Pontificia Universidad Javeriana

Departamento de Ingeniería de Sistemas

Estructura de Datos

Entrega Final - Proyecto Estructura de Datos

Nombre de integrantes:

Tomás Figueroa Sierra

Santiago Camilo Rey Benavides

Juan Manuel Sánchez Bermúdez

Profesora: Andrea Rueda

Mayo 31, 2023

Bogotá, Colombia

Contenido

1. Corrección entrega 1:

1.1. Documento diseño:

- Los contenedores deben indicar exactamente el tipo de dato que están almacenando.

  Se cambiaron los tipos de datos en los contenedores del TAD Curiosity de modo que coincidieran exactamente con los tipos Análisis, Movimiento y Puntointeres.

- El diagrama de relación entre TADs no es un diagrama de clases, no debería existir diagrama de clases todavía porque en teoría aún no se ha implementado nada.

  Ahora se incluye el diagrama de relación entre TADs.

- No se incluyen esquemáticos (diagramas de flujo, de actividad, de secuencia) para cada uno de los comandos del sistema.

  Se incluyeron los diagramas de flujo para los comandos del sistema en este documento.

- No se incluye el análisis de entradas, salidas y condiciones para cada uno de los comandos del sistema.

  El análisis de entradas salidas y condiciones para los comandos se incluye en este documento.

1.2. Plan de pruebas:

- Es importante complementar el plan de pruebas con pantallazos que evidencien la ejecución.

  Se realizaron nuevos planes de pruebas con las especificaciones del enunciado 2 y se incluyeron los pantallazos de la ejecución.

1.3. Código fuente:

- Si los TADs no incluyen o están definidos con plantillas, no se deben implementar en archivos .hxx, sino en archivos .cxx o .cpp.

Los archivos de implementación se cambiaron a archivos cpp y cxx.

- En general, casi todos los comandos muestran información que no es necesaria en su ejecución. Lo único que deberían imprimir en pantalla los comandos son los mensajes indicados en el enunciado, de acuerdo a la situación particular.

  Se eliminaron las salidas en pantalla con información innecesaria para los comandos de carga y de simulación, de modo que solo se muestra lo indicado. En el caso de los comandos de carga de comandos y elementos se muestra cuantos comandos y puntos fueron cargados del archivo de carga.

- No es necesario agregar 0 o 1 al inicio de cada línea del archivo de comandos para diferenciarlos entre movimiento y análisis. En teoría los tipos de movimiento y los tipos de análisis ya están bien definidos y no son tantos como para poder validarlos.

  Se utilizaron las categorías de movimientos y análisis para clasificar los comandos y así evitar el uso de 0s y 1s en el archivo de comandos. A pesar de esto en el curiosity se sigue guardando un contenedor con el orden de los comandos para escribir el archivo generado por el comando guardar.

1.3.1. Comando Ayuda:

- La ayuda para cada comando no está disponible. Yo debería poder teclear ayuda guardar, y me debería salir en pantalla la información de cómo llamar al comando guardar y qué es lo que hace el comando.

  Se modificó el comando ayuda y su función para que de información específica del comando tecleado como segundo argumento.

1.3.2. Comando cargar_comandos:

- Si hay comandos que no son válidos, esa información debería mostrarse aquí al momento de cargar los comandos desde el archivo, y no al momento de la simulación.

  Ahora la función del comando cargar_comandos muestra en pantalla la línea del archivo que no contiene el formato de los comandos y posteriormente muestra cuantos comandos se agregaron de las líneas del archivo. Adicional a lo anterior, al momento de agregar el comandos a las listas de movimiento y análisis puede que los datos no correspondan al tipo de comando, si esto sucede se lee en pantalla “La información del análisis no corresponde a los datos esperados” , no se agrega ningún comando y se retorna false de la función agregar.

1.3.3. Comando cargar_elementos:

- No parece estar funcionando el comando, genera violación de segmento cuando lo intenté probar con un archivo propio de prueba.

  Es muy probable que haya pasado debido a los comentarios del archivo, pues no se estaban aceptando. Se cambió la función para agregar puntos de interés y de esta manera acepta archivos con comentarios con la forma del ejemplo propuesto en el enunciado 2.

1.3.4. Comando agregar_analisis:

- No me permite agregar un análisis sin comentario, cuando se supone que el comentario es opcional.

  La corrección del comentario anterior soluciona el problema ocasionado por el comentario obligatorio, esto siempre y cuando este tecleado entre comillas simples.

2. Corrección entrega 2:

2.1. Código Fuente:

2.1.1. Comando guardar:

- Revisar en el comando de guardar por qué no quedan guardados los comentarios de los análisis.

  Sucede que al agregar los análisis se guardaban los comentarios usando el carácter (‘) utilizado en el ejemplo del enunciado del proyecto. Se modificó la función agregar_analisis de modo que se guardaran los comentarios con comillas simples también.

2.2. Documento de diseño:

- En el diagrama de relación entre TADs no es necesario incluir los parámetros de las operaciones.

  Se quitaron los parámetros de las operaciones en el diagrama de relación.

3. Diagrama de Relación:

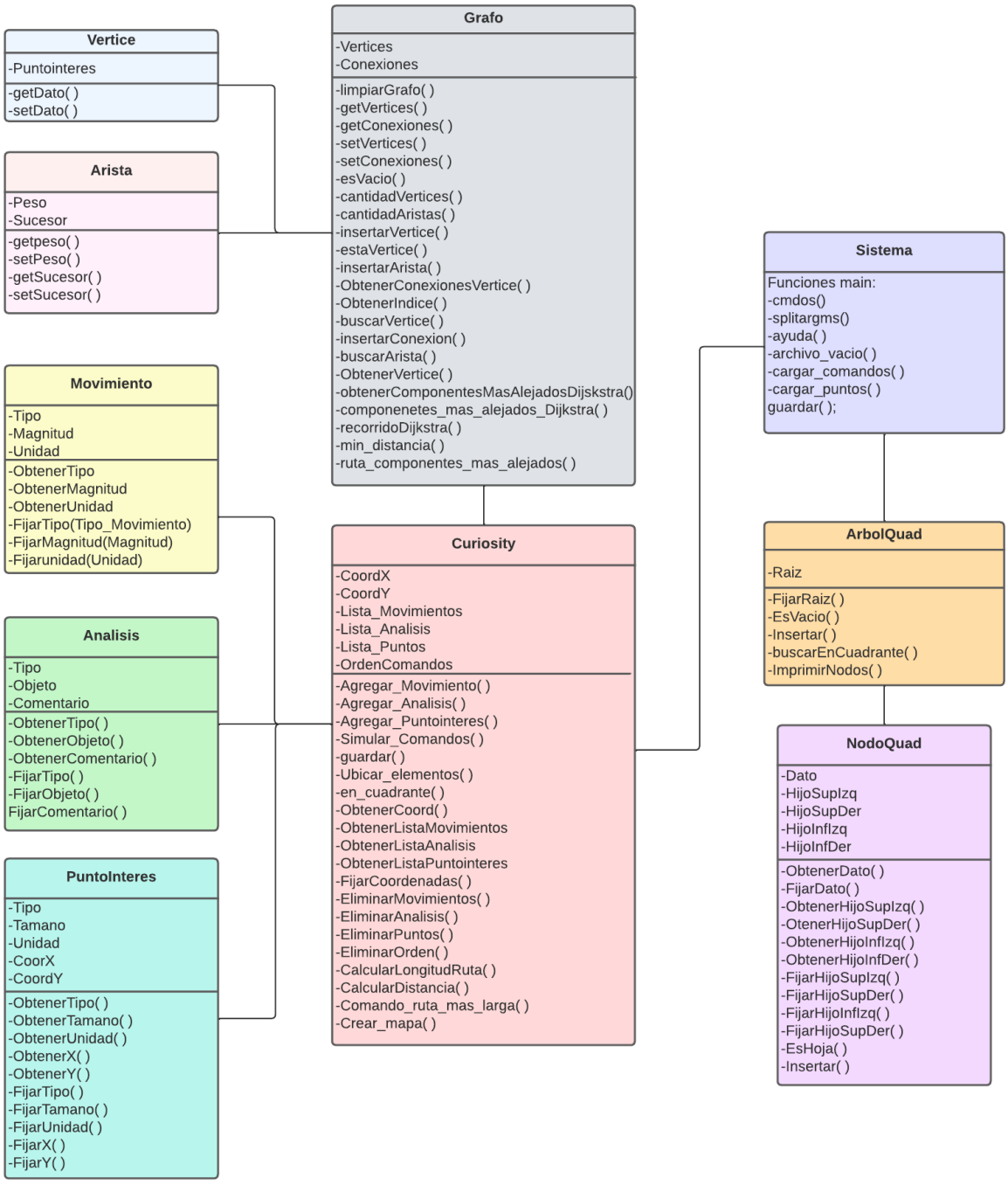


Figura 1. Diagrama de relación

4. TADs

4.1. TAD Análisis

Atributos:

- Tipo de análisis, cadena de caracteres, indica el tipo de análisis a realizar.
- Objeto, cadena de caracteres, se trata del objeto sobre el cual se hace el análisis.
- Comentario, cadena de caracteres, descripción e información adicional sobre el análisis.

Operaciones:

- Obtener_tipo(), retorna el tipo de análisis.
- Obtener_objeto(), retorna el objeto del análisis.
- Obtener_comentario(), retorna el comentario del análisis.
- Fijar_tipo (n_tipo), fija un nuevo tipo de análisis como n_tipo.
- Fijar_objeto(n_objeto), fija un nuevo objeto para el análisis como n_objeto.
- Fijar_objeto(n_comentario), fija un nuevo comentario n_comentario.

4.2. TAD Movimiento

Atributos:

- Tipo de movimiento, cadena de caracteres, indica si se trata de un giro o el avance del vehículo (“avanzar” o “girar”).
- Magnitud, número real, es la magnitud (distancia o grados) del movimiento realizar.
- Unidad de medida, cadena de caracteres, establece la unidad de medida de la magnitud (cm o m).

Operaciones:

- Obtener_tipo(), retorna el tipo de movimiento.
- Obtener_magnitud (), retorna la magnitud del movimiento .
- Obtener_unidad (), retorna la unidad de medida.
- Fijar_tipo (n_tipo), fija n_tipo como nuevo tipo de movimiento.
- Fijar_objeto(n_objeto), fija n_objeto como magnitud.
- Fijar_unidad (n_unidad), fija n_unidad como unidad de medida.

4.3. TAD Puntointeres

Atributos:

- Tipo de elemento, carácter, carácter que indica de que se trata el punto de interés.

- Unidad de medida, cadena de caracteres, establece la unidad con la que se midió el objeto.
- Posición X, numero real, representa la magnitud de la coordenada del eje x.
- Posición Y, número real, representa la magnitud de la coordenada del eje y.

Operaciones:

- Obtener_tipo(), retorna el tipo de terreno.
- Obtener_unidad (), retorna la unidad de medida.
- Obtener_coordenadas (), retorna la posición del punto de interés.
- Fijar_tipo (n_tipo), fija n_tipo como tipo de terreno.
- Fijar_unidad (n_unidad), fija n_unidad como unidad de medida.
- Fijar_X (M), Establece M como coordenada en el eje X.
- Fijar_Y (N), Establece N como coordenada en el eje N.

4.4. TAD Curiosity

Atributos:

- Movimientos, lista de Movimiento, incluye los movimientos que debe realizar el vehículo manteniendo el orden en que fueron ingresados.
- Análisis, lista de Analisis, incluye los análisis con sus respectivos tipos que debe realizar el curiosity
- Puntos_interes, lista de Puntointeres, contiene los puntos de interés para el curiosity.
- Lista orden, lista de enteros, lista de 1s y 0s que indican el orden en que deben ser ejecutados los comandos de movimiento y análisis.
- Posición X, numero real, representa la magnitud de la coordenada del eje x.
- Posición Y, número real, representa la magnitud de la coordenada del eje y

Operaciones:

- Curiosity(), constructor del objeto curiosity, inicializa las coordenadas en el origen
- Simular_coordenadas(), retorna el resultado de la posición del vehículo como simulación de los comandos de movimiento cargados y la posición indicada.
- Guardar(tipo_archivo, nombre_archivo), recibe el tipo de archivo y guarda los comandos del curiosity o los puntos de interés.
- ObtenerListaMovimientos(), retorna la lista de los movimiento cargados o agregados al curiosity;
- ObtenerListaAnalisis(), retorna la lista de tipo análisis con los objetos de este tipo cargados.
- ObtenerListaPuntos(), retorna la lista con los puntos de interés agregados o cargados al curiosity.
- ObtenerOrdenComandos(), retorna la lista de1s y 0s con el orden de los comandos.
- Agregar_Movimiento(cadena de caracteres), recibe una cadena de caracteres con los elementos del comando de movimiento, los tokeniza, crea un objeto de tipo Movimiento al que agrega estos elementos. Añade este elemento a la lista de movimientos y agrega un 0 a la lista de orden de comandos.
- Agregar_Analisis(cadena de caracteres), recibe una cadena de caracteres con los elementos del comando de análisis, los tokeniza, crea un objeto de tipo

Análisis al que agrega estos elementos. Añade este elemento a la lista de movimientos y agrega un 1 a la lista de orden de comandos.
- Agregar_Puntointeres(cadena de caracteres), recibe una cadena de caracteres con los elementos del puntos de interes, los tokeniza, crea un objeto de tipo Puntointeres al que agrega estos elementos. Añade este elemento a la lista de Puntointeres .
- fijarCoordenadas(numero real, numero real), fija las coordenadas actuales del curiosity.
- eliminarMovimientos(), vacía la lista de movimientos.
- eliminarAnalisis(); vacía la lista de Analisis.
- eliminarPuntos(), vacia la lista de Puntos.
- simular_comandos(coordX, coordY), toma las coordenadas ingresadas y ejecuta los comandos cargados para mostrar en qué posición quedaría el vehículo.
- ubicar_elementos(), ubica los elementos en la estructura de datos Quadtree
- en_cuadrante(coordX1 coordX2 coordY1 coordY2), retorna una lista que esta dentro del cuadrante

4.5. TAD ArbolQuad

Atributos:

- raiz, tipo de dato nodo, crea la raíz del árbol.

Operaciones:

- ArbolQuad(), método constructor.
- ArbolQuad(val), crea un objeto de la clase ArbolQuad inicializando la data con el
- elemento pasado como parámetro.
- ArbolQuad(), método destructor.
- datoRaiz(), obtiene el dato que se encuentra en la raíz del árbol.
- obtenerRaiz(), retorna el dato que se encuentra en la raíz del árbol.
- fijarRaiz(n_raiz),actualiza la raíz del árbol.
- esVacio(), evalúa si el árbol se encuentra vacío.
- buscarEnCuadrante(nodo_actual, x1, y1, x2, y2, vector elementos_encontrados), busca por todo el arbol si encuentra algun punto de interés dentro de los límites(x1, x2, y1, y2) y lo guarda en el vector elementos_encontrados
- insertar(punto), inserta un nuevo punto de interes dentro del arbol, si esta vacia crea un nuevo y sino llama a la funcion insertar de NodoQuad
- imprimirNodos(nodo_actual), imprime todos los valores de X y Y de cada uno de los nodos del arbol

4.6. TAD NodoQuad

Atributos:

- dato, dato de tipo entero, representa el dato que almacena el nodo.
- hijoSupIzq, apuntador a NodoQuad, representa el hijo superior izquierdo del nodo.

- hijoSupDer, apuntador a NodoQuad, representa el hijo superior derecho del nodo.
- hijoInfIzq, apuntador a NodoQuad, representa el hijo inferior izquierdo del nodo.
- hijoInfDer, apuntador a NodoQuad, representa el hijo inferior derecho del nodo.

Operaciones:

- NodoQuad(), método constructor.
- NodoQuad(val), crea un objeto de la clase NodoQuad inicializando la data con el
- elemento pasado como parámetro.
- NodoQuad(), método destructor.
- obtenerDato(), retorna el dato almacenado por el nodo.
- fijarDato(val), actualiza el dato almacenado por el nodo.
- obtenerHijoSupIzq(), retorna el hijo superior izquierdo del nodo.
- obtenerHijoSupDer(), retorna el hijo superior derecho del nodo.
- obtenerHijoInfIzq(), retorna el hijo inferior izquierdo del nodo.
- obtenerHijoInfDer(), retorna el hijo inferior derecho del nodo.
- fijarHijoSupIzq(sizq), actualiza el hijo superior izquierdo del nodo con la data pasada
- como parámetro.
- fijarHijoSupDer(sder),actualiza el hijo superior derecho del nodo con la data pasada
- como parámetro.
- fijarHijoInfIzq(iizq), actualiza el hijo inferior izquierdo del nodo con la data pasada
- como parámetro.
- fijarHijoInfDer(ider), actualiza el hijo inferior derecho del nodo con la data pasada
- como parámetro.
- esHoja(), identifica si el nodo es una hoja.
- insertar(punto), inserta un nuevo nodo dentro del arbol de acuerdo a los valores valores de X y Y del nuevo nodo

4.7. TAD Arista

Atributos:

- peso, tipo de dato flotante, indica el peso que tiene la conexión.
- sucesor, tipo de dato vértice, indica el vértice siguiente.

Operaciones:

- Arista(), método constructor.
- Arista(), método constructor que crea una arista dado el peso y el vértice sucesor.
- getPeso(), obtiene el peso de la arista.
- getSucesor(), obtiene el vértice sucesor.
- setPeso(), fija el peso de la arista.
- setSucesor(), fija el vértice sucesor.

4.8. TAD vértice

Atributos:

- dato, tipo de dato PuntoInteres, indica un punto de interés.

Operaciones:

- Vertice(), método constructor.
- Vertice(), método constructor que crea un vértice dado un dato.
- getDato(), obtiene el dato que se encuentra asociado al vértice.
- setDato(), fija el dato en un vértice dado el dato.

4.9. TAD Grafo

Atributos:

- conexiones: Es una lista de listas de Arista. Cada lista interna representa las conexiones de un vértice en el grafo.
- vertices: Es un vector de Vertice. Cada elemento del vector representa un vértice en el grafo.

Operaciones:

- Grafo(), método constructor del grafo,
- getVertices(), retorna el dato que se encuentra en el vértice.
- getConexiones(), retorna el dato con el peso de la arista.
- setVertices(), fija el dato que se pasa por parámetro en el vértice.
- setConexiones (), fija una nueva arista con el peso y un apuntador del vértice sucesor.
- esVacio(), indica si el grafo está vacío.
- cantidadVertices(), retorna la cantidad de vértices que se encuentran en el grafo.
- cantidadAristas(), retorna la cantidad de vértices que se encuentran en el grafo.
- insertarVertice(), inserta un nuevo espacio para un nuevo vértice en el grafo.
- insertarVertice(), inserta un dato en el nuevo vértice creado.
- estaVertice(), válida si existe el vértice dado el vértice en el grafo.
- estaVertice(), válida si existe el vértice dado el dato del vértice en el grafo.
- insertarArista(), inserta una nueva arista dados los datos de los vértices y el peso.
- obtenerConexionesVertice(), retorna el apuntador de la lista de aristas(conexiones) de la multilista de conexiones dado el índice.
- obtenerIndice(), retorna la posición del vértice en la lista de vértices.
- buscarVertice(), retorna el apuntador del vértice de la lista de vértices dado el parámetro.
- insertarConexion(), inserta la nueva arista (conexión) en la lista pasada como parámetro.
- verticeVisitado(), indica si el vértice ya fue visitado.
- ObtenerVertice(), obtiene el vértice visitado.
- obtenerDistanciasMasLargas(), obtiene la distancia más larga presente en el grafo.
- obtenerComponentesMasAlejados(), obtiene los componentes más alejados.
- calcularDistancia(), calcula la distancia dado dos puntos de interés.
- ruta_mas_larga(), retorna un vector con la ruta más larga.
- dfs_ruta_mas_larga(), indica la ruta más larga dada un punto de interés.
- DFS(),
- calcularDistanciaEntrePuntos(), calcula distancia dados dos puntos de interés.
- obtenerComponentesMasAlejadosDijskstra(), se obtiene el componente más alejado dado un vector de distancias.

- recorridoDijkstra(), recorrido Dijkstra en el grafo.
- min_distancia(), retorna la menor distancia dado un conjunto de vértices y de conexiones.
- resultado_ruta_componenetes_mas_alejados_Dijkstra(), retorna un vector con los componentes más alejados usando el recorrido de Dijktra.
- obtener_ruta_mas_larga_Dijkstra(), retorna la ruta más larga de los puntos de interés usando el recorrido de Dijkstra.

5. Comandos

5.1. Comando: ayuda

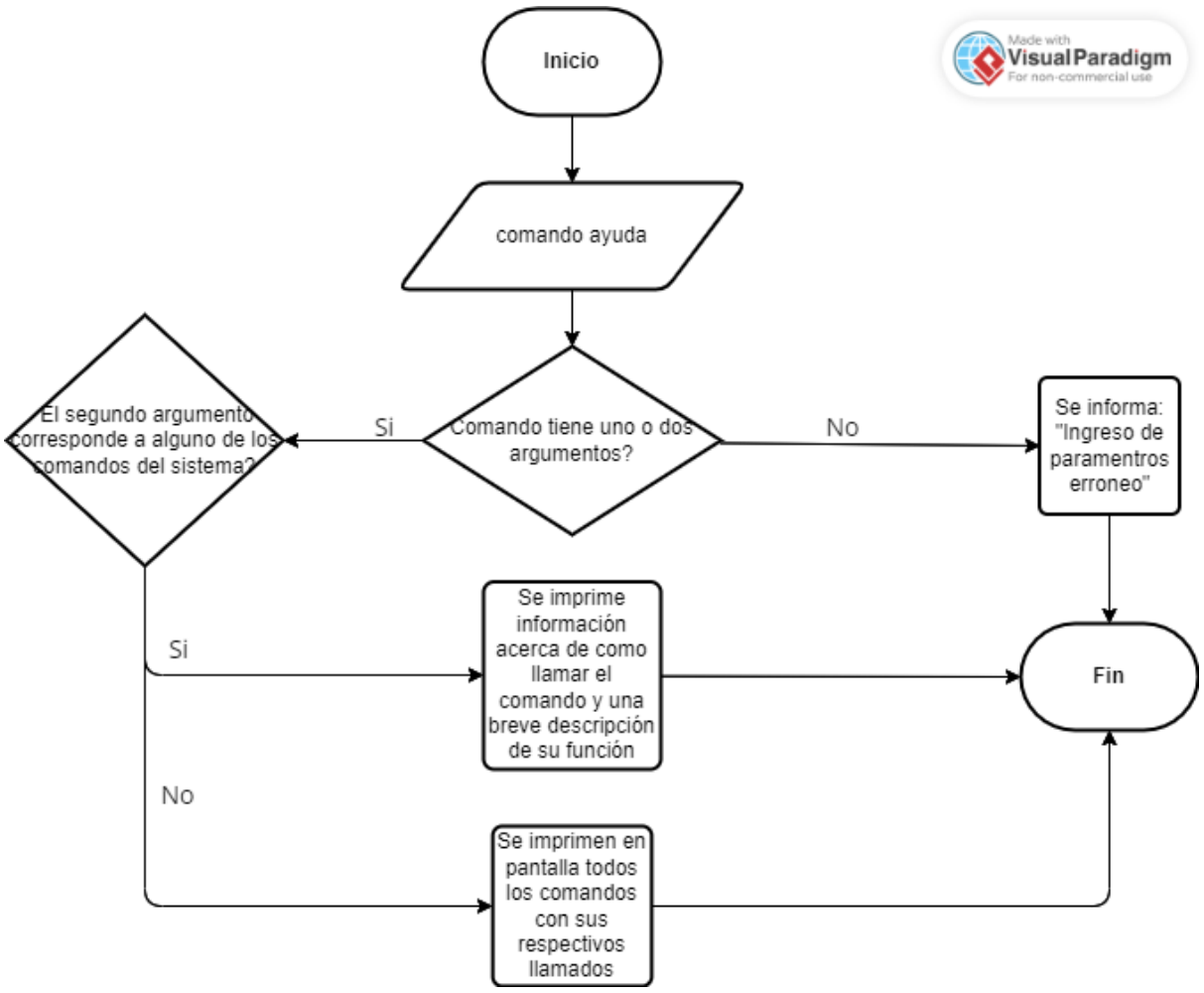


Figura 2. Diagrama de flujo comando ayuda

***Descripción*:** muestra por consola una descripción del comando deseado.

***Entrada*:** Comando “ayuda”.

Salida: Se tienen tres posibles salidas.

- Si los argumentos no se encuentran completos se le notifica al usuario.
- Si uno de los argumentos es erróneo se le notifica al usuario del error.
- Si la cantidad de argumentos y del comando se encuentran se le notifica el éxito mostrando el comando y una pequeña descripción.

5.2. Comando: cargar_comandos.

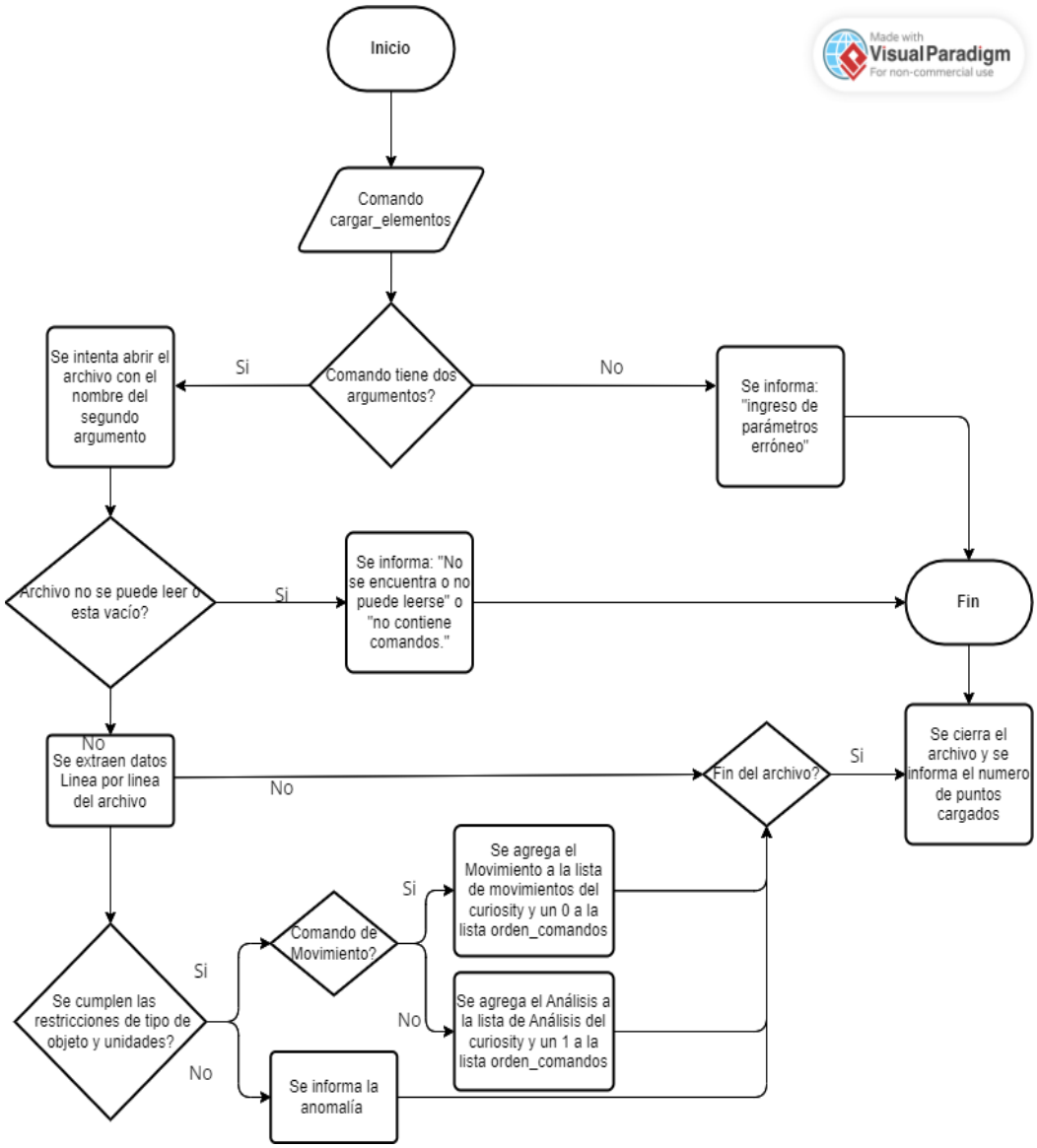


Figura 3. Diagrama de flujo comando cargar_comandos

***Descripción*:** Carga en memoria los comandos de análisis y movimiento del respectivo archivo.

***Entrada*:** Se hace la lectura del respectivo archivo y se extrae los comandos tanto de análisis como de movimiento.

Salida: Se tienen dos posibles salidas.

- Si el archivo no existe o se encuentra vacío devuelve un mensaje de aviso al usuario.
- Si el archivo existe y cumple con los parámetros se carga los comandos de análisis y de movimiento avisando de su éxito al usuario.

-

5.3. Comando: cargar_elementos.

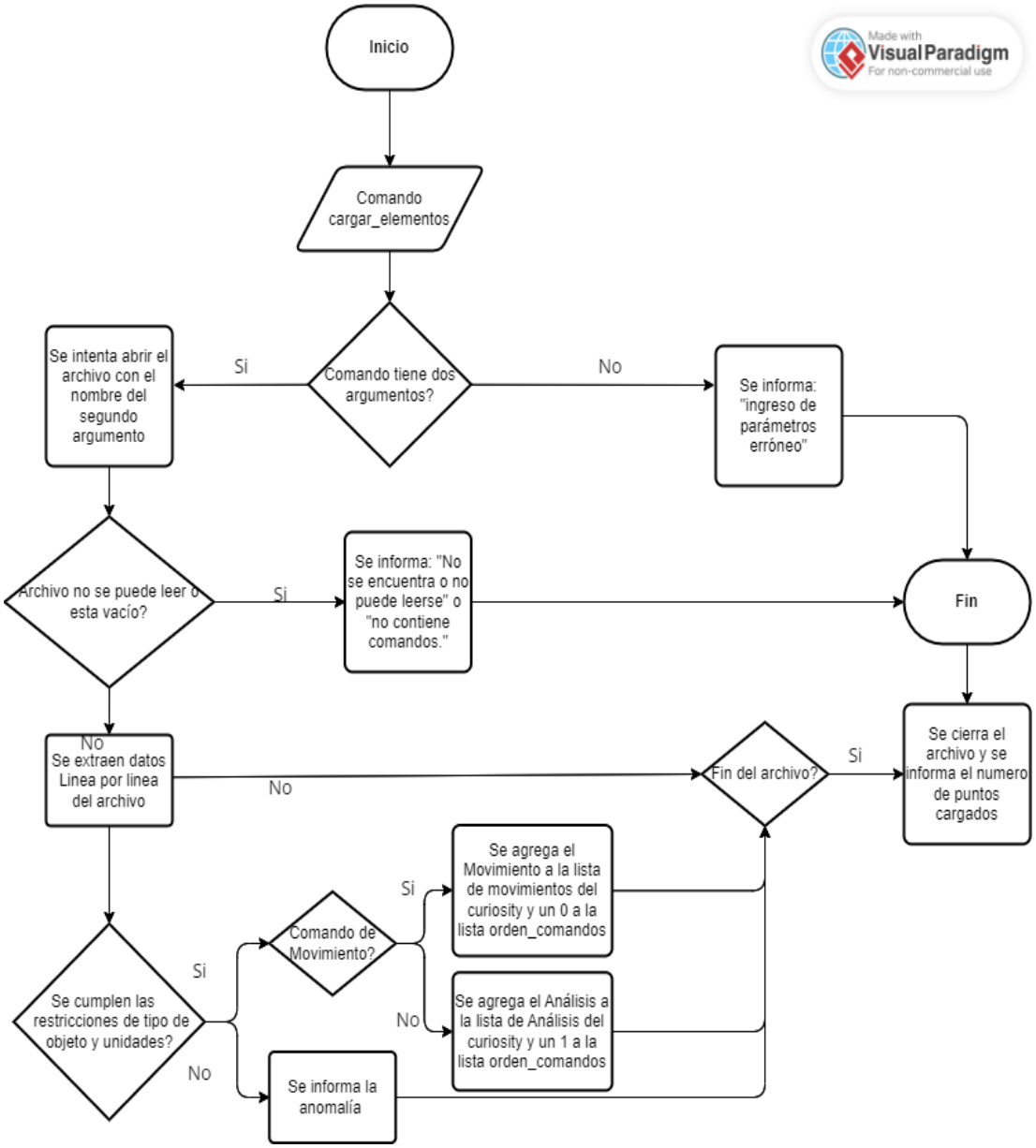


Figura 4. Diagrama de flujo comando cargar_elementos

***Descripción*:** Carga en memoria los comandos de cargar elementos que se encuentren en el respectivo archivo.

***Entrada*:** Se hace la lectura del respectivo archivo y se extrae el o los elementos deseados existentes.

Salida: Se tienen dos posibles salidas

- Si el archivo no existe o se encuentra vacío devuelve un mensaje de aviso al usuario.
- Si el archivo existe y cumple con los parámetros se carga los elementos y posteriormente avisando de su éxito al usuario.

5.4. Comando: agregar_movimiento.

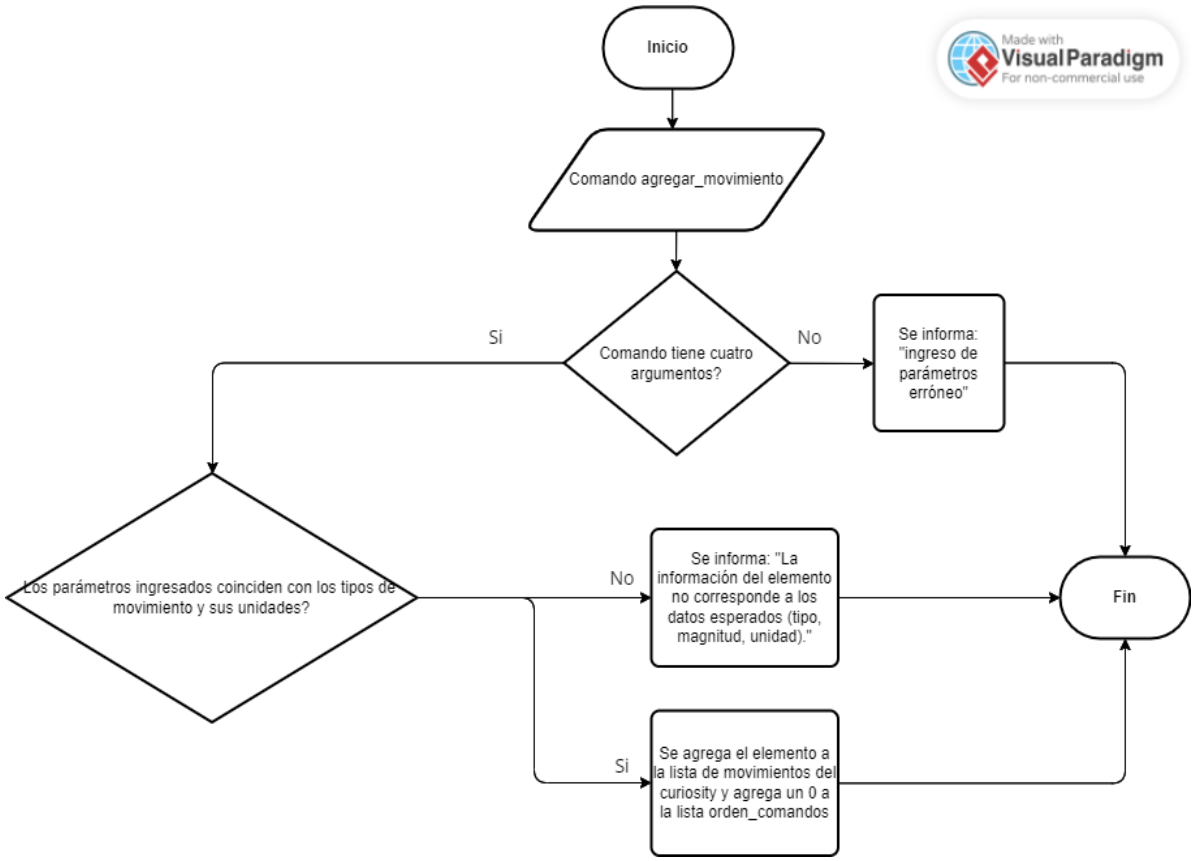


Figura 5. Diagrama de flujo comando agregar_movimiento

***Descripción*:** Carga en memoria el nuevo comando de movimiento ingresado por el usuario.

***Entrada*:** Nuevo movimiento.

Salida: Se tienen dos posibles salidas.

- En el caso de que los datos estén incompletos o sean erróneos se le notificará al usuario.
- En el caso de que los datos sean correctos se procederá a agregarlos a la lista de movimientos notificando al usuario el éxito del proceso.

5.5. Comando: agregar_analisis.

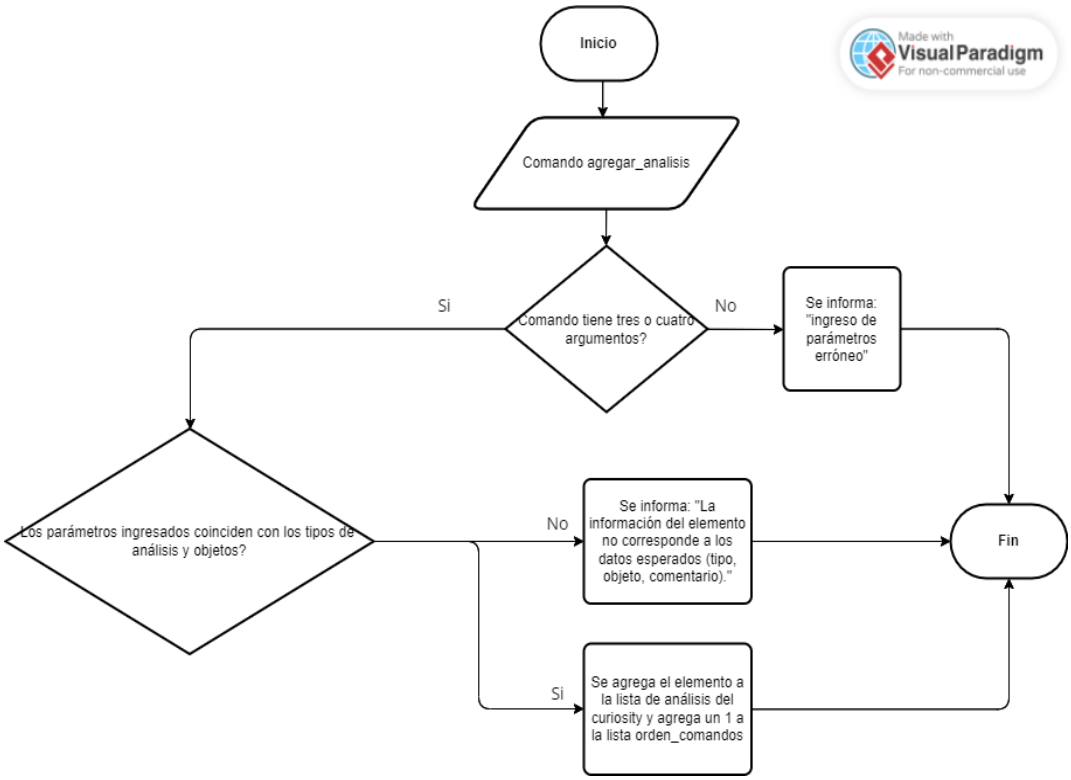


Figura 6. Diagrama de flujo comando agregar_analisis

***Descripción*:** Carga en memoria el nuevo comando de análisis ingresado por el usuario.

***Entrada*:** Nuevo análisis.

Salida: Se tienen dos posibles salidas.

- En el caso de que los datos estén incompletos o sean erróneos se le notificará al usuario.
- En el caso de que los datos sean correctos se procederá a agregarlos a la lista de análisis notificando al usuario el éxito del proceso.

5.6. Comando: agregar_elemento.

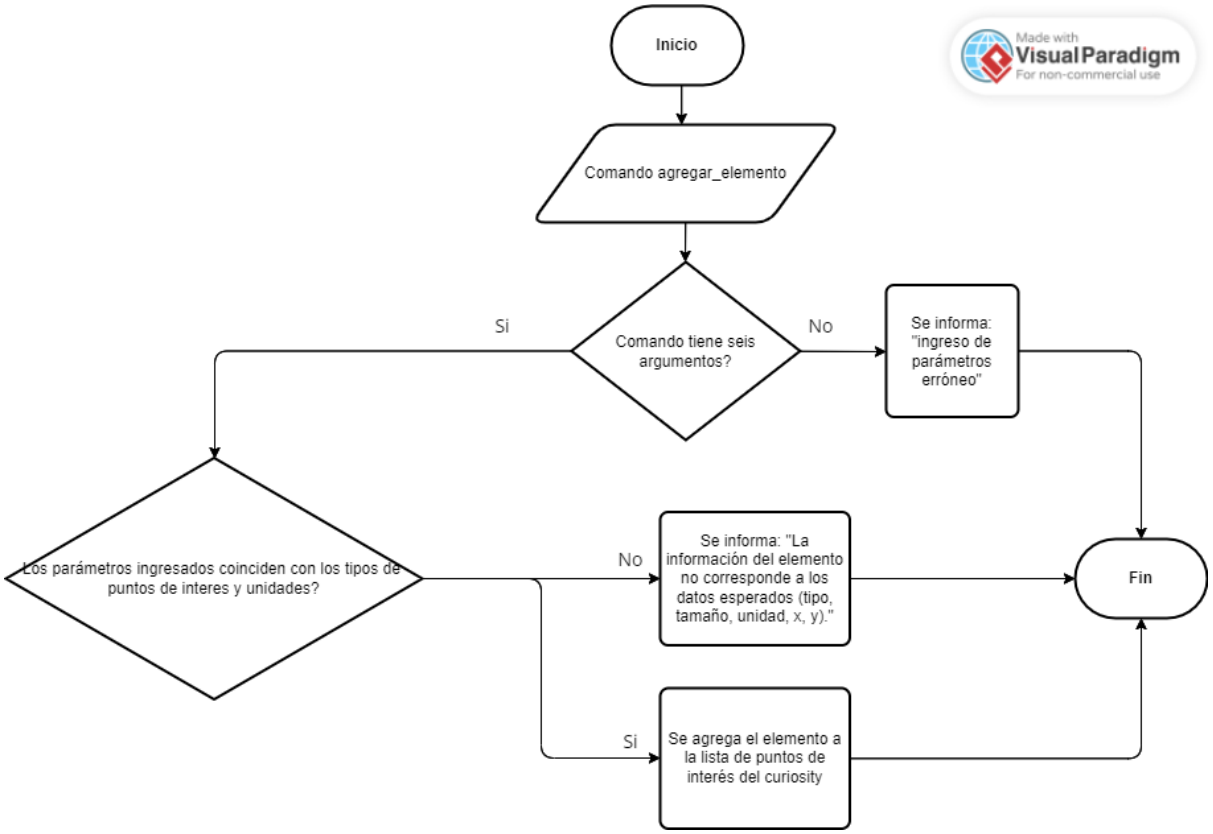


Figura 7. Diagrama de flujo comando agregar_elemento

***Descripción*:** Carga en memoria el nuevo elemento ingresado por el usuario.

***Entrada*:** Nuevo elemento.

Salida: Se tienen dos posibles salidas.

- En el caso de que los datos estén incompletos o sean erróneos se le notificará al usuario.
- En el caso de que los datos sean correctos se procederá a agregarlos a la lista de puntos de interés del robot notificando al usuario el éxito del proceso.

5.7. Comando: guardar.

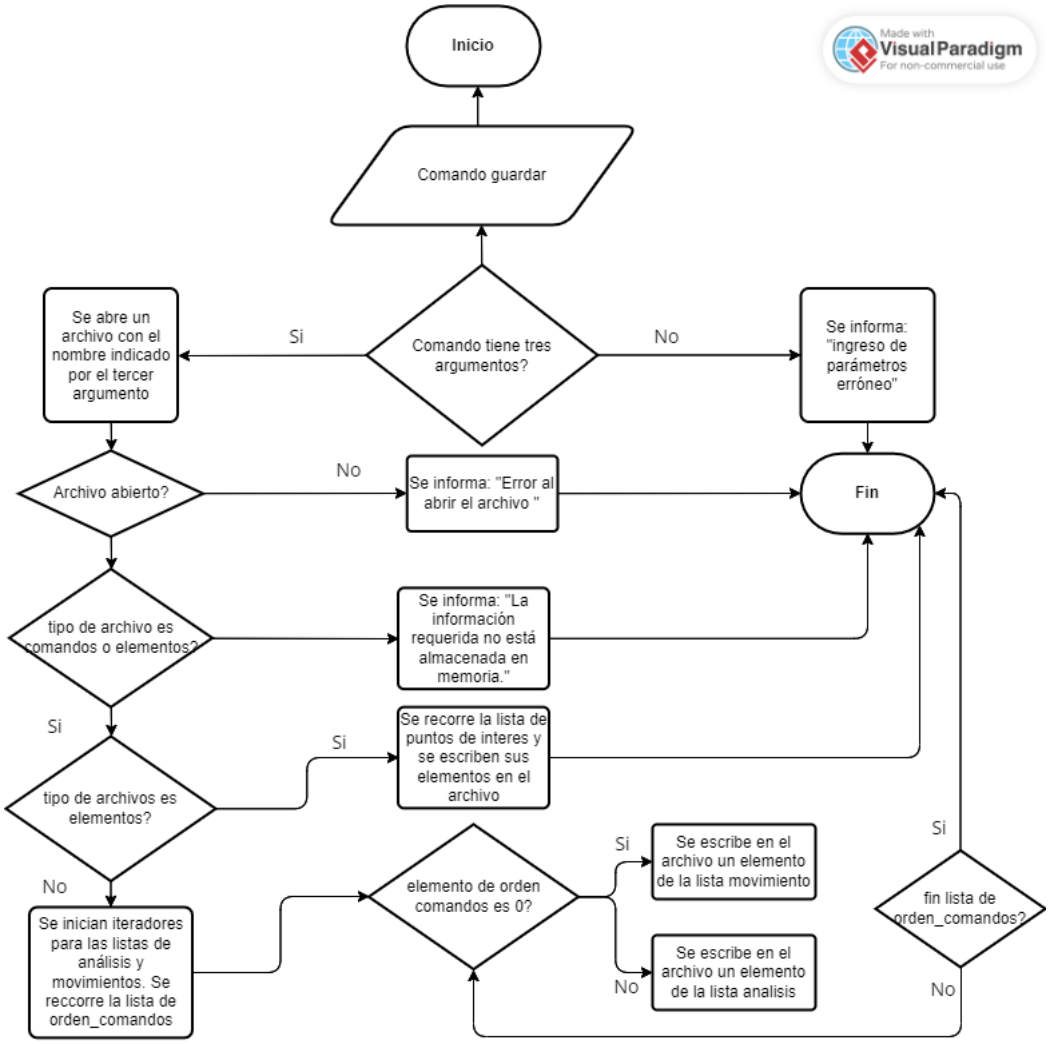


Figura 8. Diagrama de flujo comando guardar

***Descripción*:** Guarda en un archivo la información correspondiente.

***Entrada*:** Recibe el nombre que se le va a asignar y el tipo de archivo.

Salida: Se tienen tres posibles salidas.

- En el caso de que el archivo se encuentre abierto se le notifica al usuario del error.
- En el caso de que la extensión o que el nombre indicado sea incorrectos se le notifica al usuario del error.
- En el caso de que tanto el nombre como el tipo de archivo sean correcto y además se encuentre cerrado se le guardará la información en el archivo indicando el éxito al usuario.

5.8. Comando: simular_comandos.

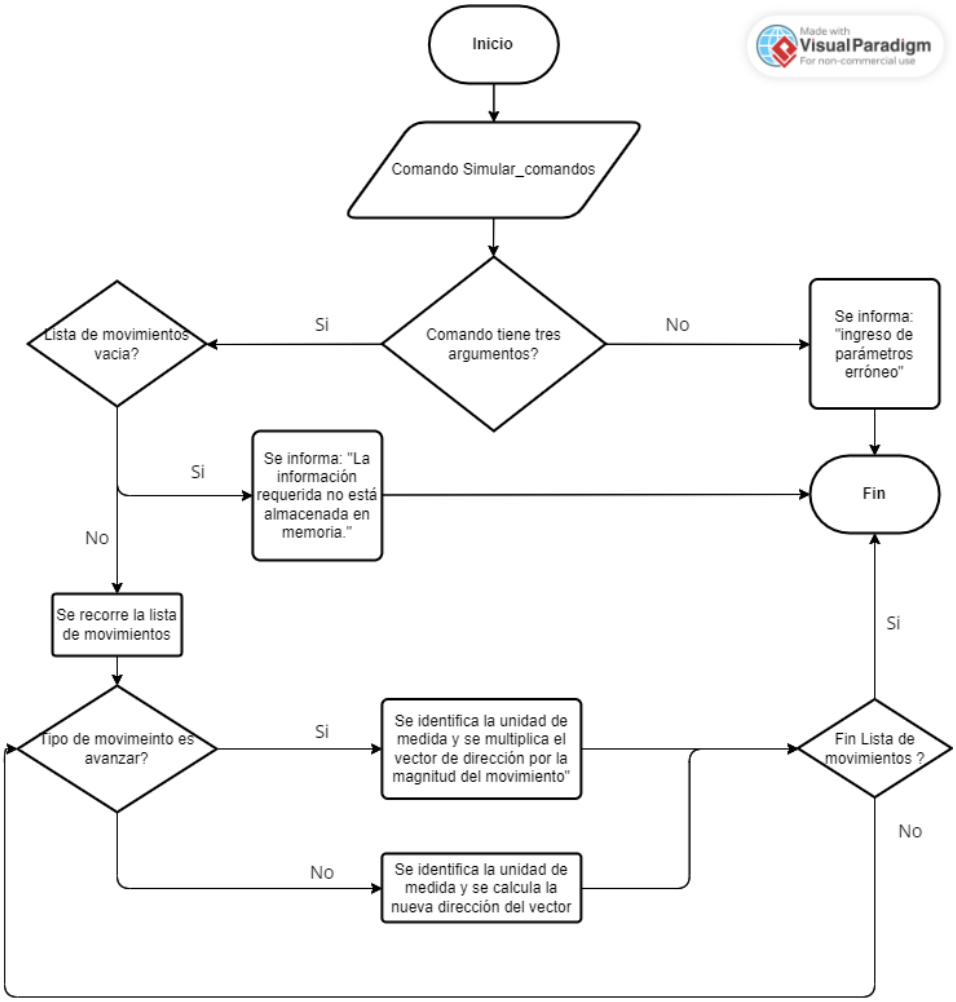


Figura 9. Diagrama de flujo comando simular_comandos

***Descripción*:** Simula el comportamiento de los comandos de movimiento que se le enviaran al robot lo cual permite validar la nueva posición donde se encuentra ubicado.

***Entrada*:** Coordenadas en “x” y en “y”.

Salida: Se tienen dos posibles salidas.

- En el caso de que los datos sean incorrectos o se encuentren incompletos se le notifica al usuario del error.
- En el caso de que los datos se encuentren completos sin errores se le notifica al usuario del éxito del proceso.

5.9. Comando: ubicar_elementos.

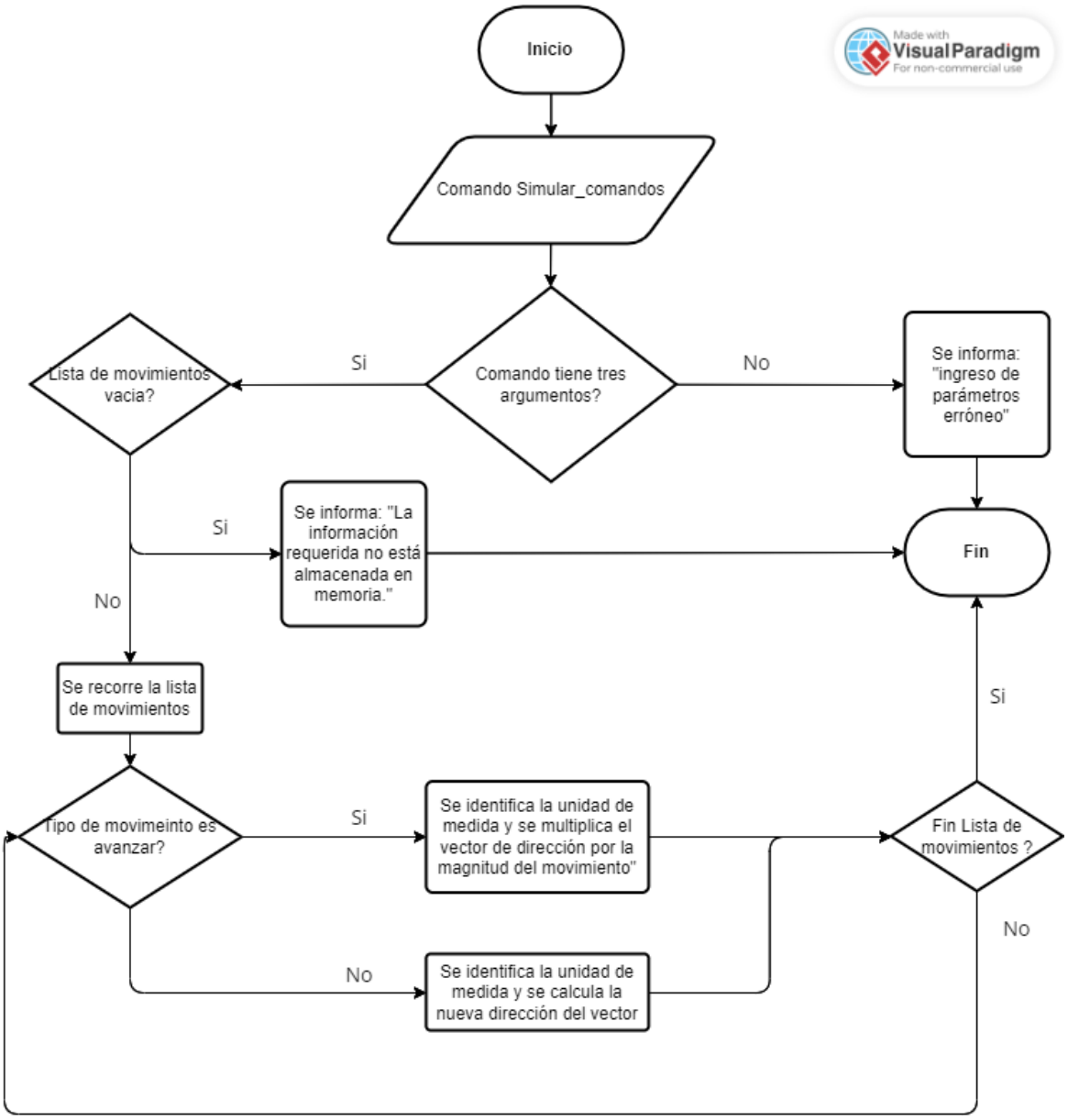


Figura 10. Diagrama de flujo comando ubicar_elementos

***Descripción*:** Utiliza la información de puntos de interés almacenados que permite luego realizar consultas geográficas sobre estos elementos.

***Entrada*:** nombre comando.

Salida: Se tienen dos posibles salidas.

5.10. Comando: en_cuadrante

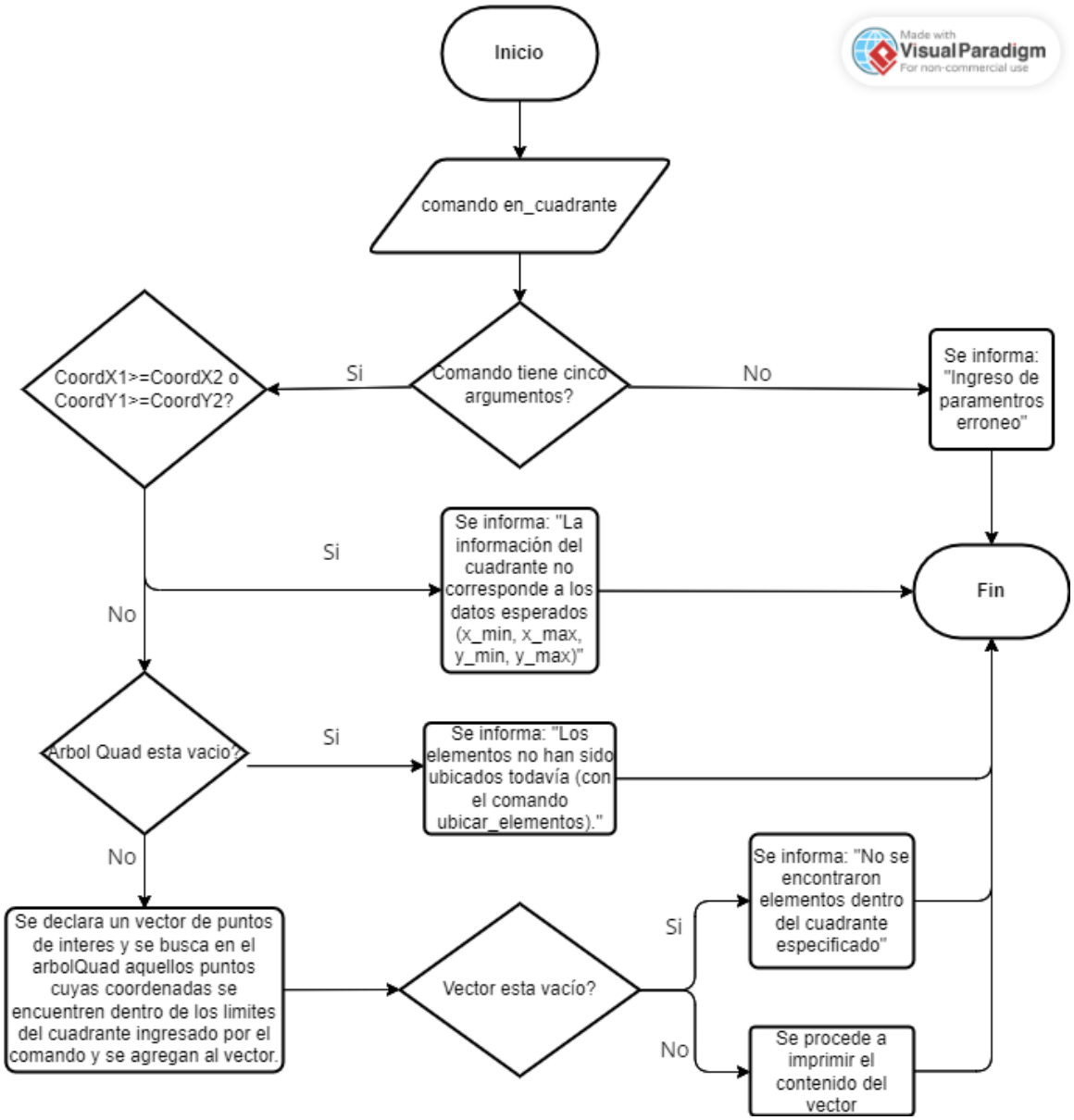


Figura 11. Diagrama de flujo comando en_cuadrante

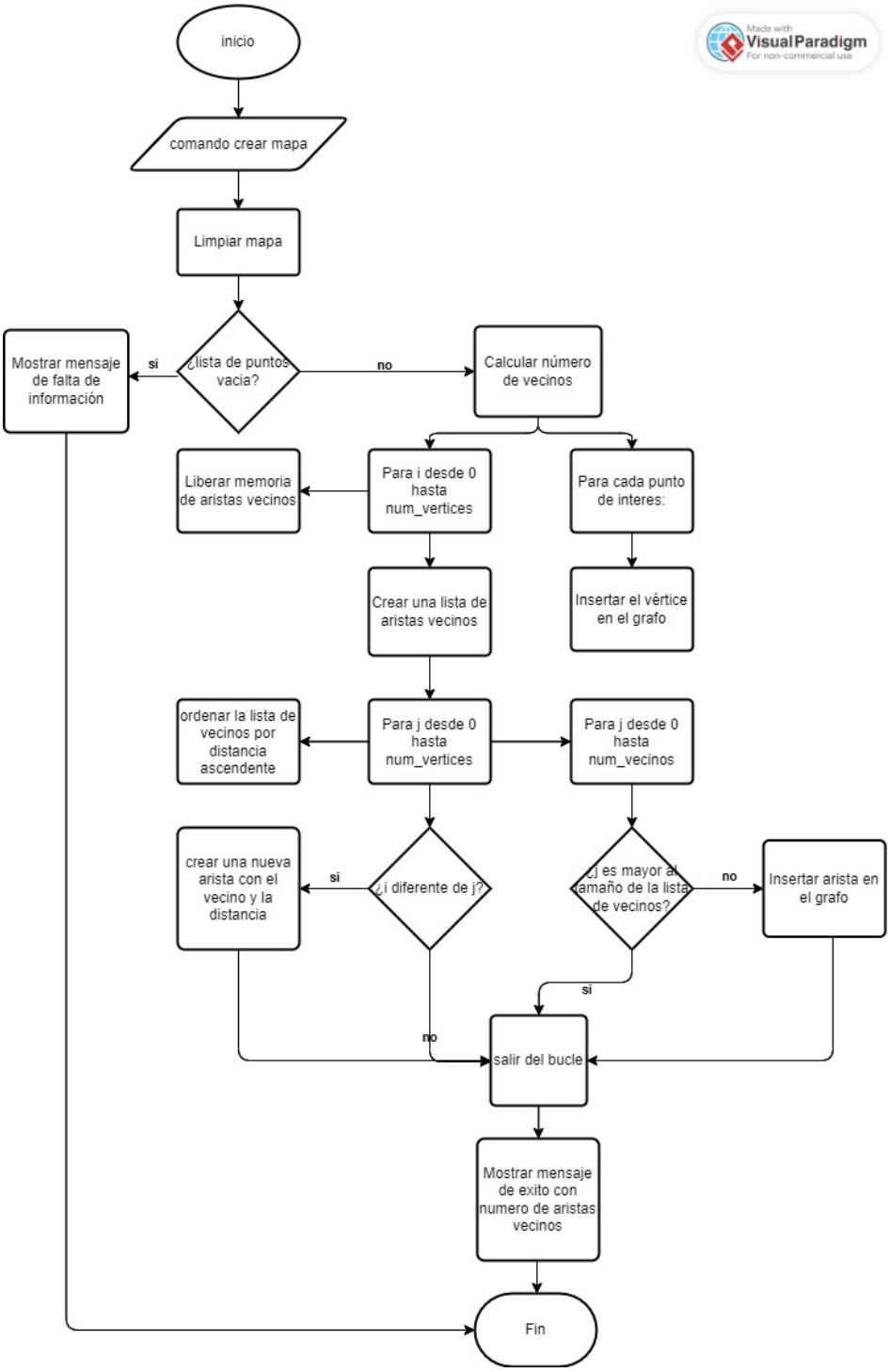
***Descripción*:** Utiliza la estructura creada con el comando anterior para retornar una lista de los componentes o elementos que están dentro del cuadrante geográfico descrito por los límites de coordenadas en “x” y “y”.

***Entrada*:** nombre comando y el conjunto de coordenadas, dos en “x” y dos “y”.

Salida: Se tienen dos posibles salidas.

- Retorna una lista de componentes utilizando la estructura del comando anterior.
- En el caso de que los datos sean erróneos se le notificará al usuario del error.

5.11. Comando: crear mapa



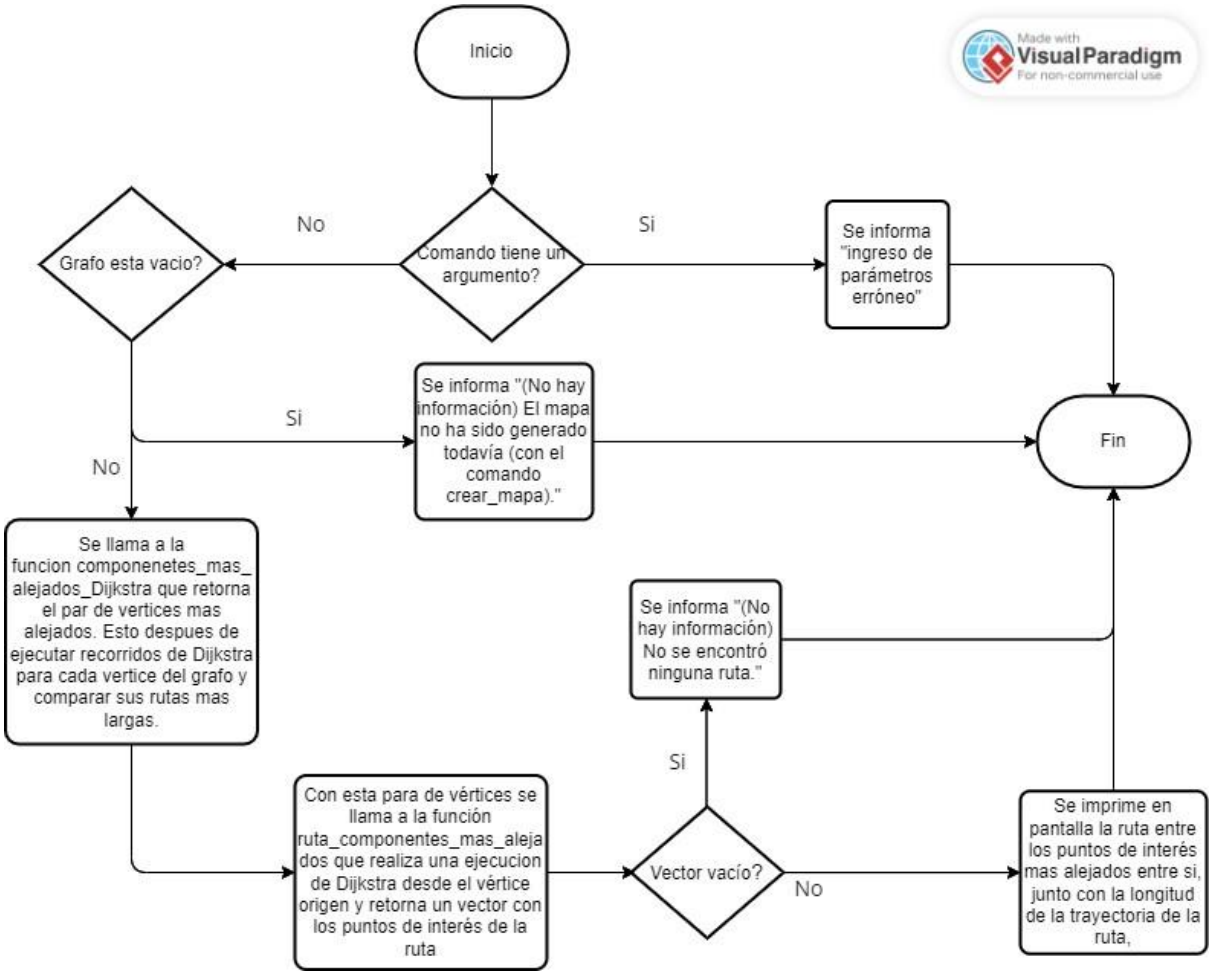
Descripción: Esta función se encarga de crear un mapa representado por un grafo, utilizando una lista de puntos de interés y un coeficiente de conectividad. El objetivo es conectar los puntos de interés en el grafo de manera que cada punto tenga un número específico de vecinos determinado por el coeficiente de conectividad.

Entradas: Una referencia al objeto del tipo Grafo en el cual se construirá el mapa. y el coeficiente_conectividad: Un número decimal que indica la proporción de vecinos que

cada punto de interés debe tener. Este coeficiente determina el número de vecinos en función del tamaño de la lista de puntos de interés.

Salidas: Mensajes en la consola que indican el progreso y el resultado del proceso de creación del mapa.

5.12. Comando: ruta_mas_larga



Descripción: La función se encarga de encontrar y mostrar la ruta más larga entre los puntos de interés en un grafo. Utiliza el algoritmo de Dijkstra para determinar los dos puntos más alejados en el grafo y luego encuentra la ruta que los conecta.

Entradas: Referencia al objeto del tipo Grafo en el cual se buscará la ruta más larga.

Salidas: Mensajes en la consola que indican el progreso y el resultado del proceso de búsqueda de la ruta más larga.

6. Plan de pruebas

6.1. Plan de pruebas función simular comandos:

Descripción plan de pruebas:

6.1.1. Prueba 1: No hay información en el sistema

En esta prueba intentamos usar la función de simular comandos para comprobar que cuando no hubiera información de que movimientos realizar en el sistema, se imprimiera por pantalla un mensaje de error que comunicara al usuario de que no había información suficiente para simular comandos.

- **Valores ingresados por el usuario:** (0,0)
- **Salida esperada:** Mensaje de error (La información requerida no esta almacenada en memoria)
- **Evidencia de resultado:**

Figura 12. Evidencia prueba 1 simular_comandos

6.1.2. Prueba 2: Valores ingresados de manera incorrecta

En esta prueba intentamos usar la función de simular comandos para comprobar que pasaba si ingresábamos parámetros incorrectos, por ejemplo, si ingresábamos como parámetros letras en vez de números, esperando que por pantalla también nos muestre un mensaje de error

- **Valores ingresados por el usuario:** (a,b)
- **Salida esperada:** Mensaje de error (valores ingresados incorrectamente)
- **Evidencia de resultado:**

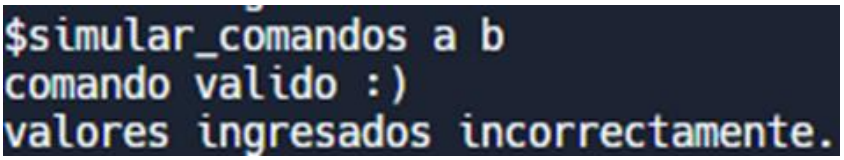


Figura 13. Evidencia prueba 2 simular_comandos

6.1.3. Prueba 3: Valores ingresados correctamente

En esta prueba intentaremos usar el comando después de haber almacenado en memoria la información necesaria y utilizar valores correctos para llamarla.

- **Información cargada desde memoria:**

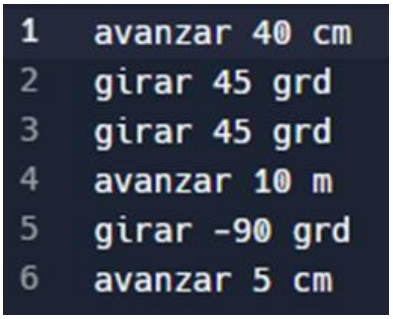


Figura 14. Evidencia prueba 3 información cargada simular_comandos

- **Valores ingresados por el usuario:** (0,0)
- **Salida esperada:** (0.45, 10)
- **Evidencia de resultado:**

Figura 15. Evidencia prueba 3 simular_comandos

6.1.4. Prueba 4: Valores ingresados correctamente desde posiciones negativas

En esta prueba intentaremos usar el comando ya habiendo almacenado en memoria la información necesaria y empezando desde posiciones negativas

- **Información cargada desde memoria:**

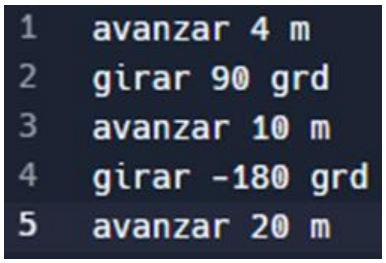


Figura 16. Evidencia prueba 4 información cargada simular_comandos

- **Valores ingresados por el usuario:** (-5,-5)
- **Salida esperada:** (-1, -15)
- **Evidencia de resultado:**

```
$simular_comandos -5 -5
comando valido :)
La simulación de los comandos, a partir de la posición (-5,-5), deja al robot en la nueva posición (-1,-15).
Unidades en metros.
```

Figura 17. Evidencia prueba 4 simular_comandos

6.1.5. Prueba 5: Valores ingresados correctamente desde posiciones positivas

En esta prueba intentaremos usar el comando, ya con la información cargada en memoria y empezando la simulación desde posiciones positivas

- **Información cargada desde memoria:**

```
1  avanzar 4 m
2  girar 90 grd
3  avanzar 10 m
4  girar -180 grd
5  avanzar 20 m
```

Figura 18. Evidencia prueba 5 información cargada simular_comandos

- **Valores ingresados por el usuario:** (100,200)
- **Salida esperada:** (104, 190)
- **Evidencia de resultado:**

```
$simular_comandos 100 200
comando valido :)
La simulación de los comandos, a partir de la posición (100,200), deja al robot en la nueva posición (104,190).
Unidades en metros.
```

Figura 19. Evidencia prueba 5 simular_comandos

Plan de pruebas: Función simular comandos				
Caso	**Información cargada en memoria**	**Valores de entrada**	**Resultado esperado**	**Resultado obtenido**
Prueba 1	NO	(0,0)	Mensaje de error	Mensaje de error (La información requerida no está almacenada en memoria)

Prueba 2	SI	(a,b)	Mensaje de error	Mensaje de error (valores ingresados incorrectamente)
Prueba 3	SI	(0,0)	(0.45, 10)	(0.45, 10)
Prueba 4	SI	(-5,-5)	(-1, -15)	(-1, -15)
Prueba 5	SI	(100,200)	(104, 190)	(104, 190)

Figura 20. Tabla plan de pruebas simular_comandos

6.2. Plan de pruebas función en cuadrante:

Descripción plan de pruebas:

6.2.1. Prueba 1: Elementos no ubicados

En esta prueba intentamos usar la función de simular comandos para comprobar que cuando no hubiera información de los objetos ubicados, lanzara un mensaje de error indicándolo.

- **Valores ingresados por el usuario:** (0, 5, 10, 15)
- **Salida esperada:** Mensaje de error (Los elementos no han sido ubicados todavía (con el comando ubicar_elementos)
- **Evidencia de resultado:**

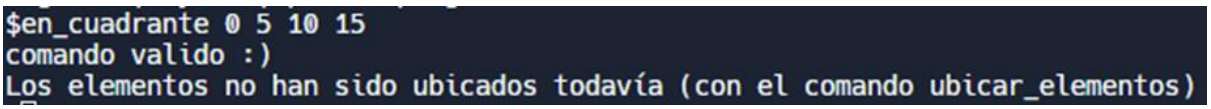


Figura 21. Evidencia prueba 1 en_cuadrante

6.2.2. Prueba 2: Valores ingresados de manera incorrecta

En esta prueba intentamos usar la función de simular comandos para comprobar que pasaba si ingresábamos parámetros no esperados, es decir que, a pesar de ya haber ubicado los elementos, al momento de llamar a la función, no ingresamos bien los parámetros, por ejemplo, si ponemos el primero parámetro mayor al segundo parámetro de X

- **Valores ingresados por el usuario:** (1, 0, -5, 1)

- **Salida esperada:** Mensaje de error (La información del cuadrante no corresponde a los datos esperados (x_min, x_max, y_min, y_max))

- **Evidencia de resultado:**

Figura 22. Evidencia prueba 2 en_cuadrante

6.2.3. Prueba 3: Valores ingresados de manera incorrecta

En esta prueba intentaremos usar el comando después de haber almacenado en memoria la información necesaria, pero utilizando valores incorrectos como letras en vez de números

- **Valores ingresados por el usuario:** (a, 0, 3, b)

- **Salida esperada:** Mensaje de error (valores ingresados incorrectamente.)

- **Evidencia de resultado:**

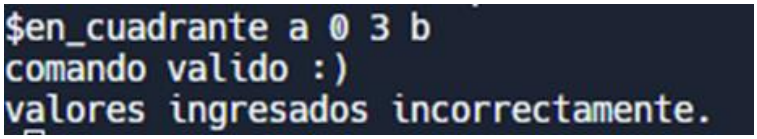


Figura 23. Evidencia prueba 3 en_cuadrante

Información cargada desde memoria para las siguientes pruebas:

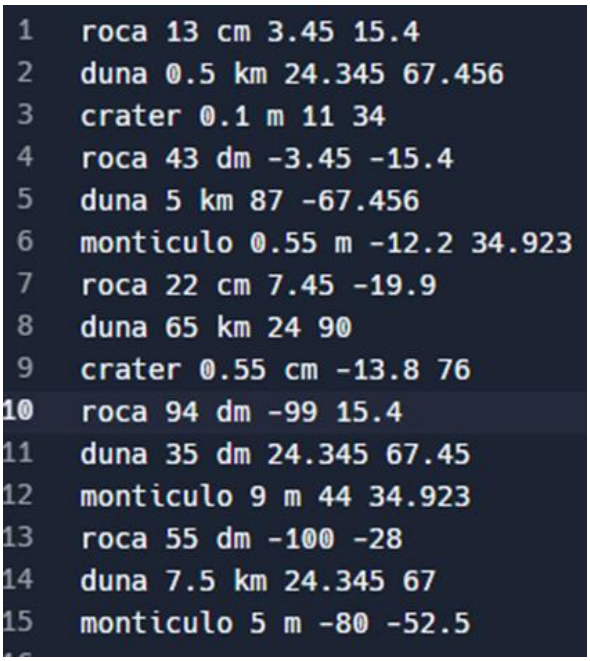


Figura 24. Evidencia información cargada en_cuadrante

6.2.4. Prueba 4: Valores ingresados correctamente cuadrante 1

En esta prueba intentaremos usar el comando ya habiendo almacenado en memoria la información necesaria y probando que aparezcan todos los elementos como resultado del cuadrante 1

- **Valores ingresados por el usuario:** (0, 100, 0, 100)
- **Salida esperada:**
  - roca (3.45, 15.4)
  - duna (24.345, 67.456)
  - duna (24, 90)
  - crater (11, 34)
  - duna (24.345, 67.45)
  - duna (24.345, 67)
  - monticulo (44, 34.923)
- **Evidencia de resultado:**

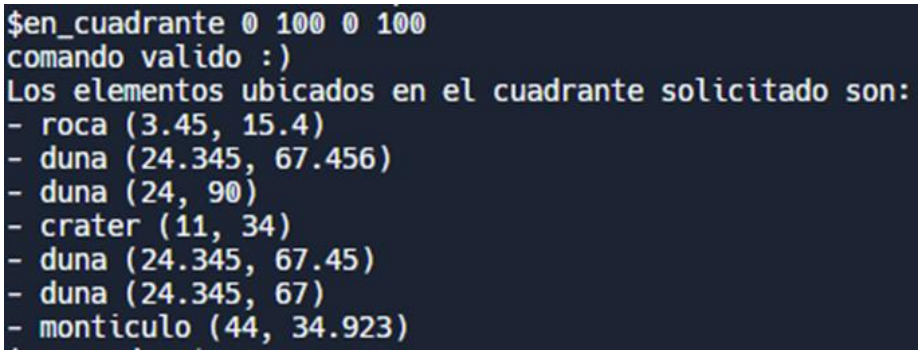


Figura 25. Evidencia prueba 4 en_cuadrante

6.2.5. Prueba 5: Valores ingresados correctamente cuadrante 2

Ya habiendo almacenado en memoria los resultados, queremos comprobar que los elementos del cuadrante 2 aparezcan correctamente en memoria

- **Valores ingresados por el usuario:** (-100, 0, 0, 100)
- **Salida esperada:**
  - monticulo (-12.2, 34.923)
  - crater (-13.8, 76)

- roca (-99, 15.4)

- **Evidencia de resultado:**

```
$en_cuadrante -100 0 0 100
comando valido :)
Los elementos ubicados en el cuadrante solicitado son:
- monticulo (-12.2, 34.923)
- crater (-13.8, 76)
- roca (-99, 15.4)
```

Figura 26. Evidencia prueba 5 en_cuadrante

6.2.6. Prueba 6: Valores ingresados correctamente cuadrante 3

Usaremos el comando para comprobar que una vez almacenada la información en memoria, aparecen los elementos correctos del cuadrante 3

- **Valores ingresados por el usuario:** (-100, 0, -100, 0)

- **Salida esperada:**

  - roca (-3.45, -15.4)

  - roca (-100, -28)

  - monticulo (-80, -52.5)

- **Evidencia de resultado:**

```
$en_cuadrante -100 0 -100 0
comando valido :)
Los elementos ubicados en el cuadrante solicitado son:
- roca (-3.45, -15.4)
- roca (-100, -28)
- monticulo (-80, -52.5)
```

Figura 27. Evidencia prueba 6 en_cuadrante

6.2.7. Prueba 7: Valores ingresados correctamente cuadrante 4

Comprobaremos al igual que con las anteriores pruebas que ya estando la información guardada en memoria, aparecen todos los elementos del cuadrante 4

- **Valores ingresados por el usuario:** (0, 100, -100, 0)

- **Salida esperada:**

  - duna (87, -67.456)

- roca (7.45, -19.9)

- **Evidencia de resultado:**

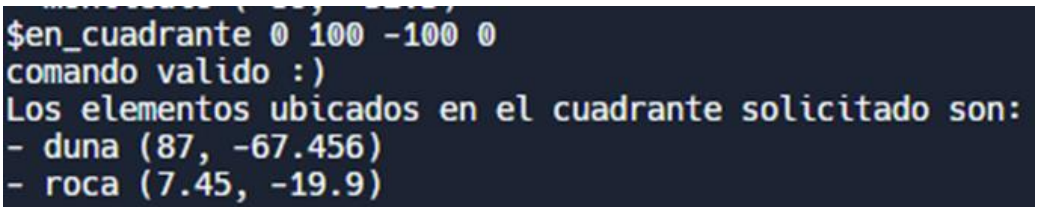


Figura 28. Evidencia prueba 7 en_cuadrante

Plan de pruebas: Función en cuadrante				
Caso	**Elementos ubicados**	**Valores de entrada**	**Resultado esperado**	**Resultado obtenido**
Prueba 1	NO	(0, 5, 10, 15)	Mensaje de error	Mensaje de error (Los elementos no han sido ubicados todavía (con el comando ubicar_elementos)
Prueba 2	SI	(1, 0, -5, 1)	Mensaje de error	Mensaje de error (La información del cuadrante no corresponde a los datos esperados (x_min, x_max, y_min, y_max))
Prueba 3	SI	(a, 0, 3, b)	Mensaje de error	Mensaje de error (valores ingresados incorrectamente.)

Prueba 4	SI	(0, 100, 0, 100)	- roca (3.45, 15.4) - duna (24.345, 67.456) - duna (24, 90) - crater (11, 34) - duna (24.345, 67.45) - duna (24.345, 67) - monticulo (44, 34.923)	- roca (3.45, 15.4) - duna (24.345, 67.456) - duna (24, 90) - crater (11, 34) - duna (24.345, 67.45) - duna (24.345, 67) - monticulo (44, 34.923)
Prueba 5	SI	(-100, 0, 0, 100)	- monticulo (-12.2, 34.923) - crater (-13.8, 76) - roca (-99, 15.4)	- monticulo (-12.2, 34.923) - crater (-13.8, 76) - roca (-99, 15.4)
Prueba 6	SI	(-100, 0, -100, 0)	- roca (-3.45, -15.4) - roca (-100, -28) - monticulo (-80, -52.5)	- roca (-3.45, -15.4) - roca (-100, -28) - monticulo (-80, -52.5)
Prueba 7	SI	(0, 100, -100, 0)	- duna (87, -67.456) - roca (7.45, -19.9)	- duna (87, -67.456) - roca (7.45, -19.9)

Figura 29. Tabla plan de pruebas en_cuadrante

6.3. Plan de pruebas función ruta_mas_larga:

6.3.1. Prueba 1: ingresar el comando sin haber creado el mapa

En esta prueba lo que se quiere mostrar es que cuando se ingresa el comando sin haber creado previamente el mapa debe salir un mensaje informando el suceso.

```
$ruta_mas_larga
comando valido :)
(No hay información) El mapa no ha sido generado todavía (con el com
ando crear_mapa).
```

Figura 30. Figura prueba ruta_mas_larga

Una vez creado el mapa se continuará con el plan de pruebas.

6.3.2. Prueba 2: Ingresar el comando con el mapa creado previamente

En esta prueba se realiza teniendo un mapa creado previamente. En este caso se probó con el mapa creado con el coeficiente de 0.4.

- **Valores ingresados por el usuario:** 0.4
- **Salida esperada:** (Resultado exitoso) Los puntos de interés más alejados entre sí son duna (24,90) y roca (-99,15.4).
  La ruta que los conecta tiene una longitud total de longitud :264.701
  La ruta que los conecta tiene una cantidad de aristas de: 5
  duna - (24 , 90)
  cráter - (11 , 34)
  roca - (3.45 , 15.4)
  roca - (-3.45 , -15.4)
  montículo - (-80 , -52.5)
  montículo - (-99 , 15.4)
  roca - (-99 , 15.4)

- **Evidencia de resultado:**

```
(Resultado exitoso) Los puntos de interés más alejados entre sí son du
na  (24,90)     y roca  (-99,15.4).
La ruta que los conecta tiene una longitud total de longitud  :264.701
La ruta que los conecta tiene una catidad de aristas de: 5
duna  -  (24 , 90)
crater  -  (11 , 34)
roca  -  (3.45 , 15.4)
roca  -  (-3.45 , -15.4)
monticulo  -  (-80 , -52.5)
monticulo  -  (-99 , 15.4)
roca  -  (-99 , 15.4)
```

Figura 31. Figura prueba2 ruta_mas_larga

6.3.3. Prueba 3: Ingresar el comando con el mapa creado previamente con coeficiente 1

Para esta prueba se quiere mostrar otro ejemplo de un mapa, pero con un coeficiente diferente y poder evidenciar cómo se comporta el algoritmo.

- **Valores ingresados por el usuario:** 1
- **Salida esperada:** (Resultado exitoso) Los puntos de interés más alejados entre sí son duna (87,-67.456) y roca (-99,15.4).
  La ruta que los conecta tiene una longitud total de longitud :203.62
  La ruta que los conecta tiene una cantidad de aristas de: 1
  duna - (87, -67.456)
  duna - (-99, 15.4)
  roca - (-99, 15.4)

- **Evidencia de resultado:**

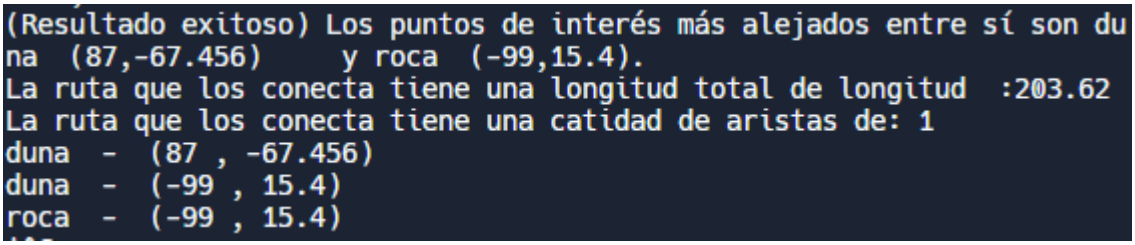


Figura 32. Figura prueba 3 ruta_mas_larga

Plan de pruebas: Función ruta_mas_larga				
Caso	**Elementos ubicados**	**Valores de entrada**	**Resultado esperado**	**Resultado obtenido**
Prueba 1	NO	NINGUNO	Mensaje de error	Mensaje de error (Los elementos no han sido ubicados todavía (con el comando ubicar_elementos)
Prueba 2	SI	0.4	(Resultado exitoso) Los puntos de interés más alejados entre sí son duna (24,90) y roca (-99,15.4).	(Resultado exitoso) Los puntos de interés más alejados entre sí son duna (24,90) y roca (-99,15.4).

			La ruta que los conecta tiene una longitud total de longitud :264.701 La ruta que los conecta tiene una cantidad de aristas de: 5 duna - (24 , 90) cráter - (11 , 34) roca - (3.45 , 15.4) roca - (-3.45 , -15.4) montículo - (-80 , -52.5) montículo - (-99 , 15.4) roca - (-99 , 15.4)	La ruta que los conecta tiene una longitud total de longitud :264.701 La ruta que los conecta tiene una cantidad de aristas de: 5 duna - (24 , 90) cráter - (11 , 34) roca - (3.45 , 15.4) roca - (-3.45 , -15.4) montículo - (-80 , -52.5) montículo - (-99 , 15.4) roca - (-99 , 15.4)
Prueba 3	SI	1	Los puntos de interés más alejados entre sí son duna (87,-67.456) y roca (-99,15.4). La ruta que los conecta tiene una longitud total de longitud :203.62 La ruta que los conecta tiene una cantidad de aristas de: 1 duna - (87, -67.456) duna - (-99, 15.4) roca - (-99, 15.4)	Los puntos de interés más alejados entre sí son duna (87,-67.456) y roca (-99,15.4). La ruta que los conecta tiene una longitud total de longitud :203.62 La ruta que los conecta tiene una cantidad de aristas de: 1 duna - (87, -67.456) duna - (-99, 15.4) roca - (-99, 15.4)