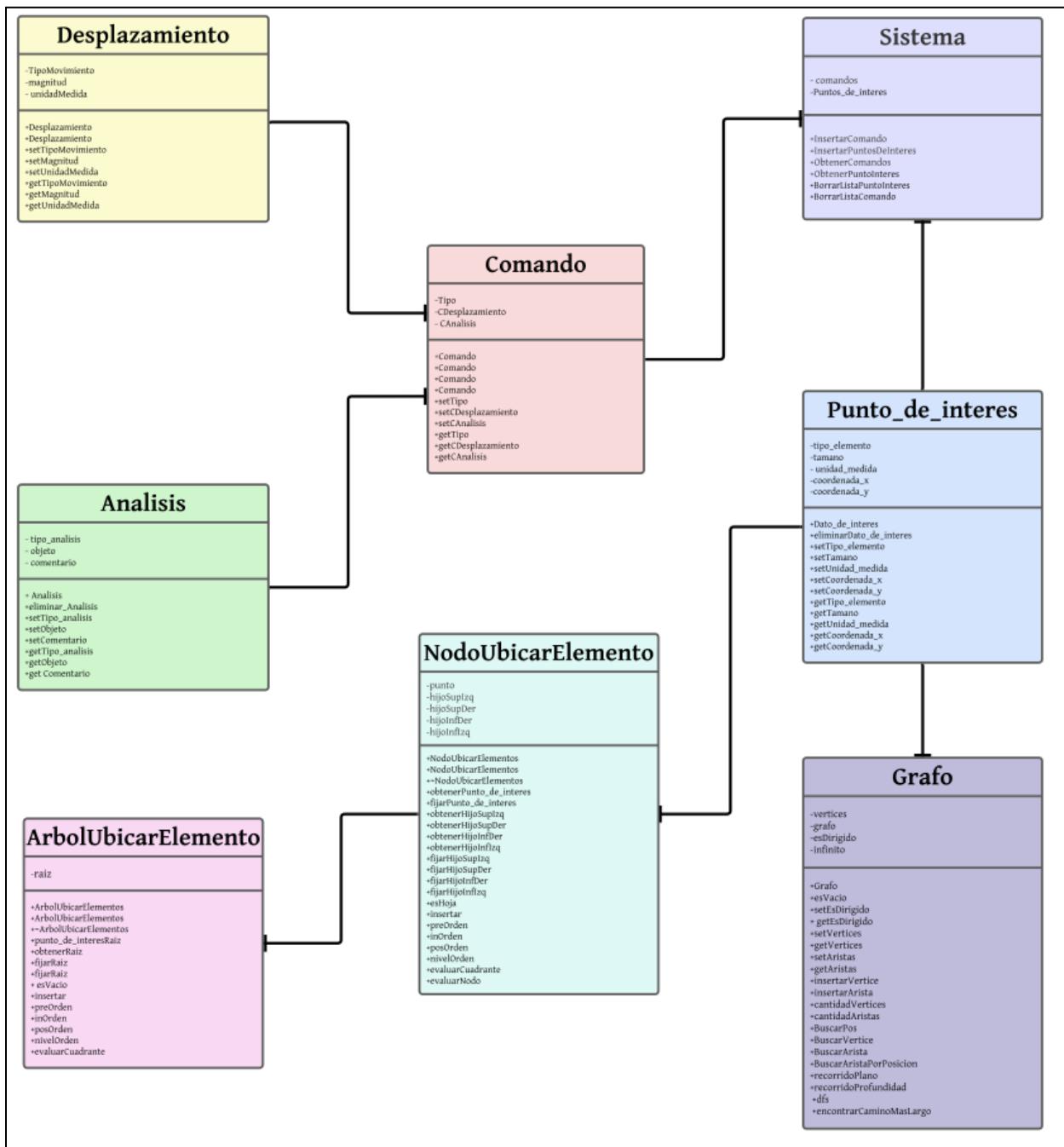


Fig 13. Diagrama de TADs

PROYECTO

5. Funciones/comandos:

A continuación se realizó la descripción de entradas, salidas y condiciones de cada uno de los comandos. Además luego de establecerlos, se mostró con un diagrama de flujo o de actividad el funcionamiento paso a paso del comando respectivo.

cargar_comandos nombre_archivo

La siguiente tabla muestra las entradas, salidas y condiciones del comando:

Tabla 1. Condiciones, entradas y salidas del comando cargar_comandos

Nombre:	comando cargar_comandos
Objetivo en Contexto	Cargar (o sobreescribir) en memoria (en la lista de comandos) los comandos de movimiento y análisis que se encuentren en un archivo.
ENTRADAS	Se recibe el comando de cargar_comandos y la lista de comandos
Pre-Condiciones (Escenario de éxito)	El comando “cargar_comandos” debe tener exactamente dos palabras, donde la primera debe ser “cargar_comandos” y la segunda palabra al final debe contener *“.<tipo_archivo>”; es decir el tipo de archivo qué se quiere leer. El archivo a leer debe ser de tipo txt.
SALIDAS	<u>si el archivo no existe o falla en abrir:</u> se debe mostrar en pantalla: “<nombre_archivo> no se encuentra o no puede leerse” <u>Si el archivo está vacío:</u> se debe mostrar en pantalla: “<nombre_archivo> no contiene elementos.” <u>Si el archivo no está vacío:</u> se debe mostrar en pantalla: “<nombre_archivo> no contiene elementos.” “<n> comando(s) cargado(s) correctamente desde <nombre_archivo>”
Post-Condiciones (Escenario de éxito)	La cantidad n de elementos leídos debe ser de tipo entero sin signo

PROYECTO

Otras condiciones	cada línea del archivo que se lee debe contener la estructura de desplazamiento [tipo_mov magnitud unidad_med] o la de analisis [analisis tipo_analisis objeto comentario(opcional)] El contenido que se lea del archivo debe sobreescribir totalmente el contenido de la lista de comandos Deberá volver a pedir al usuario un nuevo comando a ingresar; imprimiendo al inicio “\$”
--------------------------	---

Diagrama del funcionamiento del comando

El siguiente diagrama de flujo muestra el funcionamiento del comando.



PROYECTO

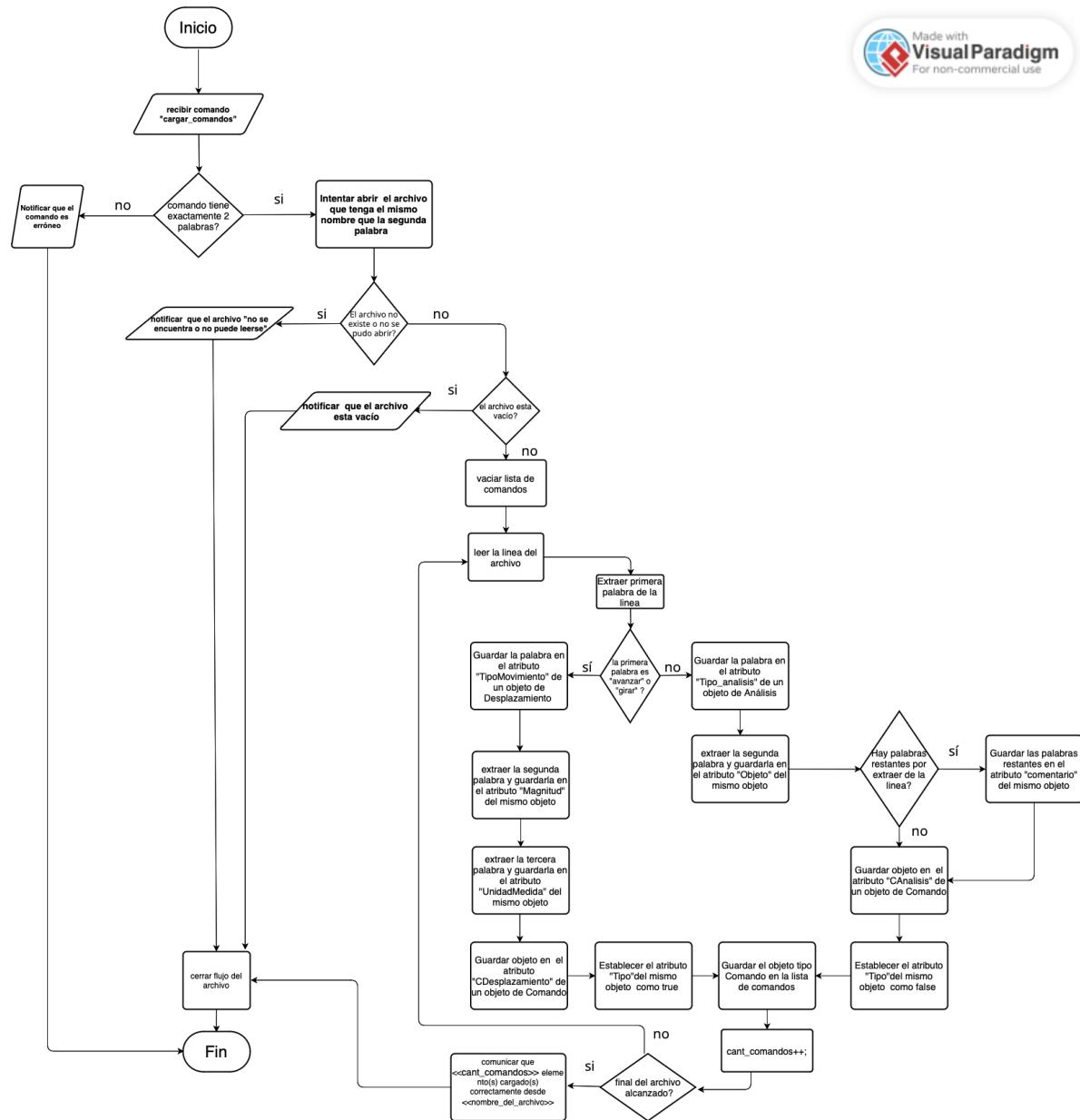


Fig 14. Diagrama de flujo del funcionamiento de la función del comando cargar_comandos
A Continuación se muestra la primera parte del mismo diagrama en una mayor escala:

PROYECTO

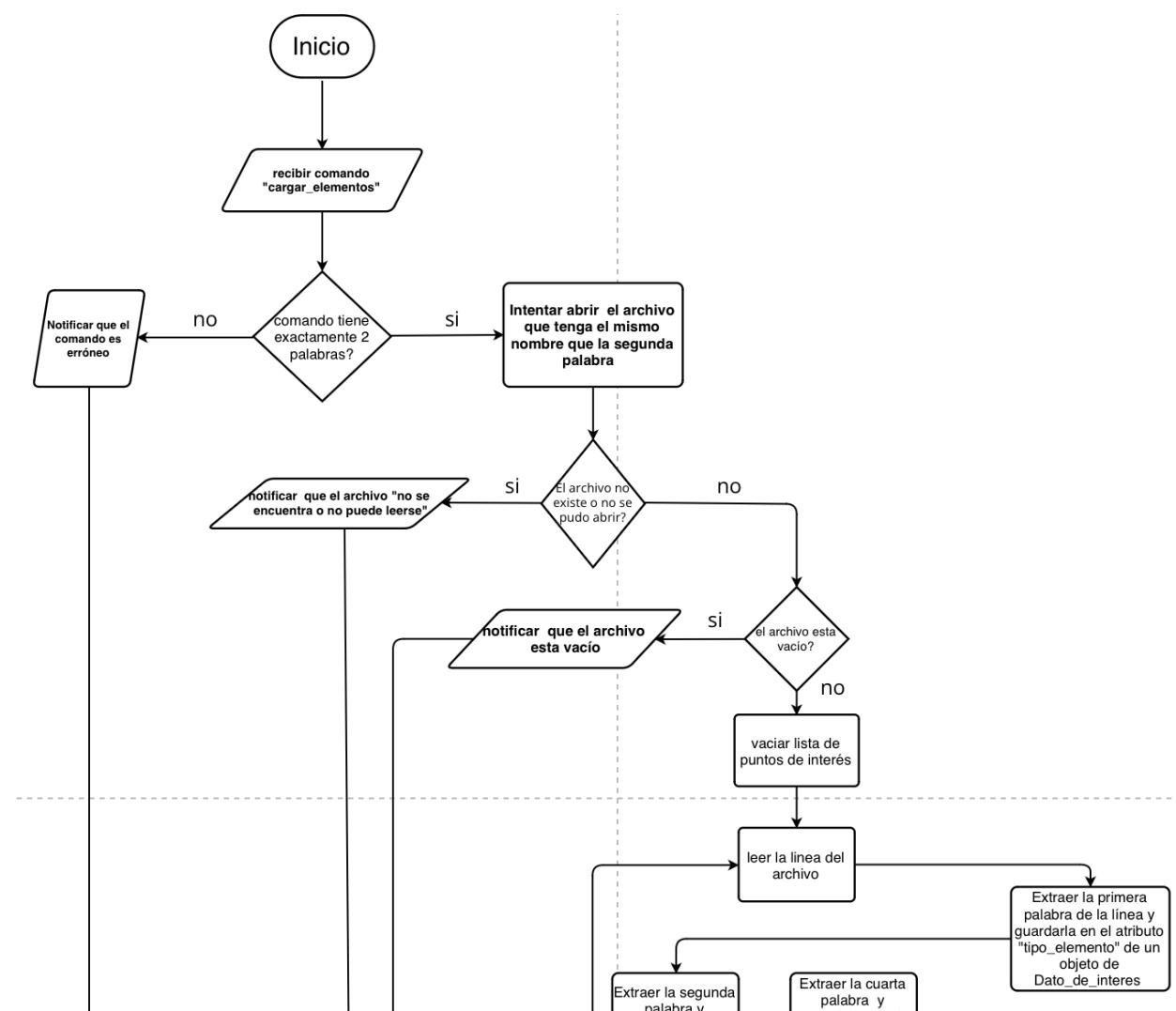


Fig 15. Primera parte del diagrama de flujo de cargar_comandos

A Continuación se muestra la segunda parte del mismo diagrama en una mayor escala:



PROYECTO

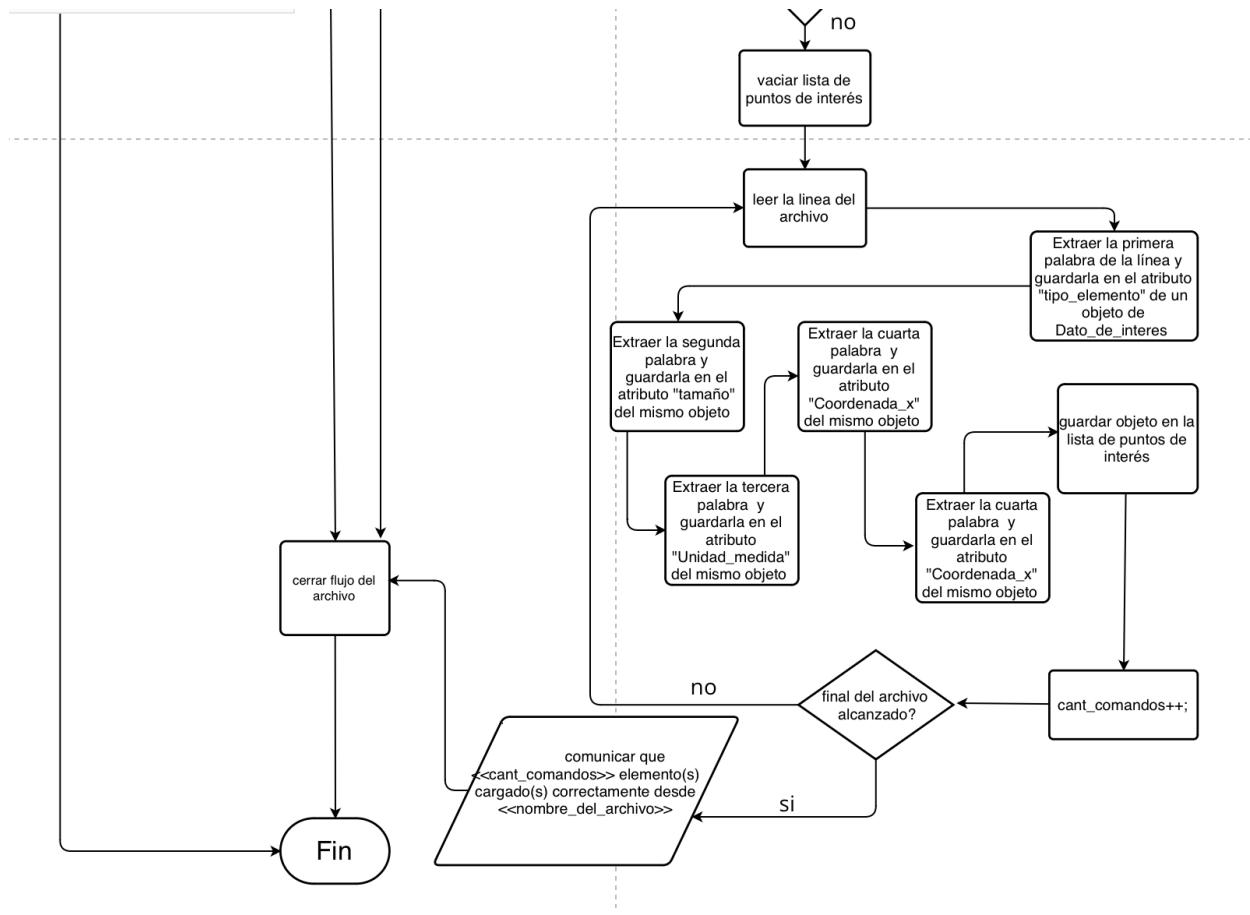


Fig 16. Segunda parte del diagrama de flujo de cargar_comandos

cargar_elementos nombre_archivo

La siguiente tabla muestra las entradas, salidas y condiciones del comando:

Tabla 2. Condiciones, entradas y salidas del comando cargar_elementos

Nombre:	comando cargar_elementos
Objetivo en Contexto	Cargar (o sobreescribir) en memoria (en la lista de puntos_de_interes) los comandos de cargar elementos que se encuentren en un archivo.
ENTRADAS	Se recibe el comando de cargar_elementos y la lista de datos_de_interes

PROYECTO

<p>Pre-Condiciones <i>(Escenario de éxito)</i></p>	<p>El comando “cargar_elementos” debe tener exactamente dos palabras, donde la primera debe ser “cargar_elementos” y la segunda palabra al final debe contener *“.<tipo_archivo>”; es decir el tipo de archivo que se quiere leer.</p> <p>El archivo a leer debe ser de tipo txt.</p>
<p>SALIDAS</p>	<p><u>si el archivo no existe o falla en abrir:</u> se debe mostrar en pantalla: “<nombre_archivo> no se encuentra o no puede leerse”</p> <p><u>Si el archivo está vacío:</u> se debe mostrar en pantalla: “<nombre_archivo> no contiene elementos.”</p> <p><u>Si el archivo no está vacío:</u> “<n> elemento(s) cargado(s) correctamente desde <nombre_archivo>”</p>
<p>Post-Condiciones <i>(Escenario de éxito)</i></p>	<p>La cantidad n de elementos leídos debe ser de tipo entero sin signo</p>
<p>Otras condiciones</p>	<p>cada línea del archivo que se lee debe contener la estructura de elemento [tipo_comp tamaño unidad_med coordX coordY]</p> <p>El contenido que se lea del archivo debe sobreescribir totalmente el contenido de la lista de puntos de interés.</p> <p>Deberá volver a pedir al usuario un nuevo comando a ingresar; imprimiendo al inicio “\$”</p>

Diagrama del funcionamiento del comando

El siguiente diagrama de flujo muestra el funcionamiento del comando.

PROYECTO

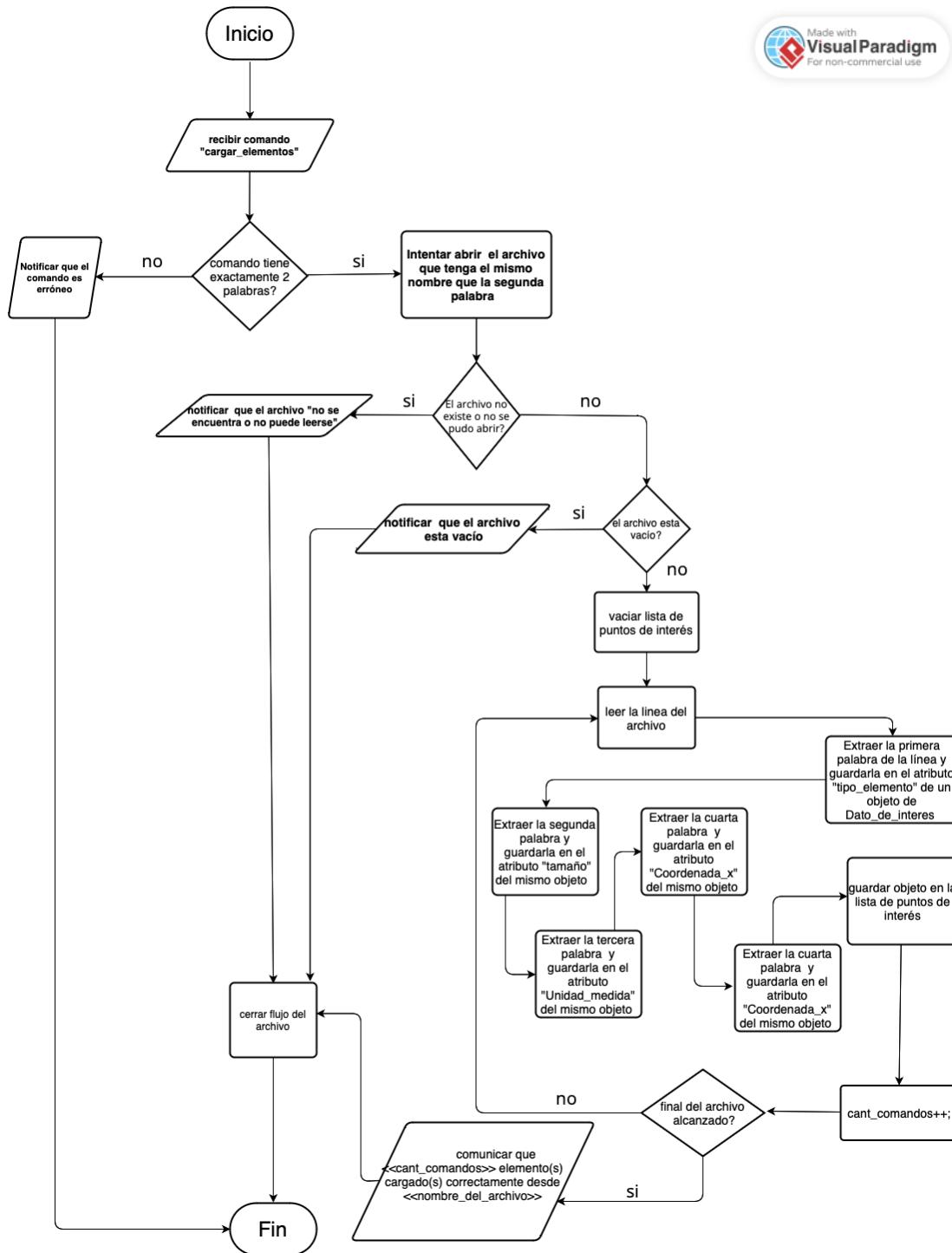


Fig 17. Diagrama de flujo del funcionamiento de la función del comando cargar_elementos .



PROYECTO

agregar_movimiento tipo_mov magnitud unidad_med

La siguiente tabla muestra las entradas, salidas y condiciones del comando:

Tabla 3. Condiciones, entradas y salidas del comando agregar_movimiento

Nombre:	comando agregar_movimiento
Objetivo en Contexto	Cargar en memoria (en la lista de comandos) el nuevo comando de movimiento ingresado por el usuario
ENTRADAS	Se recibe el comando agregar_movimiento y la lista de comandos
Pre-Condiciones (Escenario de éxito)	<p>El comando “agregar_movimiento” debe tener mínimo el tipo de movimiento, la magnitud de dicho movimiento y la unidad de medida correspondiente.</p> <p>El tipo de movimiento solo puede ser “avanzar” o “girar”.</p>
SALIDAS	<p><u>Si el comando es correcto:</u> Se debe mostrar por pantalla: “El comando de movimiento ha sido agregado exitosamente”</p> <p><u>Si el comando es incorrecto (ya sea por palabra faltante o palabra esperada en tipo_mov):</u> Se debe mostrar por pantalla: “La información del movimiento no corresponde a los datos esperados (tipo, magnitud, unidad)”</p>
Post-Condiciones (Escenario de éxito)	Únicamente se deberá mostrar en pantalla “El comando de movimiento ha sido agregado exitosamente”.
Otras condiciones	<p>La información del comando se debe guardar en la lista de comandos.</p> <p>Los comandos qué no pueden ingresarse son los que tienen ubicaciones (x,y) exactamente iguales a elementos qué ya estan ubicados en el quadtree</p> <p>Deberá volver a pedir al usuario un nuevo comando a ingresar; imprimiendo al inicio “\$”</p>

PROYECTO

Diagrama del funcionamiento del comando

El siguiente diagrama de flujo muestra el funcionamiento del comando.

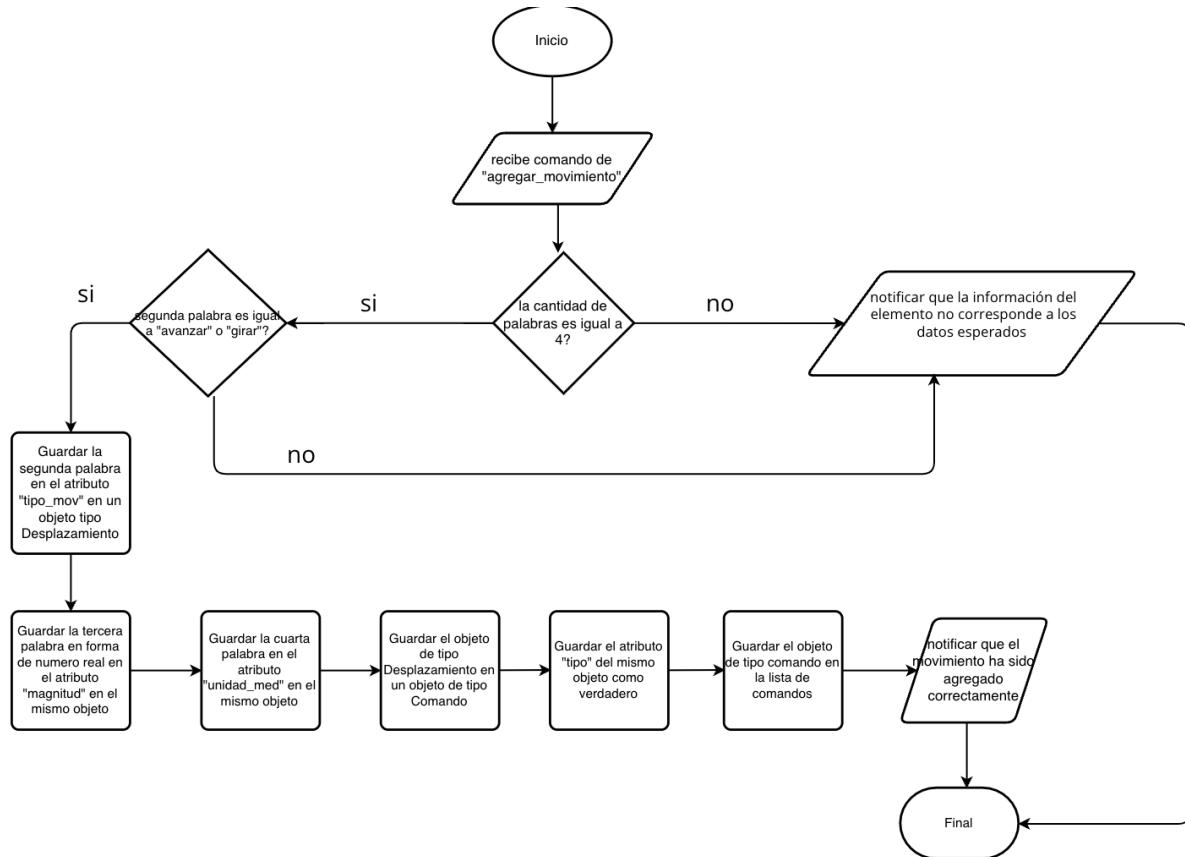


Fig 18. Diagrama de flujo del funcionamiento de la función del comando agregar_analisis

agregar_analisis tipo_analisis objeto_comentario

La siguiente tabla muestra las entradas, salidas y condiciones del comando:

Tabla 4. Condiciones, entradas y salidas del comando agregar_analisis

Nombre:	comando agregar_analisis
Objetivo en Contexto	Cargar en memoria (en la lista de comandos) el nuevo comando de análisis ingresado por el usuario

PROYECTO

ENTRADAS	Se recibe el comando de agregar_analisis y la lista de comandos
Pre-Condiciones <i>(Escenario de éxito)</i>	<p>El comando “agregar_analisis” debe tener mínimo el tipo de análisis y el objeto (tres palabras).</p> <p>El comentario qué puede ser añadido, es de tamaño de palabras variable, pero debe comenzar y terminar en comillas simples.</p> <p>El tipo de análisis debe ser únicamente: "fotografiar", "composición", "perforar"</p>
SALIDAS	<p><u>Si el comando es correcto:</u> Se debe mostrar en pantalla: "El comando de análisis ha sido agregado exitosamente."</p> <p><u>Si el comando es incorrecto (ya sea por palabra faltante o palabra esperada en tipo_analisis incorrecta):</u> Se debe mostrar en pantalla: "La información del análisis no corresponde a los datos esperados (tipo, objeto, comentario)."</p>
Post-Condiciones <i>(Escenario de éxito)</i>	Únicamente se deberá mostrar en pantalla "El comando de análisis ha sido agregado exitosamente."
Otras condiciones	La información del comando se debe guardar en la lista de comandos. Deberá volver a pedir al usuario un nuevo comando a ingresar; imprimiendo al inicio “\$”

Diagrama del funcionamiento del comando

El siguiente diagrama de flujo muestra el funcionamiento del comando.

PROYECTO

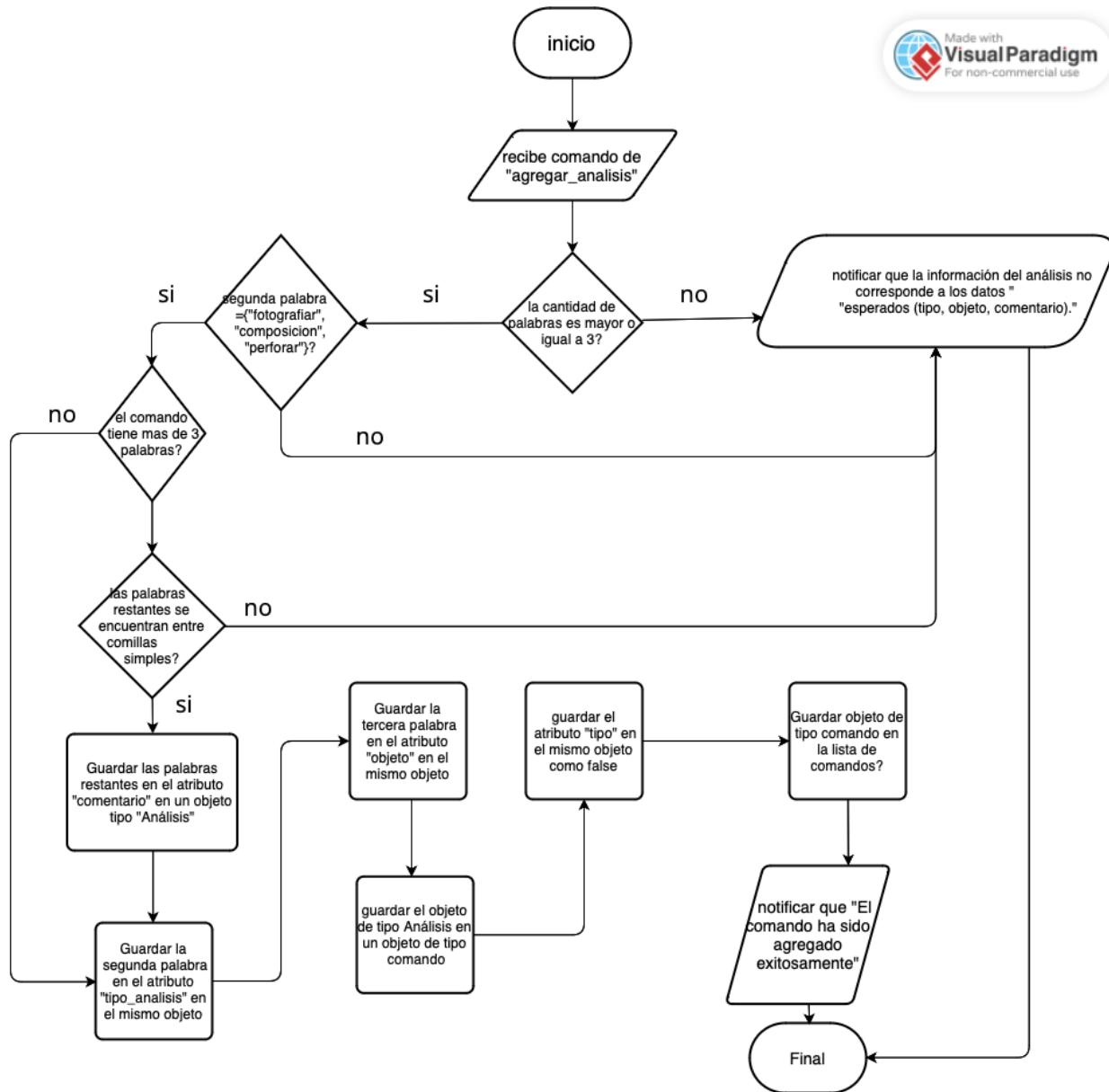


Fig 19. Diagrama de flujo del funcionamiento de la función del comando agregar_analisis

agregar_elemento tipo_comp tamaño unidad_med coordX coordY

La siguiente tabla muestra las entradas, salidas y condiciones del comando:

PROYECTO

Tabla 5. Condiciones, entradas y salidas del comando agregar_elemento

Nombre:	comando agregar_elemento
Objetivo en Contexto	Cargar en memoria (en la lista de puntos de interés) el nuevo elemento ingresado por el usuario
ENTRADAS	Se recibe el comando de agregar_elemento y la lista de puntos de interés.
Pre-Condiciones (Escenario de éxito)	El comando “agregar_elemento” debe tener mínimo el tipo de componente, el tamaño, la unidad de medida, la coordenada X y la coordenada Y. El tipo de componente debe ser uno entre roca, cráter, montículo o duna.
SALIDAS	<u>Si el comando es correcto:</u> Se debe mostrar por pantalla: “El elemento ha sido agregado exitosamente” <u>Si el comando es incorrecto (ya sea por palabra faltante o palabra esperada en tipo_comp incorrecta):</u> Se debe mostrar por pantalla: “La información del elemento no corresponde a los datos esperados. (tipo, tamaño, unidad, x, y)”
Post-Condiciones (Escenario de éxito)	Únicamente se deberá mostrar en pantalla "El elemento ha sido agregado exitosamente".
Otras condiciones	La información del comando se debe guardar en la lista de puntos de interés. Deberá volver a pedir al usuario un nuevo comando a ingresar; imprimiendo al inicio “\$”

Diagrama del funcionamiento del comando

El siguiente diagrama de flujo muestra el funcionamiento del comando.

PROYECTO

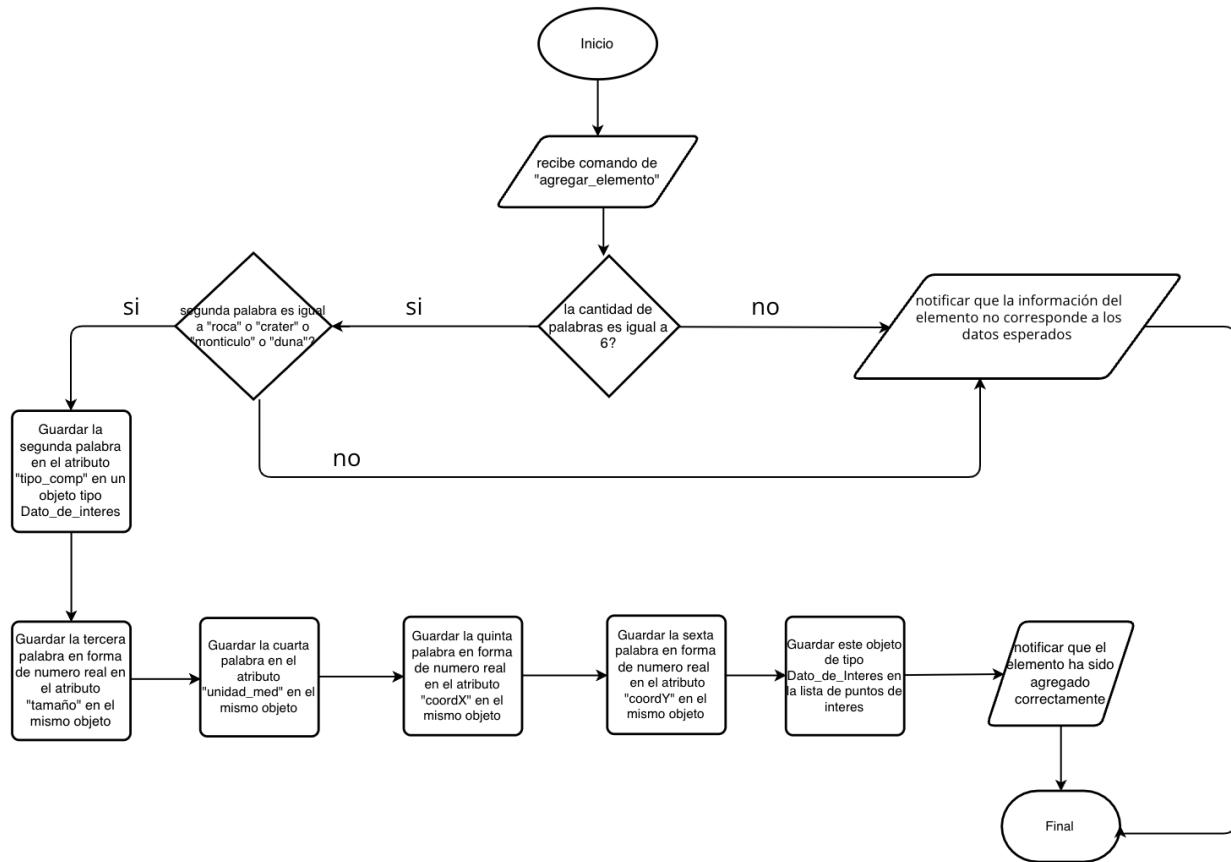


Fig 20. Diagrama de flujo del funcionamiento de la función del comando agregar_elemento

guardar tipo_archivo nombre_archivo

La siguiente tabla muestra las entradas, salidas y condiciones del comando:

Tabla 6. Condiciones, entradas y salidas del comando guardar. magnitud

Nombre:	comando guardar
Objetivo en Contexto	Guarda en el archivo “ nombre_archivo ” la información solicitada de acuerdo a “ tipo_archivo ”.
ENTRADAS	Recibe como entradas “ guardar ”, “ tipo_archivo ” y “ nombre_archivo ” .

PROYECTO

<p>Pre-Condiciones <i>(Escenario de éxito)</i></p>	<p>tipo_archivo: El tipo de archivo debe ser del tipo [Comando] o [Puntos de interés].</p> <p>nombre_archivo: El nombre del archivo no puede ser únicamente numérico o de caracteres especiales, debe ser una cadena de caracteres.</p>
<p>SALIDAS</p>	<p><u>Si el comando es correcto</u> (flujo de salida estándar):</p> <ul style="list-style-type: none"> - La información ha sido guardada en nombre_archivo . <p><u>Si el comando es incorrecto</u> (flujo de salida estándar):</p> <ul style="list-style-type: none"> - Error guardando en nombre_archivo .
<p>Post-Condiciones <i>(Escenario de éxito)</i></p>	<p>Crea exitosamente (o sobreescribe) un archivo de tipo “tipo_archivo” y con nombre “nombre_archivo.txt”</p>

Diagrama del funcionamiento del comando

El siguiente diagrama de flujo muestra el funcionamiento del comando.

PROYECTO

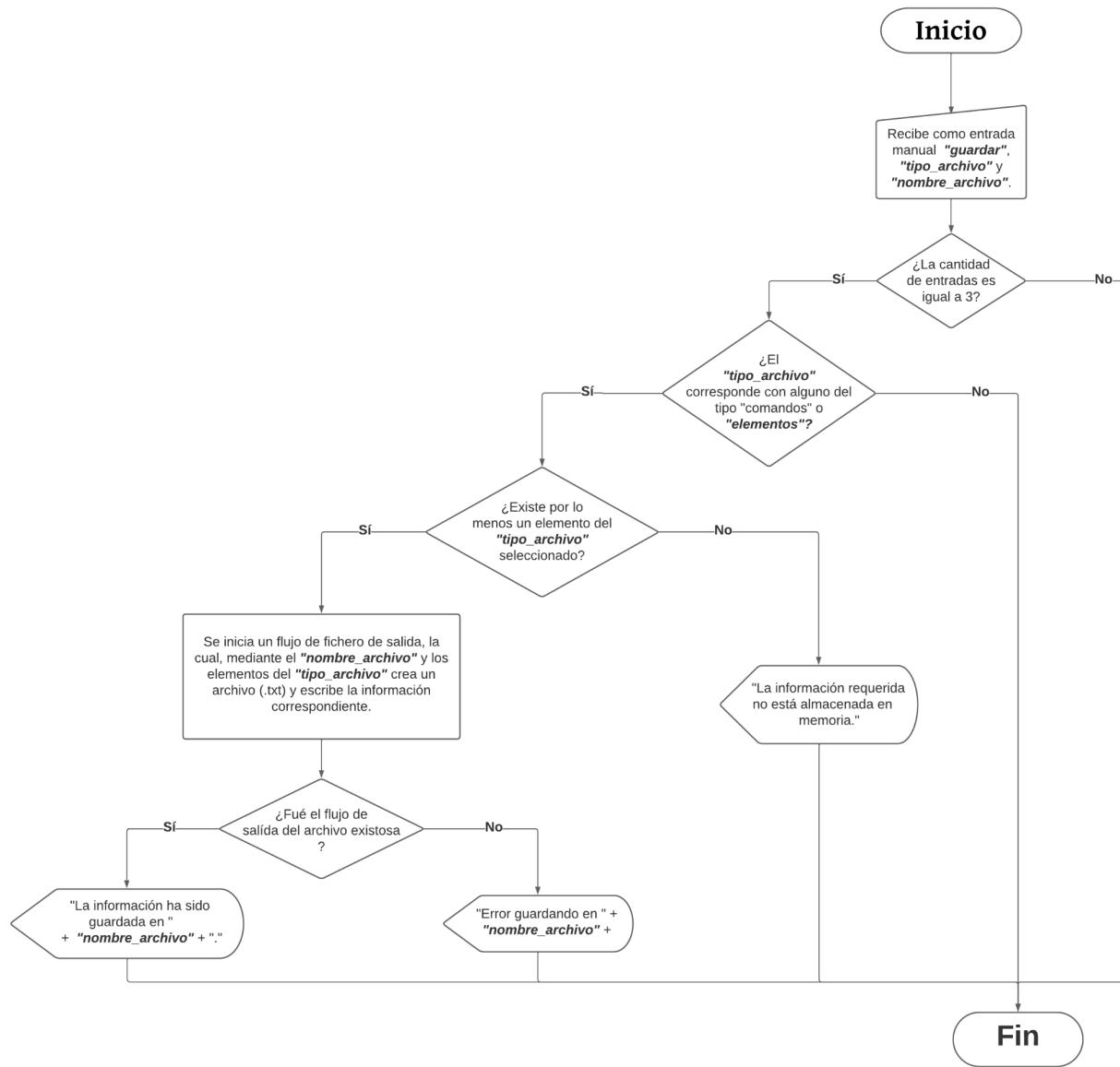


Fig 21. Diagrama de flujo del funcionamiento de la función del comando guardar tipo_archivo nombre_archivo

simular_comandos coordX coordY

La siguiente tabla muestra las entradas, salidas y condiciones del comando:

PROYECTO

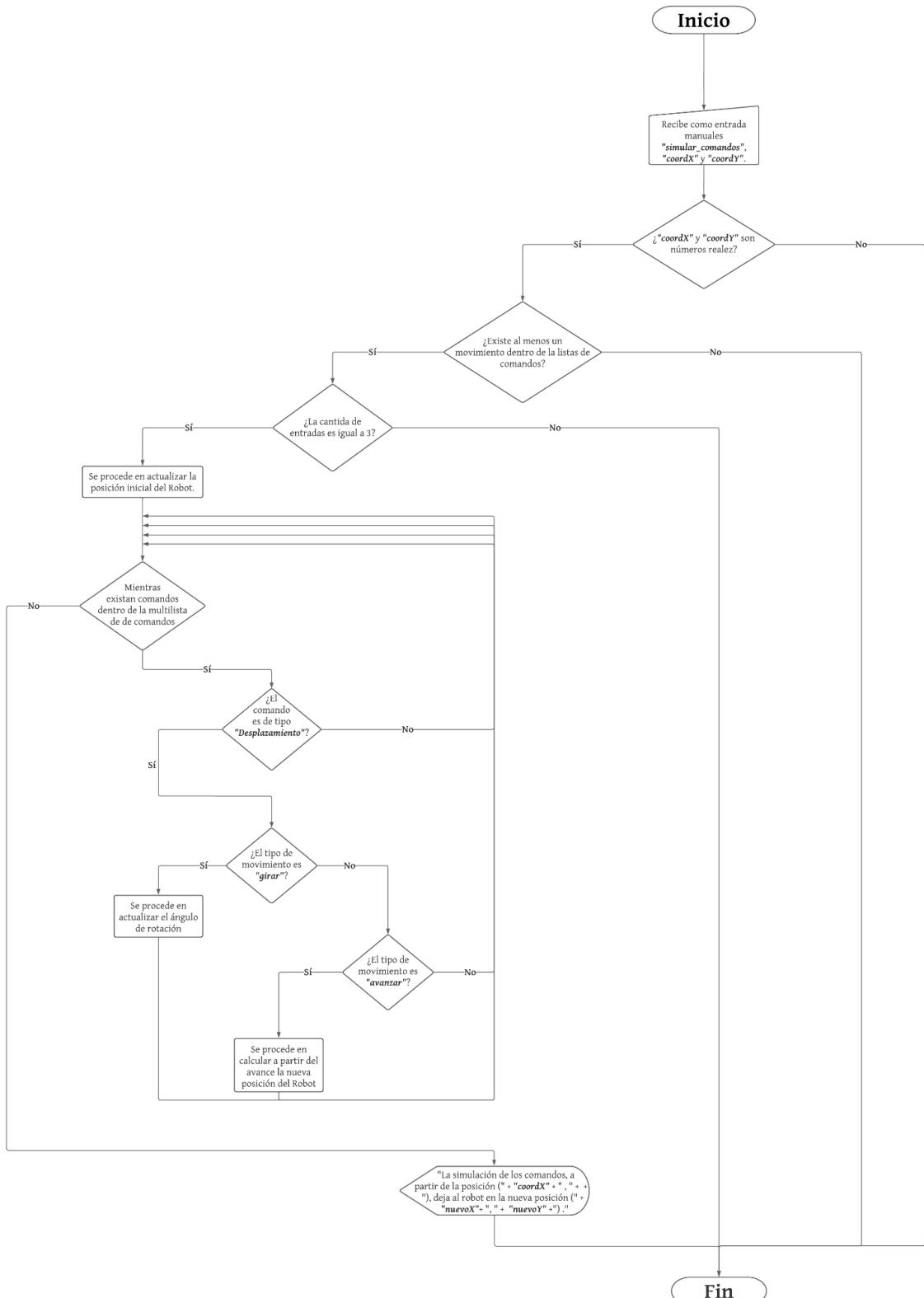
Tabla 7. Condiciones, entradas y salidas del comando *simular_comandos*.

Nombre:	comando <i>simular_comandos</i>
Objetivo en Contexto	Simula el resultado de los comandos de movimiento que se le enviarán al robot “Curiosity” desde la tierra, facilitando de esta manera la validación de la nueva posición en la que podría ubicarse.
ENTRADAS	Recibe como entradas “ <i>simular_comandos</i> ”, “ <i>coordX</i> ” y “ <i>coordY</i> ” .
Pre-Condiciones (Escenario de éxito)	<p><i>coordX</i>: Recibe únicamente un dato numérico del tipo real o entero.</p> <p><i>coordY</i>: Recibe únicamente un dato numérico del tipo real o entero.</p>
SALIDAS	<p><u>Si el comando es correcto</u> (flujo de salida estándar):</p> <ul style="list-style-type: none"> - La simulación de los comandos, a partir de la posición (<i>coordX</i>, <i>coordY</i>), deja al robot en la nueva posición (<i>nuevoX</i>, <i>nuevoY</i>). <p><u>Si no hay información</u> (flujo de salida estándar):</p> <ul style="list-style-type: none"> - La información requerida no está almacenada en memoria. <p><u>Si el comando es incorrecto</u> (flujo de salida estándar):</p> <ul style="list-style-type: none"> - Error en la simulación, parámetros incorrectos.
Post-Condiciones (Escenario de éxito)	Realiza exitosamente la simulación, de forma que actualiza las variables “ <i>nuevoX</i> ” y “ <i>nuevoY</i> ” con los nuevos datos “ <i>coordX</i> ” y “ <i>coordY</i> ” .

Diagrama del funcionamiento del comando

El siguiente diagrama de flujo muestra el funcionamiento del comando:

PROYECTO



PROYECTO

Fig 22. Diagrama de flujo del funcionamiento de la función del comando guardar tipo_archivo simular_comandos

Salir

La siguiente tabla muestra las entradas, salidas y condiciones del comando:

Tabla 8. Condiciones, entradas y salidas del comando salir

Nombre:	comando salir
Objetivo en Contexto	Terminar el funcionamiento del programa
ENTRADAS	Recibe el comando de “salir”
Pre-Condiciones (Escenario de éxito)	El comando de salir debe contener únicamente la palabra salir.
SALIDAS	<u>Si el comando es correcto:</u> Se termina el programa <u>Si el comando es incorrecto:</u> se muestra en pantalla: "El comando no existe."
Post-Condiciones (Escenario de éxito)	Solo se deberá terminar el programa.

Diagrama del funcionamiento del comando

El siguiente diagrama de actividad muestra el funcionamiento del comando:

PROYECTO

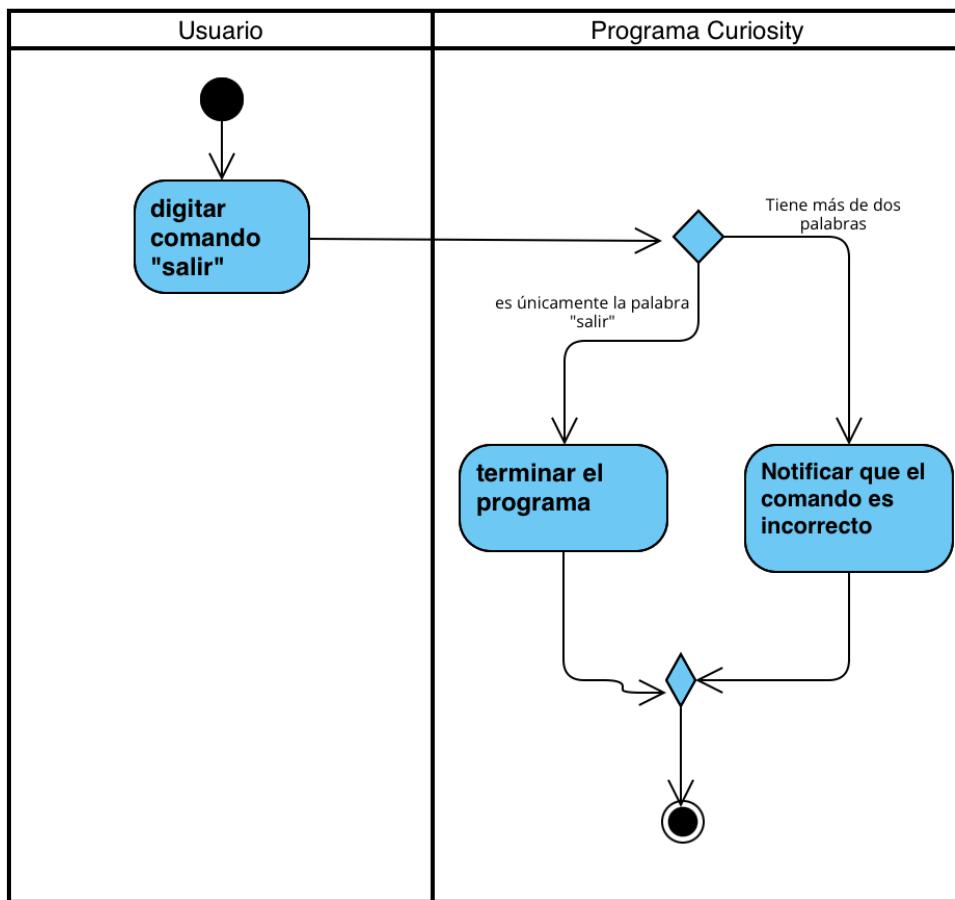


Fig 23. Diagrama de actividad del funcionamiento del comando salir

Ayuda

La siguiente tabla muestra las entradas, salidas y condiciones del comando:

Tabla 9. Condiciones, entradas y salidas del comando ayuda

Nombre:	comando ayuda
Objetivo en Contexto	Mostrar información de los comandos válidos o de la estructura de un comando válido en particular.

PROYECTO

ENTRADAS	Recibe el comando de “ayuda”
Pre-Condiciones <i>(Escenario de éxito)</i>	<p>El comando únicamente puede contener de 1 a dos palabras, donde la primera siempre deberá ser “ayuda”.</p> <p>Si tiene dos palabras, la segunda debe ser exactamente alguna de las siguientes:</p> <ul style="list-style-type: none"> ● cargar_comandos ● cargar_elementos ● agregar_movimiento ● agregar_analisis ● agregar_elemento ● guardar ● simular_comandos ● salir ● ubicar_elementos ● en_cuadrante ● crear_mapa ● ruta_mas_larga
SALIDAS	<p><u>si el comando es únicamente “ayuda”:</u> se debe mostrar en pantalla la lista de los posibles comandos que se pueden utilizar:</p> <ul style="list-style-type: none"> ● “cargar_comandos ● cargar_elementos ● agregar_movimiento ● agregar_analisis ● agregar_elemento ● guardar ● simular_comandos ● salir ● ubicar_elementos ● en_cuadrante ● crear_mapa ● ruta_mas_larga” <p><u>Si el comando contiene 2 palabras y es correcto:</u> Se debe mostrar en pantalla la estructura del comando que se está consultado y una descripción de lo que hace el comando.</p>

PROYECTO

	<u>Si el comando es incorrecto:</u> se debe mostrar en pantalla: “El comando de ayuda no existe”;;
<i>Post-Condiciones</i> <i>(Escenario de éxito)</i>	Deberá volver a pedir al usuario un nuevo comando a ingresar; imprimiendo al inicio “\$”

Diagrama del funcionamiento del comando

El siguiente diagrama de actividad muestra el funcionamiento del comando

PROYECTO

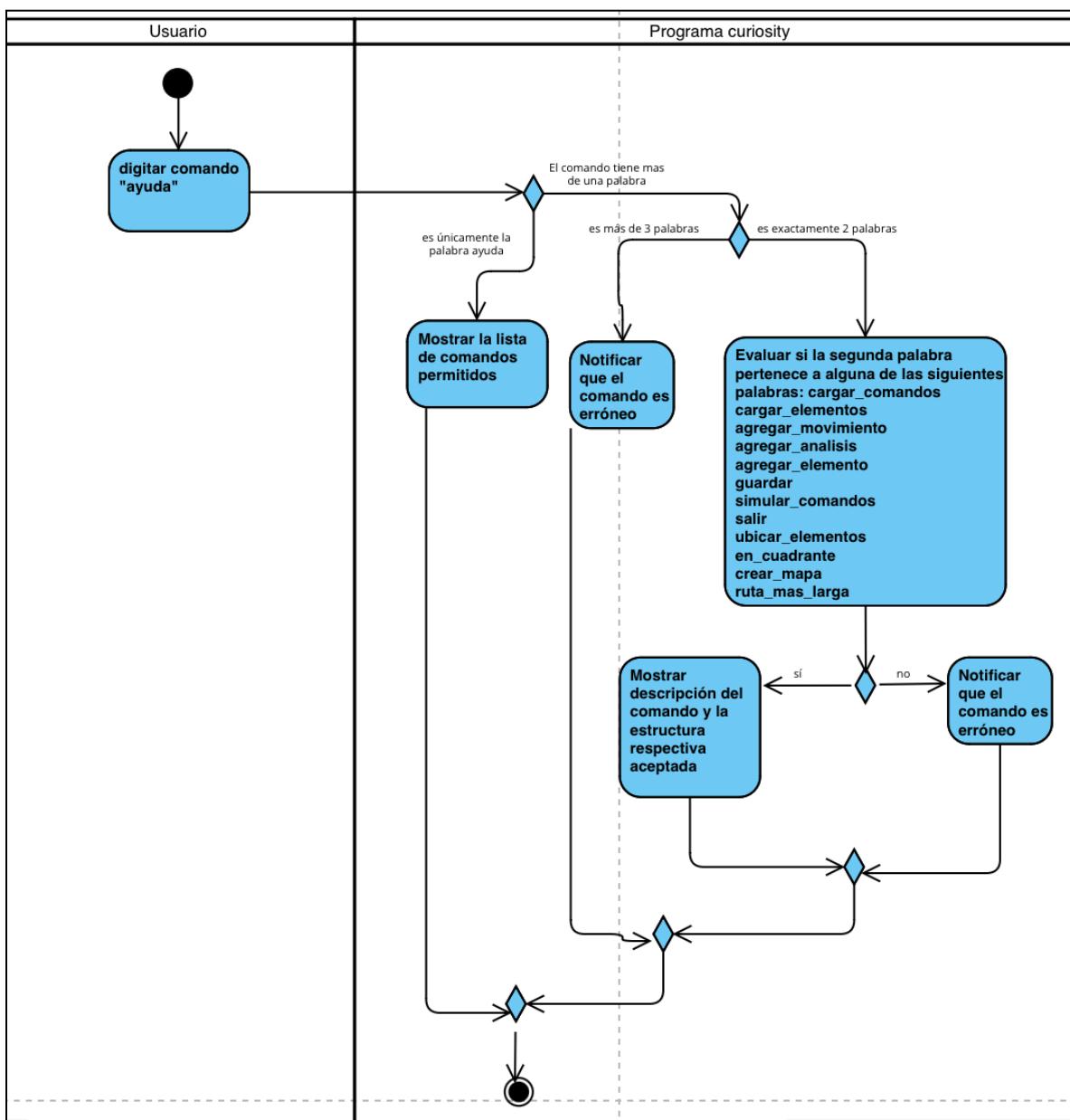


Fig 24. Diagrama de actividad del funcionamiento de la función del comando ayuda

Ubicar_elementos

La siguiente tabla muestra las entradas, salidas y condiciones del comando:

Tabla 10. Condiciones, entradas y salidas del comando ubicar_elementos

PROYECTO

Nombre:	ubicar_elementos
Objetivo en Contexto	ubicar los elementos cargados en memoria en un quadtree
ENTRADAS	Se recibe el comando de ubicar_elementos, el árbol quadtree ElementosUbicados y el sistema Curiosity
Pre-Condicion es (Escenario de éxito)	<ol style="list-style-type: none"> 1. El comando solo debe tener una palabra, la cual es ubicar_elementos 2. La lista de puntos_de_interes del sistema Curiosity debe contener datos
SALIDAS	<p><u>si la lista puntos_de_interes no contiene datos</u> se debe mostrar en pantalla: "La información requerida no está almacenada en memoria."</p> <p><u>si el formato del comando es incorrecto</u> se debe mostrar en pantalla: "La información del comando no corresponde a la estructura esperada."</p> <p><u>si hubo elementos que no pudieron ingresarse</u> se debe mostrar en pantalla: "Los siguientes elementos no pudieron procesarse adecuadamente:" y se muestra el/los elementos que no se ingresaron.</p> <p><u>si todos los elementos se ingresaron sin problema</u> se debe mostrar en pantalla: "Los elementos han sido procesados exitosamente."</p>
Post-Condicion es (Escenario de éxito)	<p>El árbol ElementosUbicados debe guardar adecuadamente los elementos cargados en memoria. Únicamente se debería mostrar en pantalla: "La información del comando no corresponde a la estructura esperada."</p>

PROYECTO

<p>Otras condiciones</p>	<ol style="list-style-type: none"> 1. Cuando se acabe la ejecución del comando, el programa deberá volver a pedir al usuario un nuevo comando a ingresar; imprimiendo al inicio “\$” 2. Los comandos que no pueden ingresarse son los que tienen ubicaciones (x,y) exactamente iguales a elementos que ya están ubicados en el quadtree 3. Para ingresar cada elemento como nodo en el árbol se seguirá las siguientes condiciones: <ol style="list-style-type: none"> a. si "elemento Coordenada x < nodo Coordenada x" y "elemento Coordenada y >= nodo Coordenada y", entonces el elemento se añadirá en un nodo superior izquierdo b. si ("elemento Coordenada x >= nodo Coordenada_x" y "elemento Coordenada y > nodo Coordenada y") o ("elemento Coordenada x > nodo Coordenada x" y "elemento Coordenada y >= nodo Coordenada y"), entonces el elemento se añadirá en un nodo superior derecho c. si "elemento Coordenada x < nodo Coordenada x" y "elemento Coordenada y < nodo Coordenada y", entonces el elemento se añadirá en un nodo inferior izquierdo d. si ("elemento Coordenada x > nodo Coordenada_x" y "elemento Coordenada y < nodo Coordenada y") o ("elemento Coordenada x >= nodo Coordenada x" y "elemento Coordenada y < nodo Coordenada y"), entonces el elemento se añadirá en un nodo inferior derecho
---------------------------------	--

Diagrama del funcionamiento del comando

El siguiente diagrama de flujo muestra el funcionamiento del comando.

PROYECTO

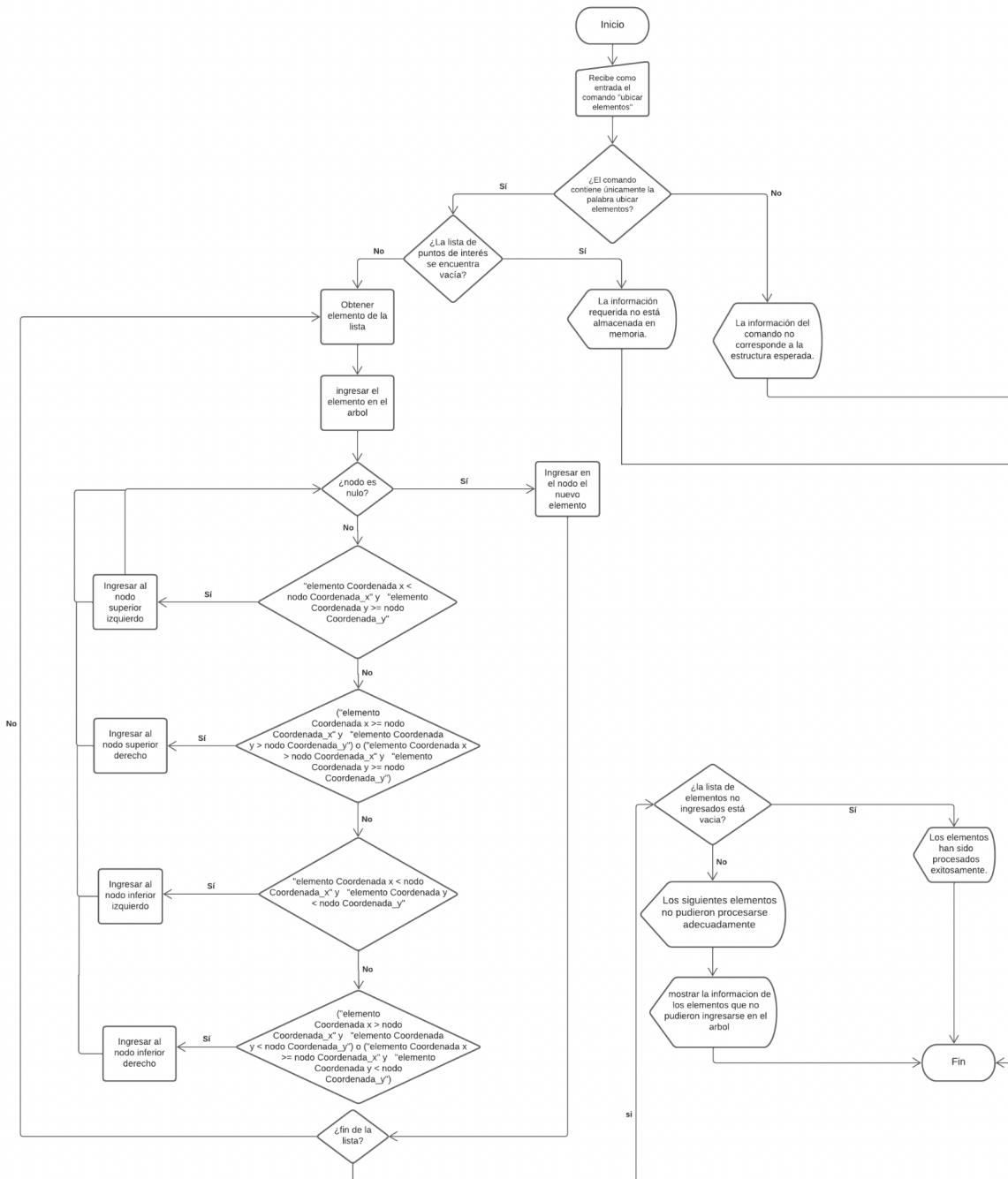


Fig 25. Diagrama de flujo del funcionamiento de la función del comando ubicar_elementos



PROYECTO

en_cuadrante coordX1 coordX2 coordY1 coordY2

La siguiente tabla muestra las entradas, salidas y condiciones del comando:

Tabla 11. Condiciones, entradas y salidas del comando en_cuadrante

Nombre:	en_cuadrante
Objetivo en Contexto	Encontrar los elementos o componentes que están dentro del cuadrante geográfico descrito por los límites de coordenadas x y y.
ENTRADAS	Se recibe el comando de en_cuadrante, el arbol quadtree ElementosUbicados
Pre-Condicioness (Escenario de éxito)	<ol style="list-style-type: none"> El comando debe tener 5 palabras, el comando de en_cuadrante, la coordenada x1, la coordenada x2, la coordenada y1 y la coordenada y2. Los elementos deben ser ubicados, es decir, haber ejecutado antes el comando ubicar_elementos. Los límites de coordenadas deben garantizar que la coordeanda x1 sea menor a la coordenada x2 y que la coordenada y1 sea menor a la coordenada y2.
SALIDAS	<p><u>si los elementos aún no han sido ubicados</u> se debe mostrar en pantalla: "Los elementos no han sido ubicados todavía (con el comando ubicar_elementos)."</p> <p><u>si el formato del comando es incorrecto</u> se debe mostrar en pantalla: "La información del cuadrante no corresponde a los datos esperados (x_min, x_max, y_min, y_max).."</p> <p><u>si no se encuentran elementos o componentes en el cuadrante especificado</u> se debe mostrar en pantalla: "No existen elementos en el cuadrante especificado"</p> <p><u>si se encuentran elementos o componentes</u> se debe mostrar en pantalla:"Los elementos ubicados en el</p>

PROYECTO

	cuadrante solicitado son: " y se muestra en pantalla el o los elementos encontrados.
<i>Post-Condiciones (Escenario de éxito)</i>	Se debe retornar la lista con el o los elementos encontrados en el cuadrante especificado y únicamente se debería mostrar en pantalla: “Los elementos ubicados en el cuadrante solicitado son: ...” seguido de cada elemento de esta lista.
<i>Otras condiciones</i>	<ol style="list-style-type: none"> 1. Cuando se acabe la ejecución del comando, el programa deberá volver a pedir al usuario un nuevo comando a ingresar; imprimiendo al inicio “\$”

Diagrama del funcionamiento del comando

El siguiente diagrama de flujo muestra el funcionamiento del comando.

PROYECTO

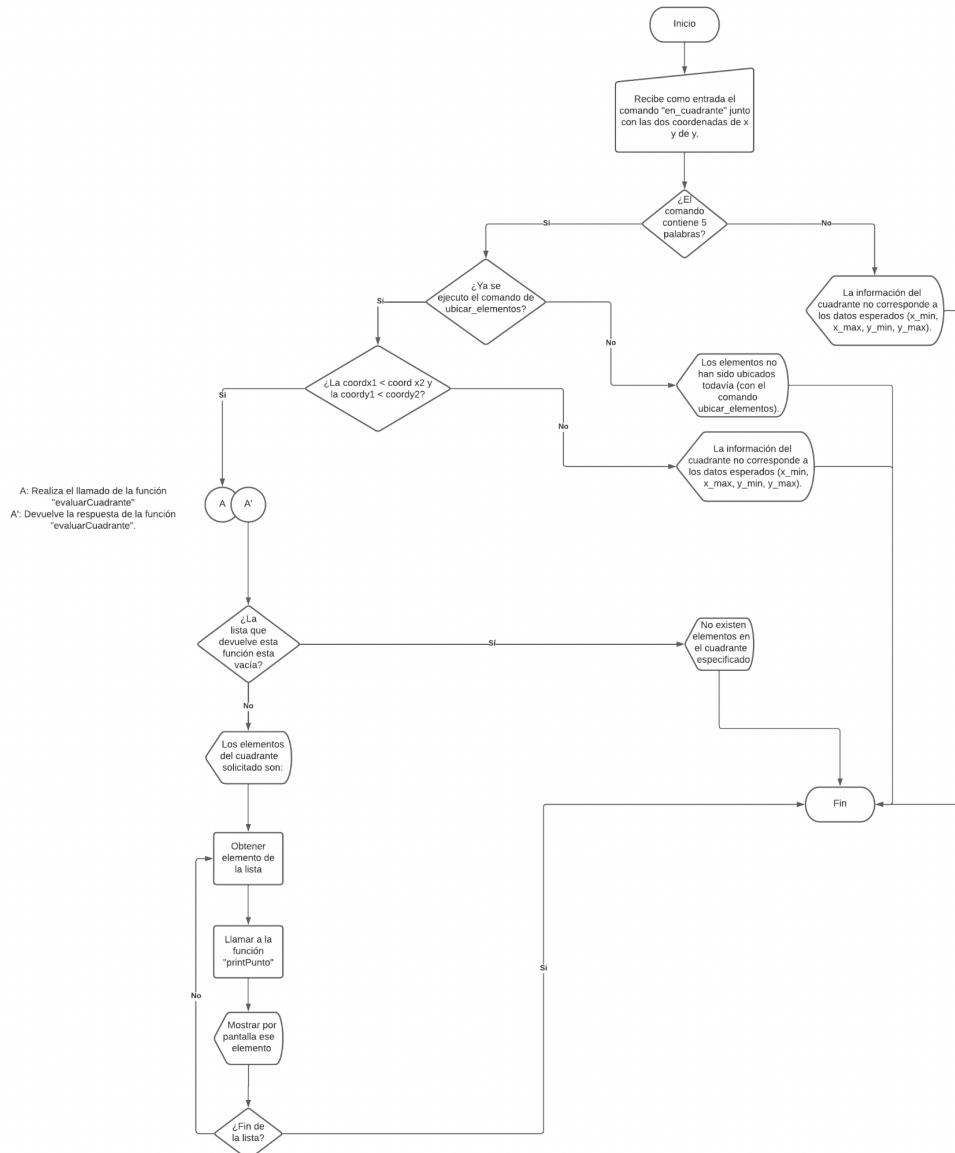


Fig 26. Diagrama de flujo del funcionamiento de la función del comando en_cuadrante

crear_mapa_coeficiente_conectividad

La siguiente tabla muestra las entradas, salidas y condiciones del comando:



PROYECTO

Tabla 12. Condiciones, entradas y salidas del comando **crear_mapa**

Nombre:	crear_mapa
Objetivo en Contexto	Construir un grafo con los elementos de la lista de puntos de interés y con base a la distancia euclíadiana entre ellos
ENTRADAS	Se recibe el comando de crear_mapa, el grafo y sistema
Pre-Condiciones (Escenario de éxito)	<ol style="list-style-type: none"> 1. El comando debe tener 2 palabras, el comando de crear_mapa y el coeficiente de conectividad 2. Deben existir elementos en el sistema 3. No existen elementos en la misma posición 4. el coeficiente de conectividad es un número menor a uno y mayor a cero
SALIDAS	<p><u>si no hay elementos en el sistema</u> se debe mostrar en pantalla: "La información requerida no está almacenada en memoria."</p> <p><u>si el formato del comando es incorrecto</u> se debe mostrar en pantalla: "Formato de comando erróneo"</p> <p><u>si hay dos elementos en la misma posición</u> se debe mostrar en pantalla: "Existen 2 elementos que se encuentran en la misma posición"</p> <p><u>si todos los elementos se ingresaron en el grafo</u> se debe mostrar en pantalla: El mapa se ha generado exitosamente. Cada elemento tiene << cantVecinos << vecinos.</p>
Post-Condiciones (Escenario de éxito)	<ul style="list-style-type: none"> • Solo se debe mostrar en pantalla El mapa se ha generado exitosamente. Cada elemento tiene << cantVecinos << vecinos. • cantVecinos debe ser el resultado de redondear la multiplicación entre el índice de conectividad y la cantidad de elementos • el grafo debe contener todos los elementos y los pesos obtenidos en el proceso

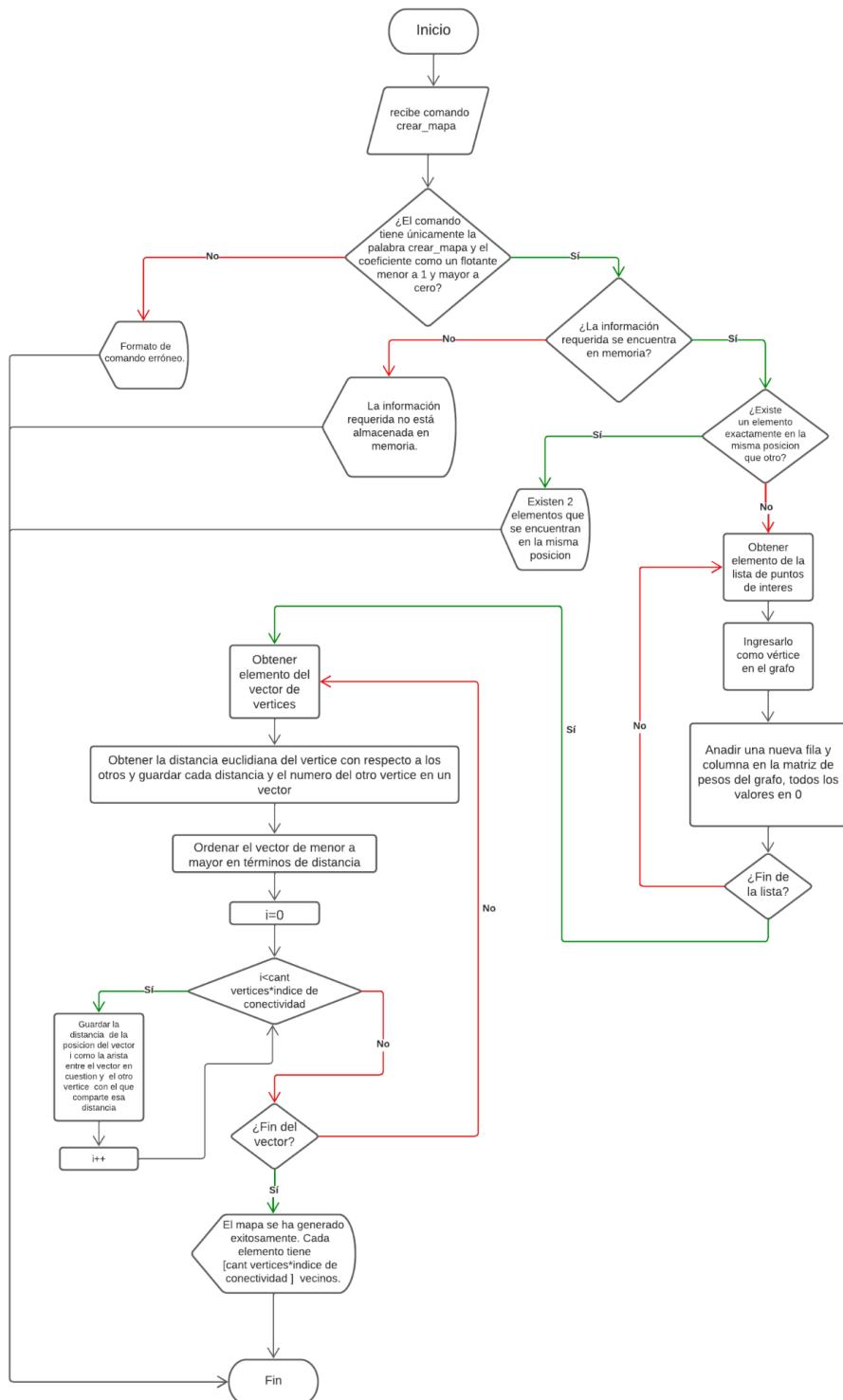
PROYECTO

<i>Otras condiciones</i>	<ol style="list-style-type: none">1. Cuando se acabe la ejecución del comando, el programa deberá volver a pedir al usuario un nuevo comando a ingresar; imprimiendo al inicio “\$”.2. Los vecinos de cada vértice deben ser el resultado de redondear la multiplicación entre el índice de conectividad y la cantidad de elementos.3. Los vecinos de cada grafo se definen con base a la distancia euclíadiana del vértice con respecto a los otros. Donde se escogerán los primeros <<cantVecinos>> con menor distancia euclíadiana con respecto al vértice.4. Los pesos del grafo son en términos de distancias euclidianas.
--------------------------	--

Diagrama del funcionamiento del comando

El siguiente diagrama de flujo muestra el funcionamiento del comando.

PROYECTO



PROYECTO

Fig 27. Diagrama de flujo del funcionamiento de la función del comando crear_mapa

ruta_mas_larga

Tabla 13. Condiciones, entradas y salidas del comando ruta_mas_larga

Nombre:	ruta_mas_larga
Objetivo en Contexto	Permite identificar los dos componentes más alejados entre sí de acuerdo a las conexiones generadas.
ENTRADAS	Se recibe el comando de ruta_mas_larga, el grafo y sistema
Pre-Condiciones (Escenario de éxito)	<ol style="list-style-type: none"> 1. El comando debe tener 1 palabras, el comando de ruta_mas_larga 2. Ya se debió haber creado el mapa del sistema
SALIDAS	<p><u>si no el mapa no ha sido generado</u> se debe mostrar en pantalla: "El mapa no ha sido generado todavía (con el comando crear_mapa)."</p> <p><u>si el formato del comando es incorrecto</u> se debe mostrar en pantalla: "Formato de comando erróneo"</p> <p><u>si el resultado es exitoso</u> se debe mostrar en pantalla: Los puntos de interés más alejados entre sí son ... y ... La ruta que los conecta tiene una longitud total de ... y pasa por los siguientes elementos:</p>
Post-Condiciones (Escenario de éxito)	<ul style="list-style-type: none"> • Solo se debe mostrar en pantalla: Los puntos de interés más alejados entre sí son ... y ... La ruta que los conecta tiene una longitud total de ... y pasa por los siguientes elementos:

PROYECTO

Otras condiciones	<p>5. Cuando se acabe la ejecución del comando, el programa deberá volver a pedir al usuario un nuevo comando a ingresar; imprimiendo al inicio “\$”</p> <p>6. Los elementos más alejados entre sí se retornan de acuerdo a las conexiones generadas, más no a la distancia euclídea.</p>
--------------------------	---

Diagrama del funcionamiento del comando

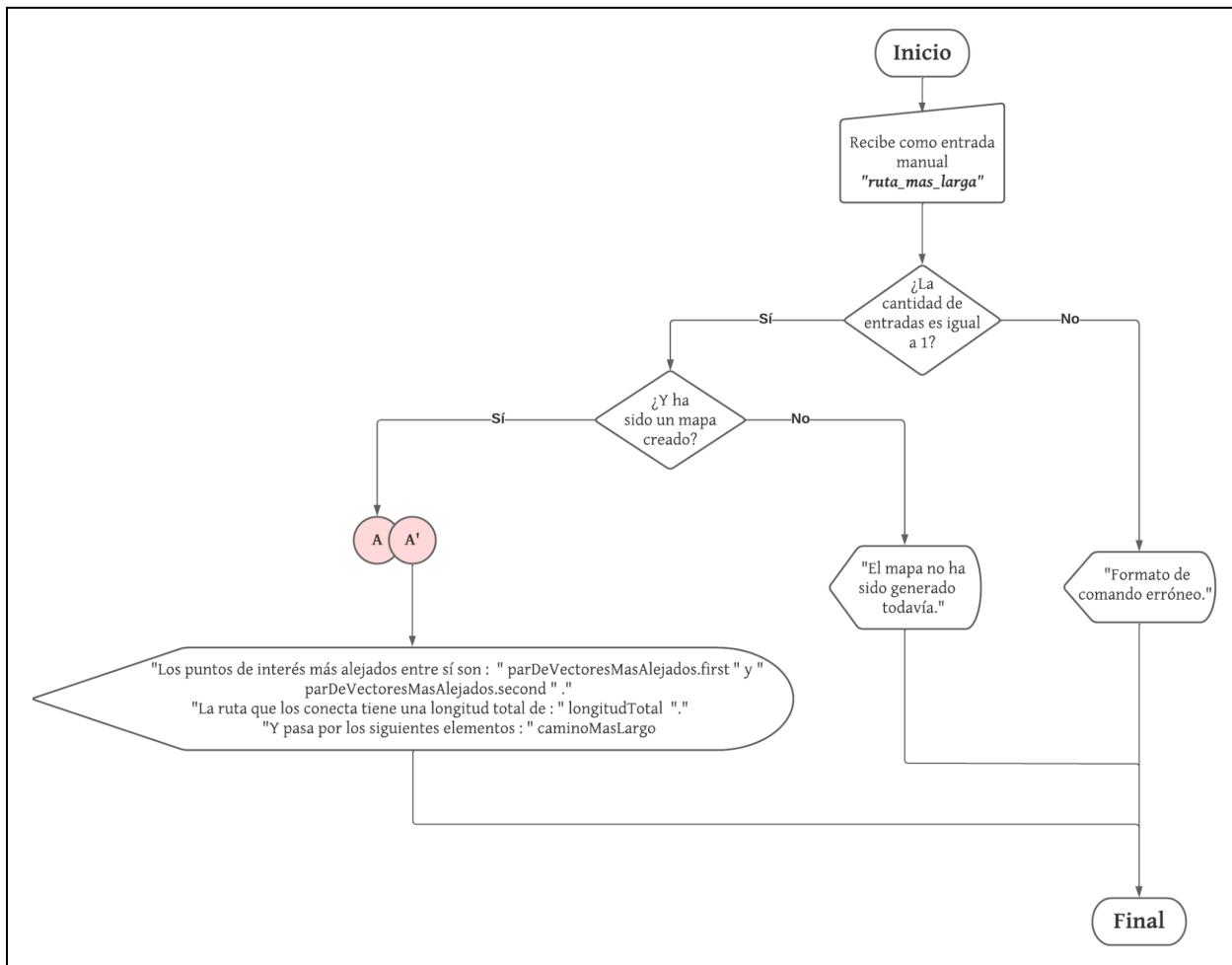


Fig 28. Diagrama de flujo del funcionamiento de la función del comando ruta_mas_larga

PROYECTO

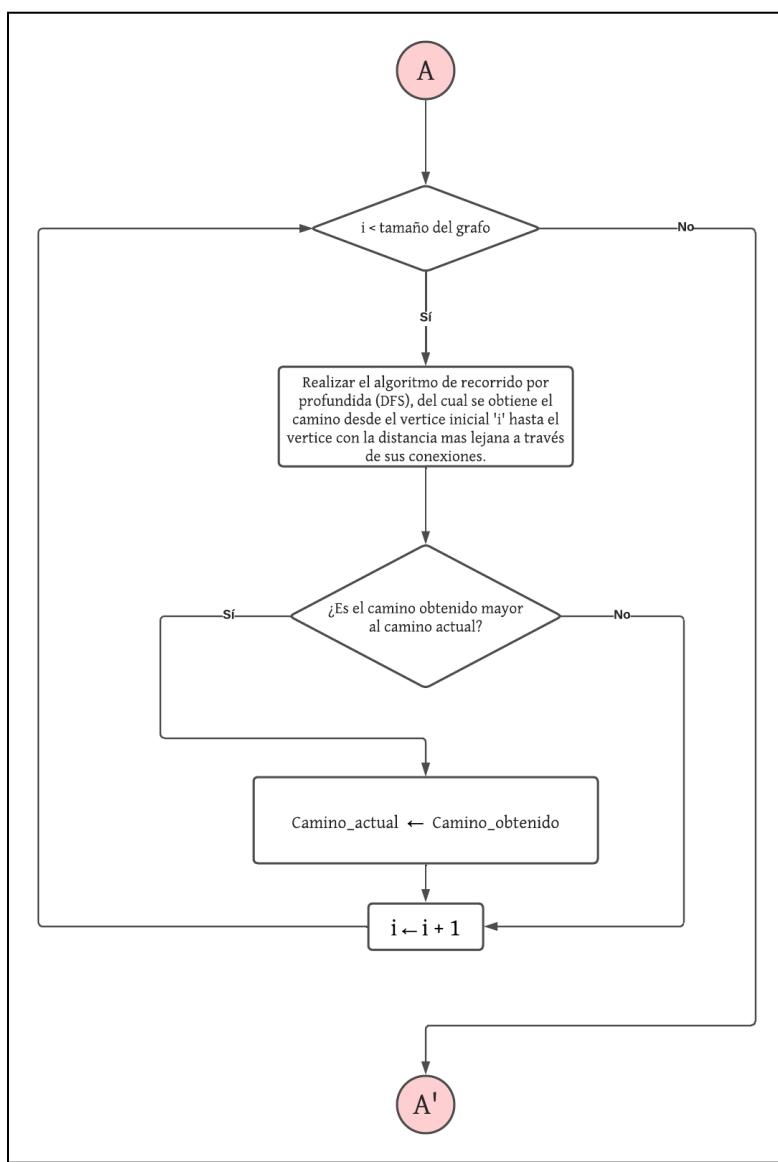


Fig 29. Diagrama de flujo del funcionamiento del ciclo de evaluación de todos las rutas dentro de ruta_mas_rapida

6. Plan de pruebas comando simular comandos

Para la realización del plan de pruebas se realizaron 5 casos de pruebas para probar el funcionamiento del comando simular comandos. A continuación se muestra cada caso con sus datos de entrada y su salida respectiva:

PROYECTO

Caso 1: Sin desplazamientos (no existen ni un solo movimiento dentro de la multi-lista de comandos)

El usuario ingresa la posición (15.0, 3589456.5).

En la lista de comandos no se encuentran valores de movimientos, debido que aun no se ha ingresado ninguno.

En este caso interesa evaluar la confiabilidad del comando para no ejecutarse en caso de no haber ingresado ningún movimiento hasta el momento, lógicamente. Por lo tanto, el resultado esperado debería ser un mensaje de error, del cual indique la falta de elementos que satisfagan el proceso normal del comando.

La posición luego de dichos comandos es incierta, ya que el proceso del comando no se ejecuta en su totalidad.

Fig 30. Evidencia del resultado de los datos de entrada.

```
$ simular_comandos 15.0 3589456.5
La información requerida no está almacenada en memoria.
```

Fig 30. Caso de prueba 1.

Caso 2: Se ingresan tipos de datos incorrectos.

El usuario ingresa la posición (23.5cm, 30km).

En la lista de comandos se encuentran los siguientes valores:

1. Desplazamiento (girar): **20.0 grados**.
2. Desplazamiento (avanzar): **6.0 m**.
3. Desplazamiento (girar): **35.0 grados**.
4. Desplazamiento (avanzar): **78.0 cm**.
5. Desplazamiento (girar): **70.0 grados**.
6. Desplazamiento (avanzar): **0.02 metros**.
7. Desplazamiento (girar): **50.0 grados**.
8. Desplazamiento (avanzar): **100.0 cm**.

PROYECTO

En este caso también se busca evaluar la confiabilidad del comando, a diferencia que en este caso se evalúa que el tipo de dato si sea el correspondiente que permita al comando ejecutar en su totalidad. Por lo tanto el valor esperado debería ser un mensaje de error que indique que el proceso no se está realizando de la forma adecuada.

La posición luego de dichos comandos es incierta, ya que el proceso del comando no se ejecuta en su totalidad.

Fig 31. Evidencia del resultado de los datos de entrada.

```
$ simular_comandos 23.5cm 30km
La información requerida no está almacenada en memoria.
```

Fig 31. Caso de prueba 2.

Caso 3: Cuadrado (donde todos los valores son enteramente positivos)

El usuario ingresa la posición (2.0, 4.0)

En la lista de comandos se encuentran los siguientes valores:

1. Desplazamiento (girar): **90.0 grados**.
2. Desplazamiento (avanzar): **10.0 metros**.
3. Desplazamiento (girar): **90.0 grados**.
4. Desplazamiento (avanzar): **10.0 metros**.
5. Desplazamiento (girar): **90.0 grados**.
6. Desplazamiento (avanzar): **10.0 metros**.
7. Desplazamiento (girar): **90.0 grados**.
8. Desplazamiento (avanzar): **10.0 metros**.

La posición esperada para este caso es la misma posición inicial o de origen, ya que al hacer la analogía de dibujar un cuadrado en el plano cartesiano guiado únicamente por la misma línea, dará como resultado la misma ubicación donde inició. Por lo tanto debería ser **(2.0, 4.0)**.

La posición luego de dichos comandos es: **(2.0, 4.0)**.

Fig 32. Evidencia del resultado de los datos de entrada.

PROYECTO

```
$ simular_comandos 2.0 4.0
La simulación de los comandos, a partir de la posición (2.00 , 4.00), deja al robot en la nueva posición (2.00, 4.00) .
```

Fig 32. Caso de prueba 3.

Caso 4: Rombo (Donde todos los valores son enteramente negativos)

El usuario ingresa la posición (-5.0 , -5.0).

En la lista de comandos se encuentran los siguientes valores:

1. Desplazamiento (girar): **-45.0 grados.**
2. Desplazamiento (avanzar): **-10.0 metros.**
3. Desplazamiento (girar): **-90.0 grados.**
4. Desplazamiento (avanzar): **-10.0 metros.**
5. Desplazamiento (girar): **-90.0 grados.**
6. Desplazamiento (avanzar): **-10.0 metros.**
7. Desplazamiento (girar): **-90.0 grados.**
8. Desplazamiento (avanzar): **-10.0 metros.**

En este caso seguimos con la misma analogía de volver al mismo punto a diferencia que en este caso se hace dibujando un rombo, donde la posición esperada para este caso es la misma posición inicial o de origen, ya que, al invertir totalmente el rombo y recorrerlo desde la cola hasta la cabeza se dará como resultado la misma ubicación donde inició. Por lo tanto debería ser **(-5.0, -5.0)**.

La posición luego de dichos comandos es: **(-5.0, -5.0)**.

Fig 33. Evidencia del resultado de los datos de entrada.

```
$ simular_comandos -5.0 -5.0
La simulación de los comandos, a partir de la posición (-5.00 , -5.00), deja al robot en la nueva posición (-5.00, -5.00) .
```

Fig 33. Caso de prueba 4.

PROYECTO

caso 5: Se ingresan datos combinados (positivos y negativos) desde una posición cero

El usuario ingresa la posición (0.0, 0.0).

En la lista de comandos se encuentran los siguientes valores:

1. Desplazamiento (girar): **45.0 grados**.
2. Desplazamiento (avanzar): **2.0 metros**.
3. Desplazamiento (girar): **-45.0 grados**.
4. Desplazamiento (avanzar): **4.0 metros**.
5. Desplazamiento (girar): **45.0 grados**.
6. Desplazamiento (avanzar): **2.0 metros**.
7. Desplazamiento (girar): **-45.0 grados**.
8. Desplazamiento (avanzar): **3.0 metros**.

Para este caso se debe hacer un seguimiento más teórico para concluir si la prueba evalúa de forma efectiva el correcto funcionamiento del comando, así como se puede ver en la figura 34:

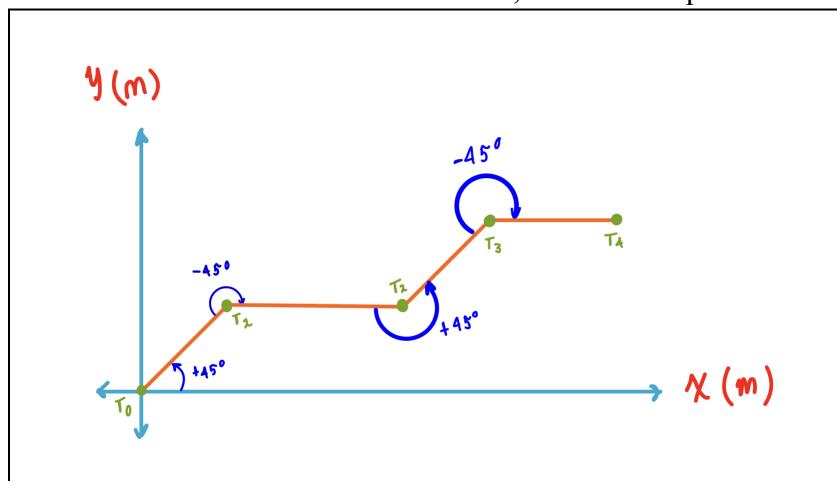


Fig 34. Caso de prueba 5.

Para este caso seguimos más el caso de forma matemática, donde teniendo en cuenta la fórmula para hallar la posición final:

$$X_{final} = distancia * \sin(grados)$$

$$Y_{final} = distancia * \cos(grados)$$

PROYECTO

Para este caso se tiene que:

$$X_{final} = 1.41 + 4.00 + 1.41 + 3$$

$$X_{final} = 9.83;$$

$$Y_{final} = 1.41 + 0 + 1.41 + 0$$

$$Y_{final} = 2.83;$$

Por lo tanto el valor esperado debería ser **(9.83, 2.83)**.

La posición luego de dichos comandos es: **(9.83, 2.83)**.

Fig 35. Evidencia del resultado de los datos de entrada.

```
$ simular_comandos 0.0 0.0
La simulación de los comandos, a partir de la posición (0.00 , 0.00), deja al robot en la nueva posición (9.83, 2.83) .
```

Fig 35. Caso de prueba 5.

Con base al análisis anterior, la siguiente tabla resume el plan de pruebas del comando simular comandos:

Tabla 14. Plan de pruebas: Comando simular comandos			
Descripción del caso	Valores de entrada	Resultado Esperado	Resultado Obtenido
1: Sin desplazamientos.	coordX = 15.0 , coordY = 3589456.5	Mensaje de error	La información requerida no está almacenada en memoria
2: Datos incorrectos.	coordX = 23.5cm , coordY = 30km	Mensaje de error	La información requerida no está almacenada en memoria
3: Cuadrado	coordX = 2.0 , coordY = 4.0	coordX = 2.0 , coordY = 4.0	coordX = 2.0 , coordY = 4.0

PROYECTO

4: Rombo	coordX = -5.0, coordY = -5.0	coordX = -5.0 , coordY = -5.0	coordX = -5.0 , coordY = -5.0
5: Combinados	coordX = 0.0 , coordY = 0.0	coordX = 9.83 , coordY = 2.83	coordX = 9.83 , coordY = 2.83

7. Plan de pruebas comando *en_cuadrante*

Para la realización del plan de pruebas se realizaron 7 casos de pruebas para comprobar el correcto funcionamiento del comando **en_cuadrante**. A continuación se muestra cada caso con sus datos de entrada y su salida respectiva.

Tabla 15. Elementos agregados y ubicados previamente.

tipo_comp	tamaño	unidad_med	coordX	coordY
roca	23	m	1	6
crater	13	m	4	4
duna	212	m	-56	-4
monticulo	1	m	6	4
crater	3	cm	4	3
duna	10	m	17	4
roca	4	cm	4	50
monticulo	3	cm	65	0
duna	7	cm	7	0

Caso 1: Se ingresan datos incompletos.

Descripción del caso: En este caso nos interesa evaluar que el comando funcione correctamente, cuando todos los datos solicitados fueron ingresados. Por lo tanto, el resultado esperado para cualquier dato donde falte al menos uno de los parámetros solicitados, deberá mostrar que la información del cuadrante no corresponde a los puntos esperados.

PROYECTO

Pre-input: Dentro de la estructura del árbol Quadtree de puntos se encuentran los valores de la *Tabla 3*.

Input: El usuario ingresa el punto inicial (**coordX1 = 4**, **coordY1 = 5**) y el punto final (**coordX2 = 3**, **coordY2 = NULL**).

Output: La localización de los elementos dentro del rango especificado no se han podido analizar debido a que faltan datos de entrada. Se puede evidenciar de forma más clara en *Fig 36*:

```
$ en_cuadrante 4 3 5
La información del elemento no corresponde a los puntos esperados (coordX1, coordX2, coordY1, coordY2).
```

Fig 36. Resultado con entradas incompletas.

Caso 2: Se ingresan datos con un formato incorrecto.

Descripción del caso: En este caso nos interesa evaluar que el comando funcione correctamente si los datos ingresados cumplen con el formato correspondiente. Por lo tanto, el resultado esperado para cualquier caso erróneo debería mostrar que la información del cuadrante no corresponde a los puntos esperados.

Pre-input: Dentro de la estructura del árbol Quadtree de puntos se encuentran los valores de la *Tabla 3*.

Input: El usuario ingresa el punto inicial (**coordX1 = 7m**, **coordY1 = 6cm**) y el punto final (**coordX2 = 3m**, **coordY2 = 4cm**).

Output: La localización de los elementos dentro del rango especificado no se han podido analizar debido a que los datos de entrada son incorrectos. Se puede evidenciar de forma más clara en *Fig 37*:

```
$ en_cuadrante 7m 3m 6cm 4cm
La información del cuadrante no corresponde a los puntos esperados (x_min, x_max, y_min, y_max).
```

Fig 37 Resultado con entradas con un formato incorrecto.

Caso 3: Se ingresan datos sin haber ubicado elementos.

Descripción del caso: En este caso nos interesa evaluar que el comando funcione correctamente si al momento de ejecutarlo se encuentra ingresado como mínimo un elemento dentro de la estructura del árbol, mediante el comando **ubicar_elementos**. Por

PROYECTO

lo tanto, el resultado esperado para cualquier caso donde el árbol aún no tenga ningún elemento debería mostrar que los elementos aún no han sido ubicados.

Pre-input: No hay datos ubicados al momento actual.

Input: El usuario ingresa el punto inicial (**coordX1** = 5.5, **coordY1** = 6.6) y el punto final (**coordX2** = 4.5, **coordY2** = 6.7).

Output: La localización de los elementos dentro del rango especificado no se han podido analizar debido a que aún no se han ubicado elementos. Se puede evidenciar de forma más clara en *Fig 38*:

```
$ en_cuadrante 5.5 4.5 6.6 6.7
```

Los elementos no han sido ubicados todavía.

Fig 38. Resultado sin elementos ubicados.

Caso 4: Verificación de coordenadas.

Descripción del caso: En este caso nos interesa evaluar que el comando funcione correctamente sólo si se cumple que **coordX1** es menor que **coordX2** y **coordY1** es menor que **coordY2**. Por lo tanto, el resultado esperado cuando la condición no se cumpla, debería mostrar error.

Pre-input: Dentro de la estructura del árbol Quadtree de puntos se encuentran los valores de la *Tabla 3*.

Input: El usuario ingresa el punto inicial (**coordX1** = 7, **coordY1** = 2) y el punto final (**coordX2** = 0, **coordY2** = 2).

Output: La localización de los elementos dentro del rango especificado no se han podido analizar debido a que la condición no ha sido cumplida. Se puede evidenciar de forma más clara en *Fig 39*:

```
$ en_cuadrante 7 0 2 2
```

La información del cuadrante no corresponde a los puntos esperados (x_min, x_max, y_min, y_max).

Fig 39. Resultado sin elementos ubicados.

Caso 5: Permite números reales.

Descripción del caso: En este caso nos interesa evaluar que el comando funcione correctamente para cualquier caso donde se ingresen números reales. Por lo tanto, el resultado esperado donde se ingresen cualquier tipo de números reales es su debido funcionamiento sin errores.

Pre-input: Dentro de la estructura del árbol Quadtree de puntos se encuentran los valores de la *Tabla 3*.

Input: El usuario ingresa el punto inicial (**coordX1 = -2, coordY1 = 1.25**) y el punto final (**coordX2 = 5.525, coordY2 = 5**).

Output: La localización de los elementos dentro del rango especificado han podido encontrar los elementos ‘crater’ y ‘crater’. Se puede evidenciar de forma más clara en *Fig 40*:

```
$ en_cuadrante -2 5.525 1.25 5
Los elementos del cuadrante solicitado son:
[Elemento: crater, tamaño: 13.00, unidad_medida: m, Coord X: 4.00, Coord Y: 4.00]
[Elemento: crater, tamaño: 3.00, unidad_medida: cm, Coord X: 4.00, Coord Y: 3.00]
```

Fig 40. Resultado sin elementos ubicados.

Caso 6: Ubica todos los elementos dentro del rango especificado.

Descripción del caso: En este caso nos interesa evaluar que el comando funcione correctamente, de forma que localicen todos los puntos almacenados dentro del árbol. Por lo tanto, el resultado esperado donde se ingrese correctamente la posición inicial y final resulte mostrando los puntos ubicados dentro dentro del cuadrante (para este caso todos).

Pre-input: Dentro de la estructura del árbol Quadtree de puntos se encuentran los valores de la *Tabla 3*.

Input: El usuario ingresa el punto inicial (**coordX1 = -60, coordY1 = -5**) y el punto final (**coordX2 = 70, coordY2 = 60**).

Output: La localización de los elementos dentro del rango especificado han podido encontrar todos los elementos que aparecen en la *Tabla 3*. Se puede evidenciar de forma más clara en *Fig 41*:



PROYECTO

```
$ en_cuadrante -60 70 -5 60

Los elementos del cuadrante solicitado son:

[Elemento: roca, tamano: 23.00, unidad_medida: m, Coord X: 1.00, Coord Y: 6.00]
[Elemento: roca, tamano: 4.00, unidad_medida: cm, Coord X: 4.00, Coord Y: 50.00]
[Elemento: crater, tamano: 13.00, unidad_medida: m, Coord X: 4.00, Coord Y: 4.00]
[Elemento: monticulo, tamano: 1.00, unidad_medida: m, Coord X: 6.00, Coord Y: 4.00]
[Elemento: duna, tamano: 10.00, unidad_medida: m, Coord X: 17.00, Coord Y: 4.00]
[Elemento: crater, tamano: 3.00, unidad_medida: cm, Coord X: 4.00, Coord Y: 3.00]
[Elemento: monticulo, tamano: 3.00, unidad_medida: cm, Coord X: 65.00, Coord Y: 0.00]
[Elemento: duna, tamano: 7.00, unidad_medida: cm, Coord X: 7.00, Coord Y: 0.00]
[Elemento: duna, tamano: 212.00, unidad_medida: m, Coord X: -56.00, Coord Y: -4.00]
```

Fig 41. Resultado sin elementos ubicados.

Caso 7: No encuentra elementos dentro del rango especificado.

Descripción del caso: En este caso nos interesa evaluar que el comando funcione correctamente, de forma que si dentro del rango no se encuentra ni un solo elemento, pues asimismo no localice ningún elemento dentro del cuadrante especificado. Por lo tanto, el resultado esperado donde se ingrese correctamente la posición inicial y final resulte muestre los puntos ubicados dentro del cuadrante (para este caso ninguno).

Pre-input: Dentro de la estructura del árbol Quadtree de puntos se encuentran los valores de la *Tabla 3*.

Input: El usuario ingresa el punto inicial (**coordX1 = 70, coordY1 = 60**) y el punto final (**coordX2 = 75, coordY2 = 65**).

Output: La localización de los elementos dentro del rango especificado no ha podido encontrar ningún elemento dentro del cuadrante especificado. Se puede evidenciar de forma más clara en *Fig 42*:

```
$ en_cuadrante 70 75 60 65

No existen elementos en el cuadrante especificado
```

Fig 42. Resultado sin elementos ubicados.

Tabla 16. Plan de pruebas: Comando en cuadrante

Descripción del caso	Valores de entrada	Resultado Esperado	Resultado Obtenido
----------------------	--------------------	--------------------	--------------------

PROYECTO

Se insertan datos incompletos	coordX1 = 4, coordY1 = 5 coordX2 = 4, coordY2 = NULL.	Mensaje de error	La información del cuadrante no corresponde a los puntos esperados (x_min, x_max, y_min, y_max).
Se ingresan datos con un formato incorrecto	coordX1 = 7m, coordY1 = 6cm, coordX2 = 3m, coordY2 = 4cm.	Mensaje de error	La información del cuadrante no corresponde a los puntos esperados (x_min, x_max, y_min, y_max).
Se ingresan datos sin haber ubicado elementos	coordX1 = 5.5, coordY1 = 6.6, coordX2 = 4.5, coordY2 = 6.7.	Mensaje de error	Los elementos no han sido ubicados todavía.
Verificación de coordenadas	coordX1 = 7, coordY1 = 2, coordX2 = 0, coordY2 = 2.	Mensaje de error	La información del cuadrante no corresponde a los puntos esperados (x_min, x_max, y_min, y_max).
Permite números reales	coordX1 = -2, coordY1 = 1.25, coordX2 = 5.525, coordY2 = 5.	Elementos localizados: 'crater' y 'crater'	Elemento: crater, tamaño: 13.00, unidad_medida: m, Coord X: 4.00, Coord Y: 4.00. Elemento: crater, tamaño: 3.00, unidad_medida: cm, Coord X: 4.00, Coord Y: 3.00.
Ubica todos los elementos dentro del rango especificado	coordX1 = -60, coordY1 = -5, coordX2 = 70,	Elementos localizados: TODOS	Elemento: roca, tamaño: 23.00,

PROYECTO

	coordY2 = 60.		<p>unidad_medida: m, Coord X: 1.00, Coord Y: 6.00.</p> <p>Elemento: roca, tamano: 4.00, unidad_medida: cm, Coord X: 4.00, Coord Y: 50.00.</p> <p>Elemento: crater, tamano: 13.00, unidad_medida: m, Coord X: 4.00, Coord Y: 4.00.</p> <p>Elemento: monticulo, tamano: 1.00, unidad_medida: m, Coord X: 6.00, Coord Y: 4.00.</p> <p>Elemento: duna, tamano: 10.00, unidad_medida: m, Coord X: 17.00, Coord Y: 4.00.</p> <p>Elemento: crater, tamano: 3.00, unidad_medida: cm, Coord X: 4.00, Coord Y: 3.00.</p> <p>Elemento: monticulo, tamano: 3.00, unidad_medida: cm, Coord X: 65.00, Coord Y: 0.00.</p>
--	----------------------	--	--

PROYECTO

			Elemento: duna, tamaño: 7.00, unidad_medida: cm, Coord X: 7.00, Coord Y: 0.00. Elemento: duna, tamaño: 212.00, unidad_medida: m, Coord X: -56.00, Coord Y: -4.00.
No encuentra elementos dentro del rango especificado	coordX1 = 70, coordY1 = 60, coordX2 = 75, coordY2 = 65.	Mensaje de error	No existen elementos en el cuadrante especificado

8. Plan de pruebas comando *ruta_mas_larga*

Para la realización del plan de pruebas acerca de la función '**ruta_mas_larga**' se realizaron 3 casos de pruebas para comprobar el correcto funcionamiento del comando. A continuación se muestra cada caso con sus datos de entrada y su salida respectiva.

Caso 1: No se encuentra un mapa creado.

Descripción del caso: En este caso nos interesa evaluar que el comando se ejecute solo si se ha ubicado cada punto de interés y ha sido correctamente conectado dentro del mapa, esto con respecto al coeficiente de conectividad especificado. Por lo tanto, el resultado esperado donde se ejecute el comando cuando el mapa no ha sido creado es que resulte mostrando un mensaje indicando se debe crear el mapa primero .

Pre-input: No se encuentran vértices dentro del grafo al momento actual.

Input: El usuario solamente ejecuta el comando sin argumentos.

Output: La determinación de la ruta más larga no ha podido ser procesada debido a que los elementos aún no han sido ubicados y conectados dentro del mapa. Se puede evidenciar de forma más clara en *Fig 43*:

```
$ ruta_mas_larga
```

El mapa no ha sido generado todavía.

Fig 43. Resultado sin mapa creado.

Caso 2: El formato del comando es incorrecto.

Descripción del caso: En este caso nos interesa evaluar que el comando funcione solo si se indica que se ejecute el comando sin ningún otro argumento. Por lo tanto, el resultado esperado donde se ejecute el comando con argumentos extras es un mensaje indicando que el formato es incorrecto primero .

Pre-input: No se encuentran vértices dentro del grafo al momento actual.

Input: El usuario solamente ejecuta el comando sin argumentos.

Output: La determinación de la ruta más larga no ha podido ser procesada debido a que el formato del comando ha sido ingresado incorrectamente 44:

```
$ ruta_mas_larga argumento_extra
```

Formato de comando erróneo.

Fig 44. Resultado sin mapa creado.

Caso 3: Los elementos han sido ubicados y correctamente conectados.

Descripción del caso: En este caso nos interesa evaluar que el comando funcione correctamente cuando los elementos procesados han sido ubicados y conectados correctamente dentro del mapa. Por lo tanto, el resultado esperado cuando se ejecute el comando es que resulte mostrando los puntos de interés más alejados, la longitud total de la ruta y la secuencia que se debe seguir para recorrerla.

PROYECTO

(Prueba 1)

```
$ crear_mapa 0.2
```

El mapa se ha generado exitosamente. Cada elemento tiene 2 vecinos.

Fig 45. Ajuste de coeficiente de conectividad #1.

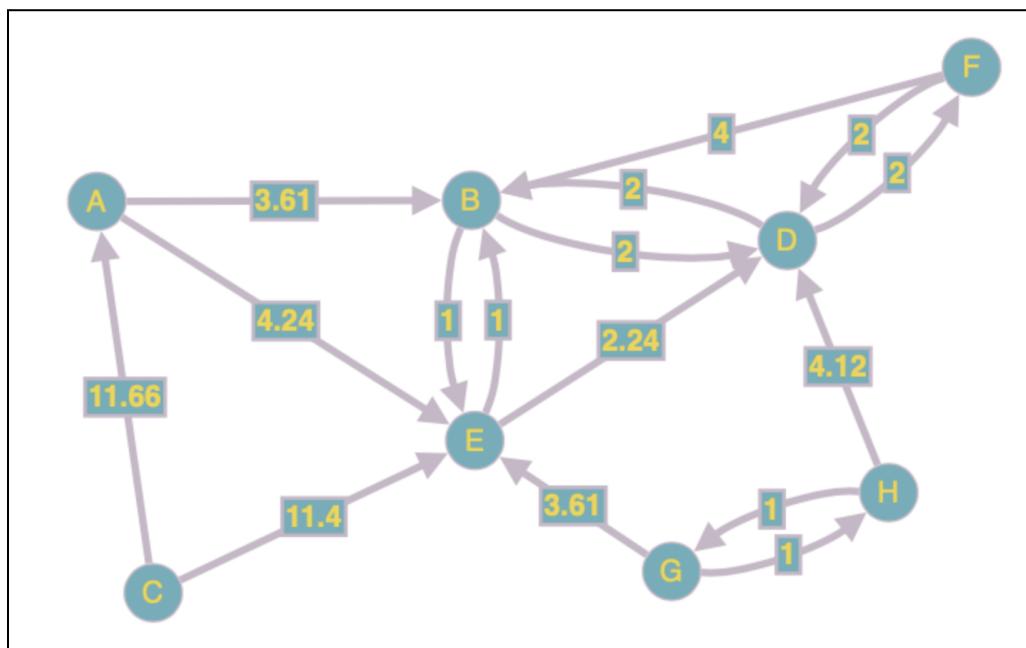


Fig 46. Plan de pruebas ruta más larga #1.

Pre-input #1: Dentro de la estructura del grafo ya se encuentran los vértices y aristas especificados por el usuario, en este caso nos referimos a los puntos de ejemplo que se pueden ver en la Fig 45 con un coeficiente de conectividad de 0.2, es decir 2 vecinos por vértice, como se puede ver en la Fig 46.

Para este caso en concreto con los puntos de los elementos de interés especificados por el usuario como se pueden ver en la Fig 45 , el resultado esperado nos debe mostrar que la ruta de los puntos más lejanos en cuanto a conexiones se refiere debería ser desde el elemento ‘C’ ubicado en el punto ($X = -5, Y = -4$), hasta el elemento ‘B’ ubicado en el punto ($X = 4, Y = 4$), con una longitud total de 24.14 y del cual el recorrido se puede observar en la Fig 47.

PROYECTO

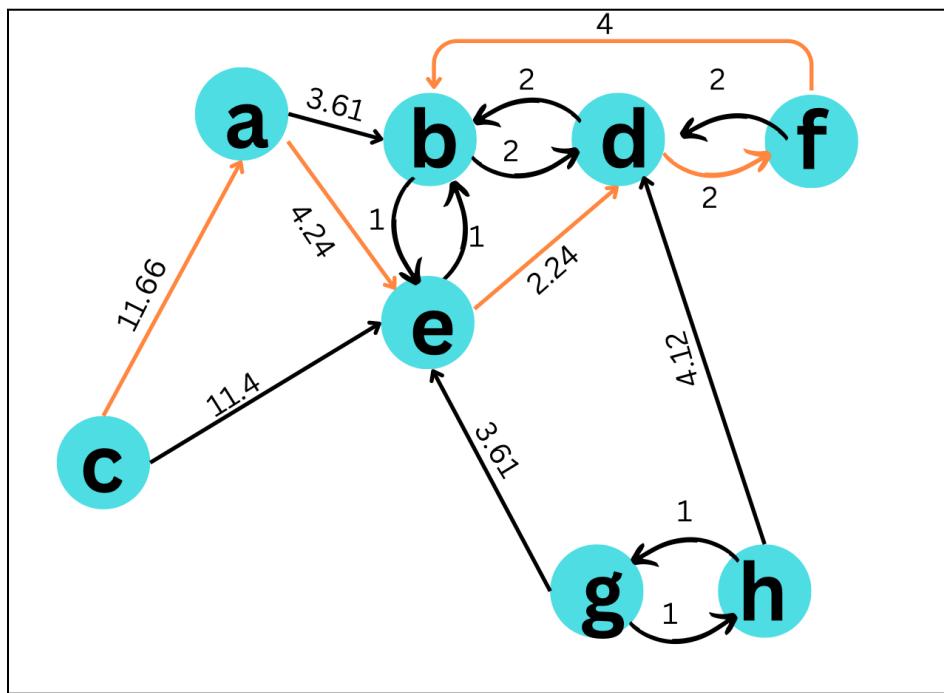


Fig 47. Recorrido de la ruta más larga.

Input: El usuario solamente ejecuta el comando sin argumentos.

Output: La ejecución del comando se realizó correctamente, dándonos como resultado los puntos más alejados ‘C’ ubicado en el punto ($X = -5, Y = -4$), y ‘B’ ubicado en el punto ($X = 4, Y = 4$), una longitud total entre los dos puntos de 24.14 y su respectivo recorrido. Se puede evidenciar de forma más clara en Fig 48:

```
$ ruta_mas_larga
Los puntos de interés más alejados entre sí son : (-5.00, -4.00) y (4.00, 4.00) .
La ruta que los conecta tiene una longitud total de : 24.14 .
Y pasa por los siguientes elementos : (-5.00, -4.00) -> (1.00, 6.00) -> (4.00, 3.00) -> (6.00, 4.00) -> (8.00, 4.00) -> (4.00, 4.00) .
```

Fig 48. Resultado correcto del comando #1.

PROYECTO

(Prueba 2)

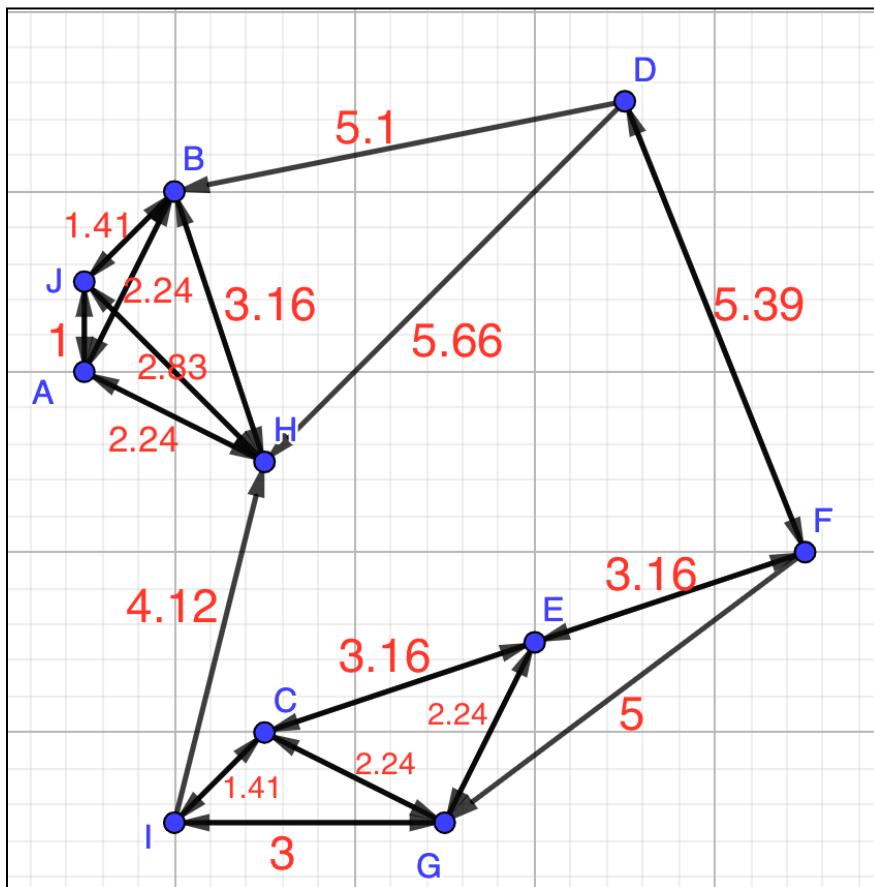


Fig 49. Plan de pruebas ruta más larga #2.

```
$ crear_mapa 0.3
```

El mapa se ha generado exitosamente. Cada elemento tiene 3 vecinos.

Fig 50. Ajuste de coeficiente de conectividad #2.

Pre-input #2: Dentro de la estructura del grafo ya se encuentran los vértices y aristas especificados por el usuario, en este caso nos referimos a los puntos de ejemplo que se pueden ver en la Fig 49 con un coeficiente de conectividad de 0.3, es decir 3 vecinos por vértice, como se puede ver en la Fig 50.

Para este caso en concreto con los puntos de los elementos de interés especificados por el usuario como se pueden ver en la Fig 49, el resultado esperado nos debe mostrar que la

PROYECTO

ruta de los puntos más lejanos en cuanto a conexiones se refiere debería ser desde el elemento 'I' ubicado en el punto ($X = 2, Y = 1$), hasta el elemento 'B' ubicado en el punto ($X = 1, Y = 7$), con una longitud total de 28.48 y del cual el recorrido se puede observar en la Fig 51:

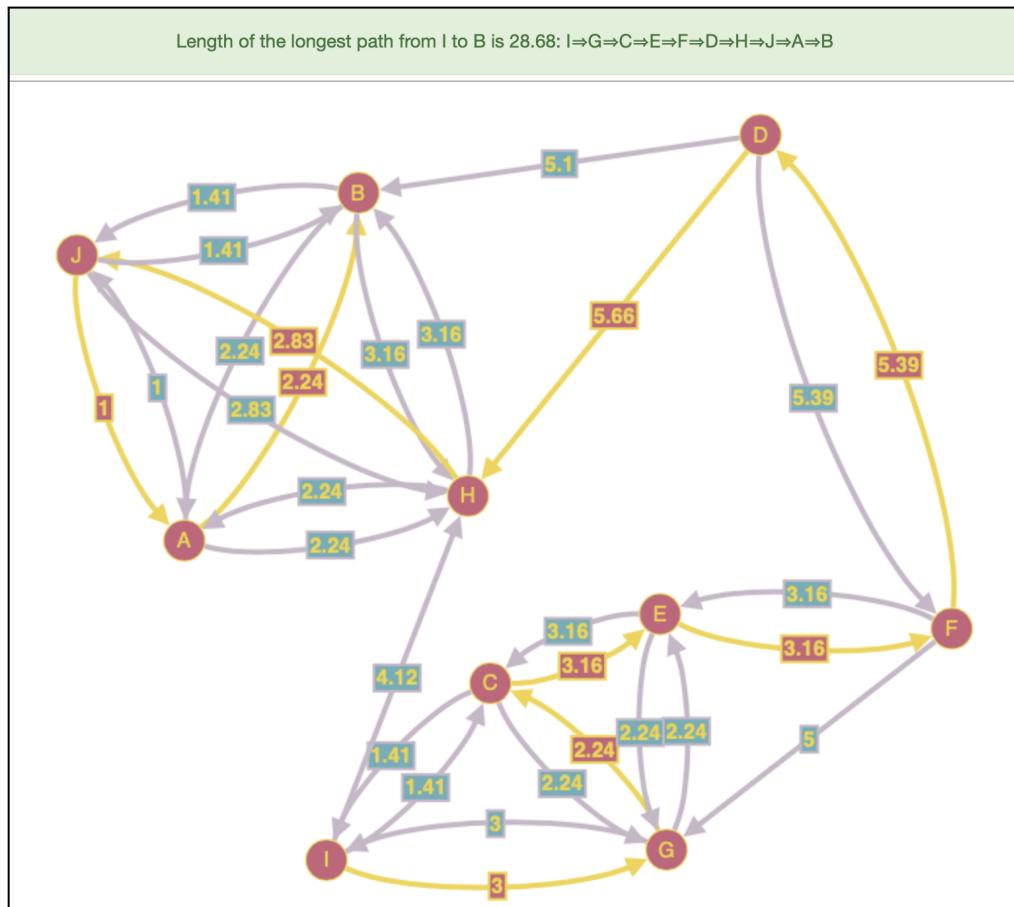


Fig 51. Recorrido de la ruta más larga #2.

Input: El usuario solamente ejecuta el comando sin argumentos.

Output: La ejecución del comando se realizó correctamente, dándonos como resultado los puntos más alejados 'I' ubicado en el punto ($X=2, Y=1$), y 'B' ubicado en el punto ($X=1, Y=7$), una longitud total entre los dos puntos de 24.14 y su respectivo recorrido. Se puede evidenciar de forma más clara en Fig 52:

```
$ ruta_mas_larga
Los puntos de interés más alejados entre sí son : (2.00, 1.00) y (1.00, 7.00) .
La ruta que los conecta tiene una longitud total de : 28.48 .
Y pasa por los siguientes elementos : (2.00, 1.00) -> (3.00, 5.00) -> (3.00, 2.00) -> (5.00, 1.00) -> (6.00, 3.00) -> (9.00, 4.00) -> (7.00, 9.00) -> (2.00, 8.00) -> (1.00, 6.00) -> (1.00, 7.00) .
```

PROYECTO

Fig 52. Resultado correcto del comando #2.

(Prueba 3)

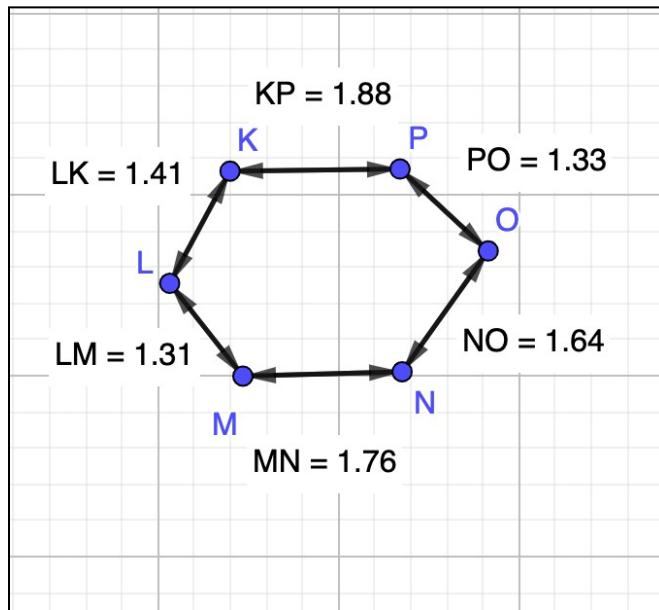


Fig 53. Plan de pruebas ruta más larga #3.

```
$ crear_mapa 0.34
```

El mapa se ha generado exitosamente. Cada elemento tiene 2 vecinos.

Fig 54. Ajuste de coeficiente de conectividad #3.

Pre-input #3: Dentro de la estructura del grafo ya se encuentran los vértices y aristas especificados por el usuario, en este caso nos referimos a los puntos de ejemplo que se pueden ver en la *Fig 53* con un coeficiente de conectividad de 0.2, es decir 2 vecinos por vértice, como se puede ver en la *Fig 54*.

Para este caso en concreto con los puntos de los elementos de interés especificados por el usuario como se pueden ver en la *Fig 54*, el resultado esperado nos debe mostrar que la ruta de los puntos más lejanos en cuanto a conexiones se refiere debería ser desde el elemento ‘L’ ubicado en el punto ($X = -13.87, Y = -7.2$), hasta el elemento ‘M’ ubicado en el punto ($X = -13.06, Y = 6$), con una longitud total de 24.14 y del cual el recorrido se puede observar en la *Fig 55*.

PROYECTO

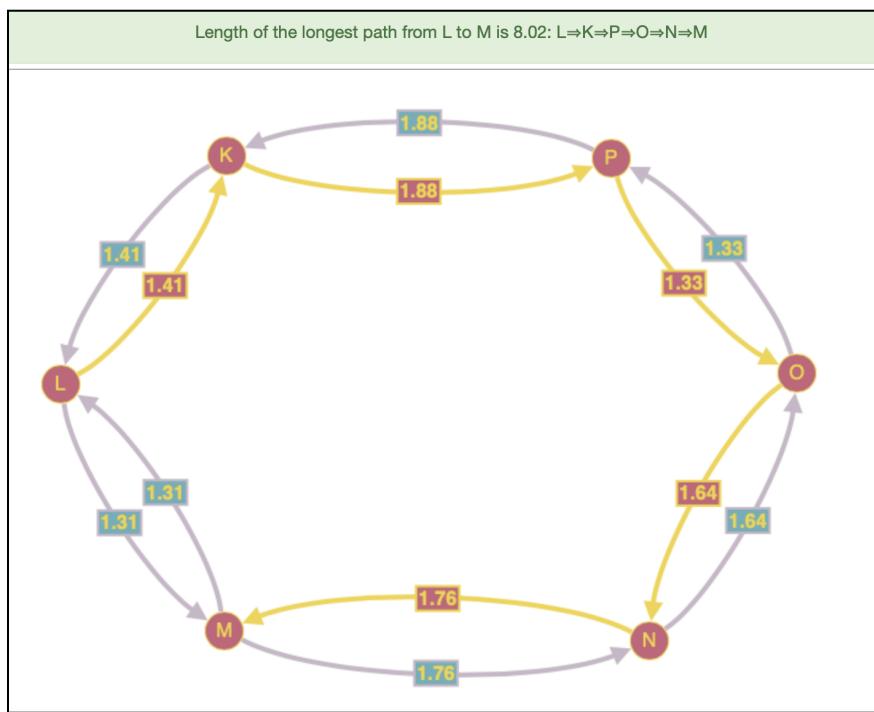


Fig 55. Recorrido de la ruta más larga #3.

Input: El usuario solamente ejecuta el comando sin argumentos.

Output: La ejecución del comando se realizó correctamente, dándonos como resultado los puntos más alejados ‘L’ ubicado en el punto ($X = -13.87, Y = -7.2$), y ‘M’ ubicado en el punto ($X = -13.06, Y = 6$), una longitud total entre los dos puntos de 8.03 y su respectivo recorrido. Se puede evidenciar de forma más clara en Fig 56:

```
$ ruta_mas_larga
Los puntos de interés más alejados entre sí son : (-13.87, 7.02) y (-13.06, 6.00) .
La ruta que los conecta tiene una longitud total de : 8.03 .
Y pasa por los siguientes elementos : (-13.87, 7.02) -> (-13.20, 8.26) -> (-11.32, 8.28) -> (-10.34, 7.38) -> (-11.30, 6.04) -> (-13.06, 6.00) .
```

Fig 56. Resultado correcto del comando #3.

Tabla 15. Plan de pruebas: Comando ruta mas larga

Descripción del caso	Valores de entrada	Resultado Esperado	Resultado Obtenido
1: No se encuentra un mapa creado.	Ninguno (el comando no tiene argumentos)	Mensaje de error	El mapa no ha sido generado todavía.

PROYECTO

2: El formato del comando es incorrecto.	Ninguno (el comando no tiene argumentos)	Mensaje de error	Formato de comandos erróneo.
3: Los elementos han sido ubicados y correctamente conectados (prueba #1).	Ninguno (el comando no tiene argumentos)	<p>Los puntos de interés más alejados entre sí son : (-5, -4) y (4, 4).</p> <p>La ruta que los conecta tiene una longitud total de : 24.14 m .</p> <p>Pasa por los siguientes elementos : (-5, -4) -> (1, 6) -> (4, 3) -> (6, 4) -> (8, 4) -> (4, 4) .</p>	<p>Los puntos de interés más alejados entre sí son : (-5.00, -4.00) y (4.00, 4.00) .</p> <p>La ruta que los conecta tiene una longitud total de : 24.14 m .</p> <p>Pasa por los siguientes elementos : (-5.00, -4.00) -> (1.00, 6.00) -> (4.00, 3.00) -> (6.00, 4.00) -> (8.00, 4.00) -> (4.00, 4.00) .</p>
3: Los elementos han sido ubicados y correctamente conectados (prueba #2).	Ninguno (el comando no tiene argumentos)	<p>Los puntos de interés más alejados entre sí son : (2, 1) y (1, 7) .</p> <p>La ruta que los conecta tiene una longitud total de : 28.48 m .</p> <p>Y pasa por los siguientes elementos : (2, 1) -> (3, 5) -> (3, 2) -> (5, 1) -> (6, 3) -> (9, 4) -> (7, 9) -> (2, 8) -> (1, 6) -> (1, 7) .</p>	<p>Los puntos de interés más alejados entre sí son : (2.00, 1.00) y (1.00, 7.00) .</p> <p>La ruta que los conecta tiene una longitud total de : 28.48 m .</p> <p>Y pasa por los siguientes elementos : (2.00, 1.00) -> (3.00, 5.00) -> (3.00, 2.00) -> (5.00, 1.00) -> (6.00, 3.00) -> (9.00, 4.00) -> (7.00, 9.00) -> (2.00, 8.00) -></p>

PROYECTO

			(1.00, 6.00) -> (1.00, 7.00) .
3: Los elementos han sido ubicados y correctamente conectados (prueba #3).	Ninguno (el comando no tiene argumentos)	<p>Los puntos de interés más alejados entre sí son : (-13.87, 7.02) y (-13.06, 6.00) .</p> <p>La ruta que los conecta tiene una longitud total de : 8.03 m.</p> <p>Y pasa por los siguientes elementos :</p> <ul style="list-style-type: none"> (-13.87, 7.02) -> (-13.20, 8.26) -> (-11.32, 8.28) -> (-10.34, 7.38) -> (-11.30, 6.04) -> (-13.06, 6.00) . 	<p>Los puntos de interés más alejados entre sí son : (-13.87, 7.02) y (-13.06, 6.00) .</p> <p>La ruta que los conecta tiene una longitud total de : 8.03 m.</p> <p>Y pasa por los siguientes elementos :</p> <ul style="list-style-type: none"> (-13.87, 7.02) -> (-13.20, 8.26) -> (-11.32, 8.28) -> (-10.34, 7.38) -> (-11.30, 6.04) -> (-13.06, 6.00) .