Proyecto del Curso

ESTRUCTURAS DE DATOS

Estudiantes:

Laura Valentina Ovalle Benítez

Valentina García Alfonso

Juan Miguel Zuluaga Suárez

Carlos Gabriel López

Nicolás Padilla Medina

PRESENTADO A:

John Jairo Corredor Franco

PONTIFICIA UNIVERSIDAD JAVERIANA

BOGOTÁ D.C

2023

1. Descripción del problema

El vehículo “Curiosity”, desarrollado por la NASA, es un robot explorador que lleva a cabo diversos experimentos científicos. Su misión principal es buscar condiciones pasadas o presentes favorables para la vida y condiciones capaces de conservar registros de vida en áreas específicas del planeta Marte. Para cumplir con su misión, el explorador debe desplazarse por el entorno marciano (caracterizado por arena, rocas y montículos) para recolectar fotografías y muestras del entorno. El desplazamiento del vehículo explorador se hace mediante secuencias de comandos enviadas desde la Tierra la noche anterior al día en que se desea sean ejecutadas.

Dependiendo de la irregularidad del terreno, el explorador tiene dos maneras diferentes de desplazarse hacia un punto geográfico:

1. Desplazarse según los comandos enviados la noche anterior.
2. Desplazarse de manera autónoma por el terreno hasta una ubicación dada.

Para diseñar su camino, el robot explorador se basa en mapas de profundidad generados gracias a dos cámaras ubicadas al frente del vehículo. El desplazamiento del “Curiosity” es una tarea crítica en el suelo marciano, ya que una mala planeación y ejecución de la trayectoria puede terminar en un estancamiento del vehículo, poniendo fin así a sus operaciones de exploración. Para validar la correcta ejecución de los comandos de desplazamiento enviados desde la Tierra, el vehículo explorador tiene definido un mecanismo de retroalimentación y validación, utilizando fotografías tomadas por medio de las cámaras instaladas en el frente del vehículo y el análisis de los componentes del suelo marciano.

2. Justificación

De acuerdo a la problemática ya mencionada, y, para llevar a cabo el desarrollo del proyecto. Se propone a lo largo de este documento dividir en 3 diferentes componentes donde se abarcará la totalidad del problema y a su vez, se propondrá una solución para la realización del proyecto “Curiosity”.

- **Componente 1:**
  En este componente se realizará todo el manejo de información. Es en esta fase donde se administrarán tanto los comandos de desplazamiento como la información de los puntos de interés encontrados en suelo marciano.

- **Componente 2:**
  En este componente se utilizará una estructura jerárquica que permita almacenar datos de los puntos de interés sobre el cielo marciano para así facilitar la planificación automática de desplazamientos

- **Componente 3:**
  En este componente, mediante la información recopilada de los puntos de interés se crearán representaciones geográficas (mapas) que permitan posteriormente identificar regiones de interés sobre el suelo marciano ya sea para aterrizajes o para exploración.

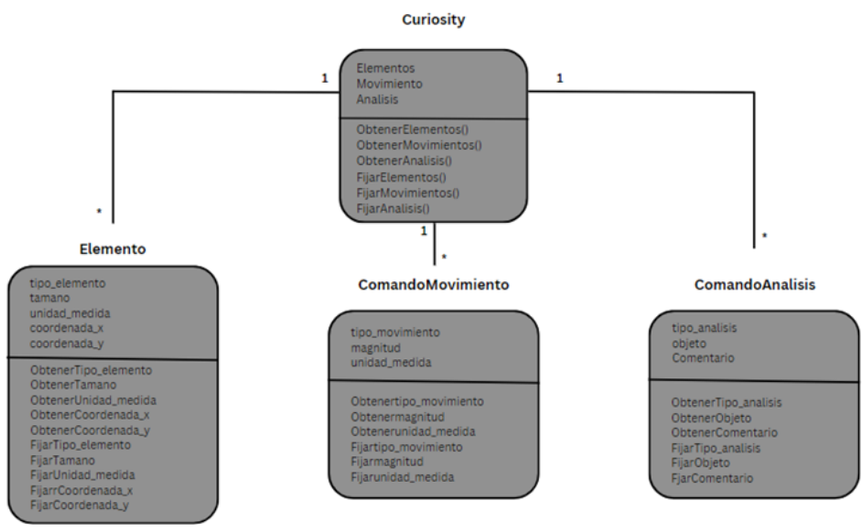
3. Desarrollo componente 1

Objetivo: Realizar la administración de los comandos de desplazamiento del vehículo y de la información de puntos de interés encontrados en el suelo marciano. Este componente se implementará utilizando las estructuras lineales que se consideren adecuadas, y contiene los siguientes comandos:

- cargar_comandos nombre_archivo
- cargar_elementos nombre_archivo
- agregar_movimiento tipo_mov magnitud unidad_med
- agregar_analisis tipo_analisis objeto comentario
- agregar_elemento tipo_comp tamaño unidad_med coordX coordY
- guardar tipo_archivo nombre_archivo
- simular_comandos coordX coordY
- salir
- ayuda

3.1. Declaración de TAD's

Para el desarrollo del primer componente del proyecto se definieron los siguientes TAD's:



a. Curiosity

Datos mínimos:

- Elementos, lista de elementos, permite identificar la complejidad del terreno sobre el cual debe moverse el vehículo
- listaMovimientos, lista de comandos de movimiento, permite al robot desplazarse sobre la superficie de Marte
- listaAnalisis, lista de comandos de análisis, permiten al robot investigar la superficie de Marte para analizar los elementos que va encontrando en su desplazamiento

Operaciones:

- ObtenerElementos(), retorna la lista de elementos actual que se encuentran en el suelo que recorre el Curiosity
- ObtenerMovimientos(), retorna la lista de comandos de movimiento actual que debe realizar el Curiosity
- ObtenerAnalisis(), retorna la lista de comandos de análisis actual que debe realizar el Curiosity

- FijarElementos(), fija una nueva lista de elementos para el suelo que recorre el Curiosity
- FijarMovimientos(), fija una nueva lista de comandos de movimiento que debe realizar el Curiosity
- FijarAnalisis(), fija una nueva lista de comando de análisis que debe realizar el Curiosity
- AgregarElemento(), agrega un elemento a la lista de elementos
- AgregarMovimiento(), agrega un movimiento a la lista de movimientos
- AgregarAnalisis(), agrega un análisis a la lista de análisis
- GetElementoBegin(), retorna el inicio de la lista de elementos
- GetElementoEnd(), retorna el fin de la lista de elementos
- GetMovimientoBegin(), retorna el inicio de la lista de movimientos
- GetMovimientoEnd(), retorna el fin de la lista de movimientos
- GetAnalisisBegin(), retorna el inicio de la lista de análisis
- GetAnalisisEnd(), retorna el fin de la lista de análisis

b. Elemento

Datos mínimos:

- Tipo_elemento, cadena de caracteres, describe el tipo del elemento que fue hallado por el Curiosity en el terreno. Puede ser roca, cráter, montículo o duna.
- Tamaño, entero, es el valor de la dimensión del elemento
- Unidad_medida, cadena de caracteres, define la unidad con la que se realizó la medición del tamaño del elemento.
- Coordenada_x, decimal, indica la posición del elemento sobre el eje x en el plano cartesiano
- Coordenada_y, decimal, indica la posición del elemento sobre el eje y en el plano cartesiano

Operaciones:

- ObtenerTipo_elemento(), retorna el tipo actual que tiene el elemento
- ObtenerTamaño(), retorna el tamaño actual que tiene el elemento
- ObtenerUnidad_medida(), retorna la unidad de medida actual con la que se realizó la medición del elemento
- ObtenerCoordenada_x(), retorna la coordenada actual del elemento en el eje x del plano cartesiano
- ObtenerCoordenada_y(), retorna la coordenada actual del elemento en el eje y del plano cartesiano
- FijarTipo_elemento(), fija un nuevo tipo para el elemento
- FijarTamaño(), fija un nuevo tamaño para el elemento
- FijarUnidad_medida(), fija una nueva unidad de medida para el elemento
- FijarCoordenada_x(), fija una nueva coordenada en el eje x para el elemento
- FijarCoordenada_y(), fija una nueva coordenada en el eje y para el elemento

c. Movimiento

Datos mínimos:

- Tipo_movimiento, cadena de caracteres, le indica al Curiosity como debe moverse sobre el cielo marciano. Puede ser avanzar o girar.
- Magnitud, de tipo decimal, indica el valor o cantidad del movimiento

- Unidad_medida, entero, es la unidad con la que se hace la medición del movimiento

Operaciones:

- ObtenerTipo_movimiento(), retorna el tipo de movimiento actual que tiene el comando
- ObtenerMagnitud(), retorna el valor o cantidad de movimiento actual que indica el comando
- ObtenerUnidad_medida(), retorna la unidad actual con la que se hizo la medición del movimiento
- FijarTipo_movimiento(), fija un nuevo valor para el tipo de movimiento que indica el comando
- FijarMagnitud(), fija un nuevo valor o cantidad de movimiento para el comando
- FijarUnidad_medida(), fija una nueva unidad con la que se hace la medición del movimiento

d. Análisis

Datos mínimos:

- Tipo_analisis, cadena de caracteres, le indica al Curiosity como debe analizar un elemento. Puede ser fotografiar, composición o perforar
- Objeto, cadena de caracteres, es el nombre del elemento sobre el cual se hace el análisis
- Comentario, cadena de caracteres, corresponde a información adicional sobre el análisis

Operaciones:

- ObtenerTipo_analisis(), retorna el tipo de análisis actual que debe realizar el Curiosity a un elemento de acuerdo al comando
- ObtenerObjeto(), retorna el nombre actual del elemento sobre el cual se hace el análisis
- ObtenerComentario(), retorna la información adicional actual sobre el análisis
- FijarTipo_analisis(), fija un nuevo tipo de análisis que debe realizar el Curiosity a un elemento
- FijarObjeto(), fija un nuevo tipo de elemento sobre el cual se debe hacer el análisis
- FijarComentario(), fija nueva información adicional para el análisis de un elemento

3.2. Descripción de entradas, salidas y condiciones

Procedimiento principal:

Entradas	Comando ayuda tipo string, lista de todos los comandos de movimiento que se desean realizar tipo list<movimiento>,lista de comandos de análisis que se desean realizar tipo list<analisis>, lista de los elementos de tipo list<elemento>
Salidas	Si se ingresa comando erróneo: Comando no reconocido.

	Si se ingresa comando correcto: Ejecuta comando.
Condiciones	El comando debe existir, sus parametros deben ser aquellos aceptados por el mismo. El número de parametros debe coincidir con los requeridos por el comando.

Comando “cargar_comandos”

Entradas	Comando “cargar_comandos” tipo string, nombre correspondiente del archivo (nombre_archivo)(string)
Salidas	(Archivo vacio) nombre_archivo no contiene comandos (Archivo erróneo) nombre_archivo no se encuentra o no puede leerse (Resultado exitoso) n comandos cargados correctamente desde nombre_archivo
Condiciones	El archivo debe ser válido, debe poder leerse

Comando “cargar_elementos”

Entradas	Comando “cargar_elementos” tipo string, nombre correspondiente del archivo (nombre_archivo)(string)
Salidas	(Archivo vacio) nombre_archivo no contiene comandos (Archivo erróneo) nombre_archivo no se encuentra o no puede leerse (Resultado exitoso) n elementos cargados correctamente desde nombre_archivo
Condiciones	El archivo debe ser válido, debe poder leerse

Comando “agregar_movimiento”

Entradas	Comando “agregar_movimiento” tipo string, dato “tipo_movimiento” de tipo string, dato “magnitud” tipo float, dato “unidad_medida” tipo string

Salidas	(Formato erróneo) La información del movimiento no corresponde a los datos esperados (tipo(string), magnitud(float), unidad(string)). (Resultado exitoso) El comando de movimiento ha sido agregado correctamente
Condiciones	Se debe cumplir con el formato del comando

Comando “agregar_analisis”

Entradas	Comando “agregar_analisis” tipo string, dato “tipo_analisis” de tipo string, dato “objeto” tipo string, dato “comentario” tipo string
Salidas	(Formato erróneo) La información del movimiento no corresponde a los datos esperados (tipo (string), objeto (string), comentario (string)). (Resultado exitoso) El comando de analisis ha sido agregado correctamente
Condiciones	Se debe cumplir con el formato del comando

Comando “agregar_elemento”

Entradas	Comando “agregar_elemento” tipo string, dato “tipo_elemento” de tipo string, dato “tamaño” tipo int, dato “unidad_medida” tipo string, dato “coordenada_x” tipo double, dato “coordenada_y” tipo double
Salidas	(Formato erróneo) La información del movimiento no corresponde a los datos esperados (tipo (string), tamaño (int), unidad (string), x (double), y (double) (Resultado exitoso) El elemento ha sido agregado correctamente
Condiciones	Se debe cumplir con el formato del comando

Comando “guardar”

Entradas	Comando guardar tipo string, dato “tipo_archivo” tipo string, dato “nombre_archivo” tipo string

Salidas	(No hay información) La información requerida no está almacenada en memoria. (Escritura exitosa) La información ha sido guardada en nombre_archivo . (Problemas en archivo) Error guardando en nombre_archivo .
Condiciones	Se debe cumplir con el formato del comando

Comando “simular_comandos”

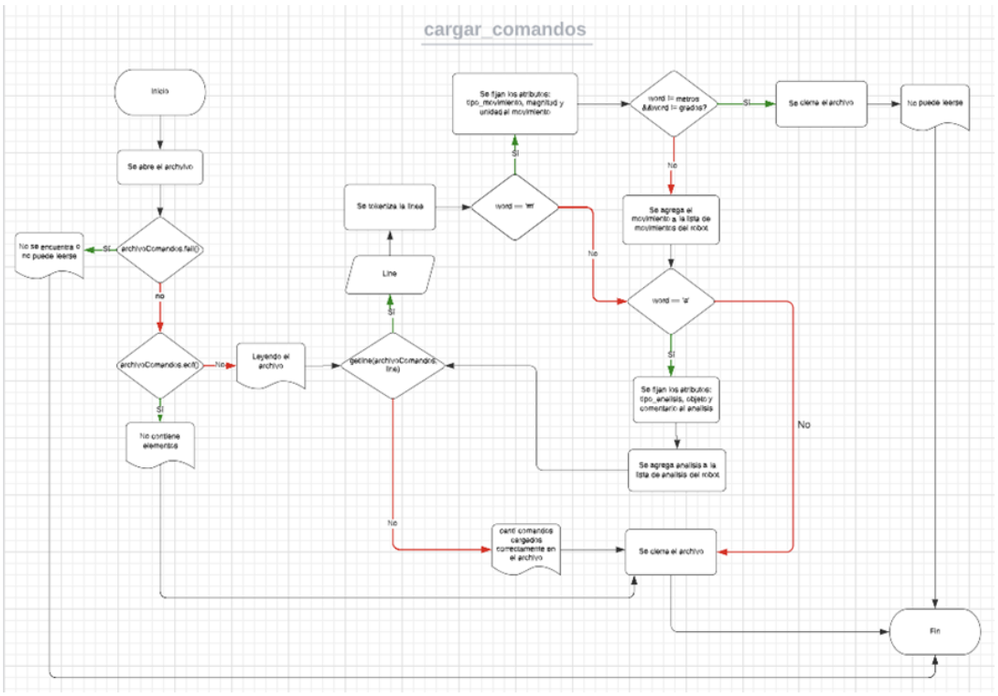
Entradas	Comando simular_comandos” tipo string, dato “coordenada_x” tipo double, dato “coordenada_y” tipo double
Salidas	(No hay información) La información requerida no está almacenada en memoria. (Resultado exitoso) La simulación de los comandos, a partir de la posición (coordX,coordY), deja al robot en la nueva posición (nuevoX ,nuevoY).
Condiciones	Se debe cumplir con el formato del comando

Comando “salir”

Entradas	Comando salir tipo string
Salidas	Sin salida por pantalla
Condiciones	No debe recibir ningún otro parámetro más allá del comando base

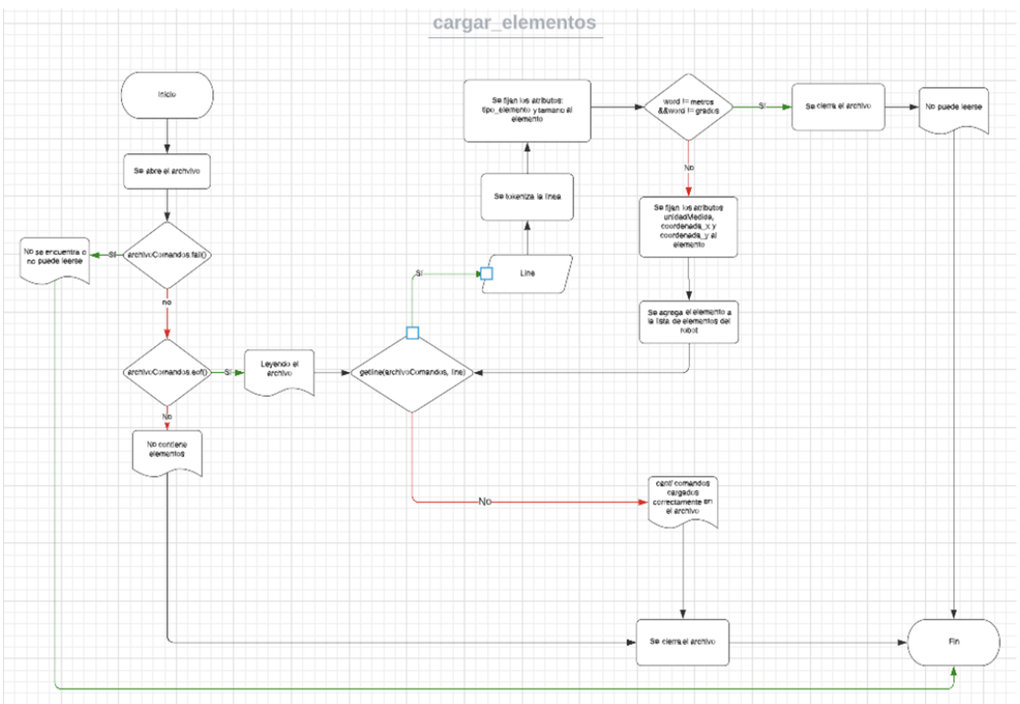
3.3. Diagramas de Flujo

a. Función “cargar_comandos”



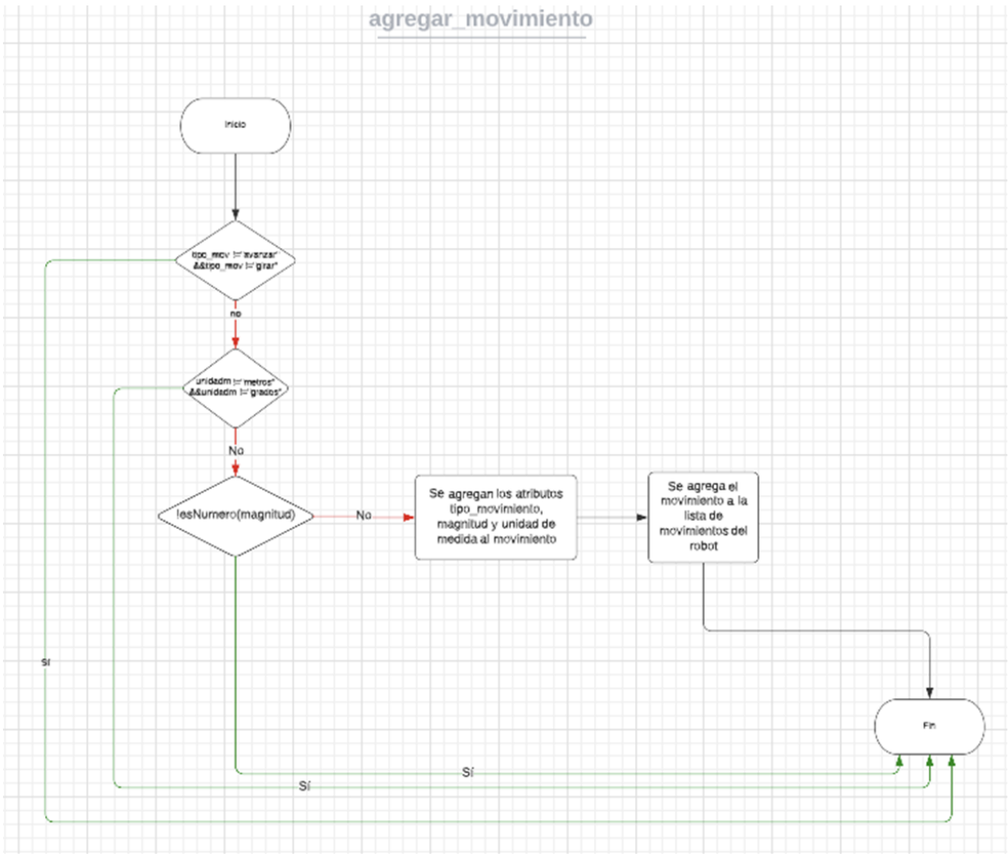
Link del diagrama:
https://lucid.app/lucidchart/7e608231-e1fe-4892-a0d0-034e46678766/edit?viewport_loc=-408%2C3%2C2911%2C1383%2C0_0&invitationId=inv_76e2b1b7-a293-4048-8560-cd0e85ec33bf

b. Función “cargar_elementos”



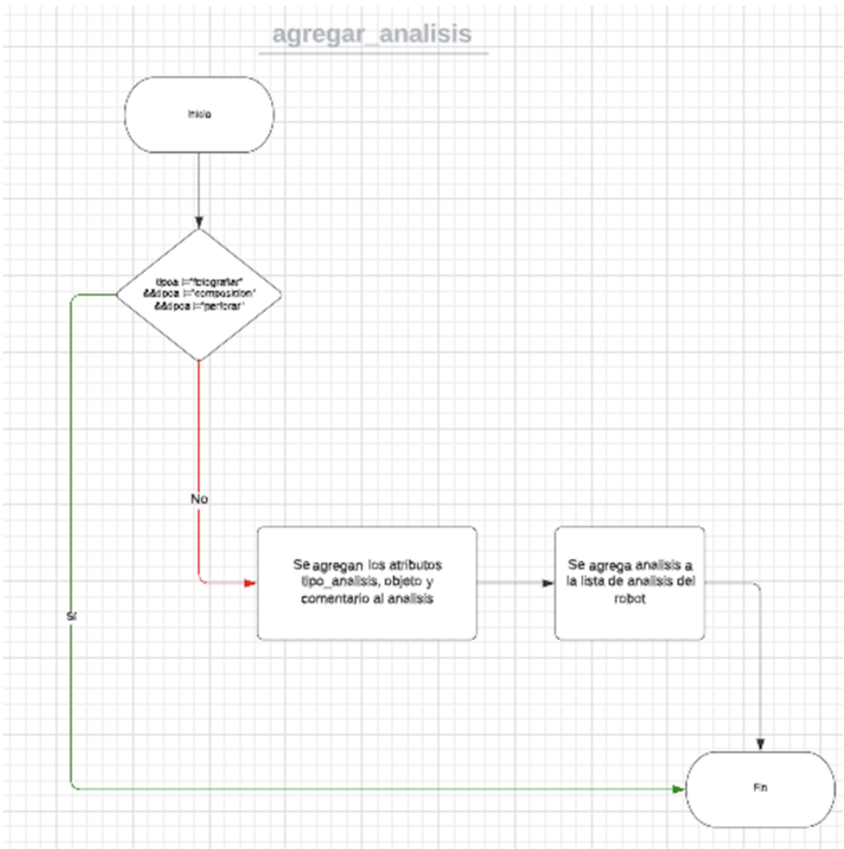
Link del diagrama:
https://lucid.app/lucidchart/7e608231-e1fe-4892-a0d0-034e46678766/edit?viewport_loc=-534%2C29%2C2826%2C1342%2CGUZCwfcK4cHs&invitationId=inv_76e2b1b7-a293-4048-8560-cd0e85ec33bf

c. Función “agregar_movimiento”



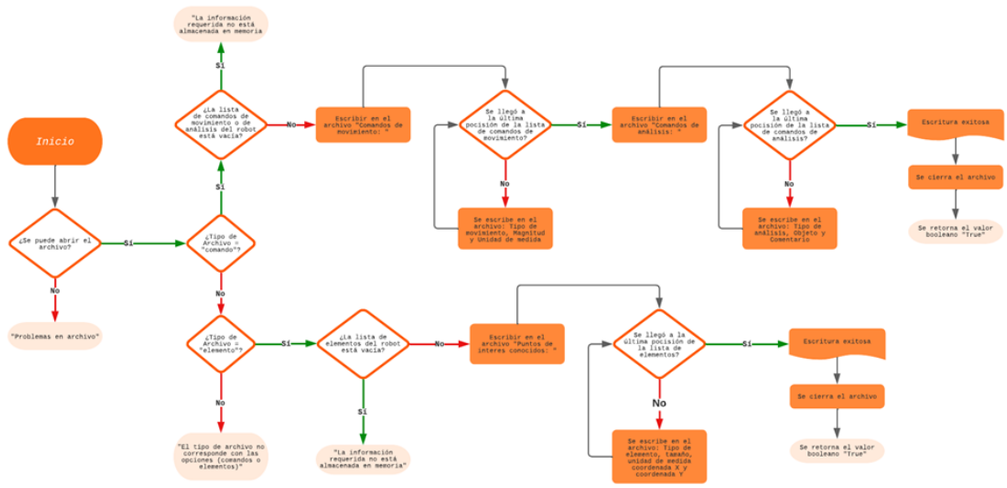
Link del diagrama:
https://lucid.app/lucidchart/7e608231-e1fe-4892-a0d0-034e46678766/edit?viewport_loc=-758%2C-97%2C3203%2C1521%2C06ZCH9JY5YVP&invitationId=inv_76e2b1b7-a293-4048-8560-cd0e85ec33bf

d. Función “agregar_analisis”

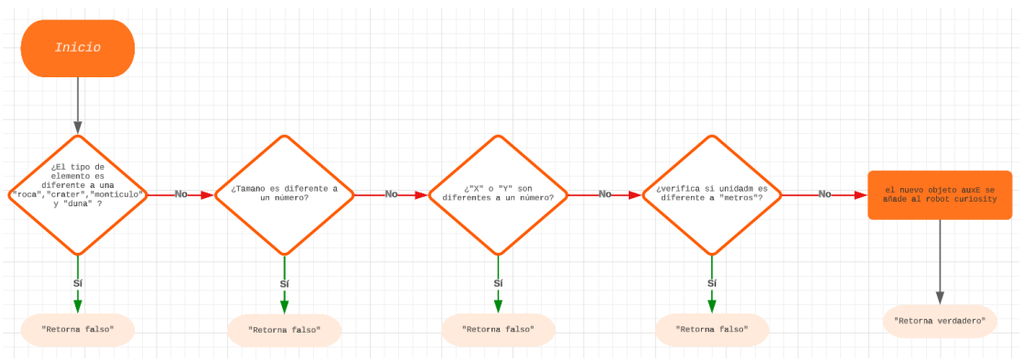


Link del diagrama:
https://lucid.app/lucidchart/7e608231-e1fe-4892-a0d0-034e46678766/edit?viewport_loc=-547%2C37%2C3347%2C1590%2Cwk0Cq4n9H5rG&invitationId=inv_76e2b1b7-a293-4048-8560-cd0e85ec33bf

e. Función “Guardar”

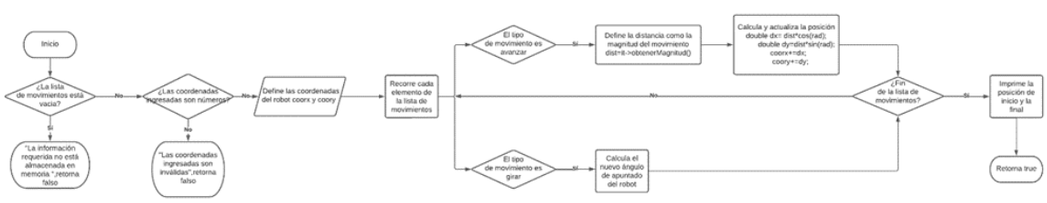


f. Función “Agregar Elemento”



Link del diagrama:
https://lucid.app/lucidchart/024d3e17-1925-43a7-8e4d-f52b0aab86ea/edit?viewport_loc=33%2C-116%2C1707%2C811%2C0_0&invitationId=inv_a477e5d4-7f4c-400c-8750-41034c425583

g. Función “Simular Comandos”



3.4. Plan de pruebas comando “simular_comandos”

Se proponen los siguientes datos para probar el correcto funcionamiento del comando:

Comandos de movimientos:

- avanzar 4 metros
- girar 360 grados
- avanzar 3 metros
- girar 180 grados
- avanzar 2 metros

Plan de pruebas comando “simular_comandos”			
Descripción de caso	Valores de entrada	Resultado esperado	Resultado obtenido
1.Posición inicial	X=0 Y=0	(3,6)	(3,6)
2.Posición obtenida de la inicial	X=3 Y=6	(6,12)	(6,12)
3.Datos no válidos	X=a Y=C	Datos no válidos	Datos no válidos
4.Un dato válido y uno inválido	X=1 Y=C	Datos no válidos	Datos no válidos

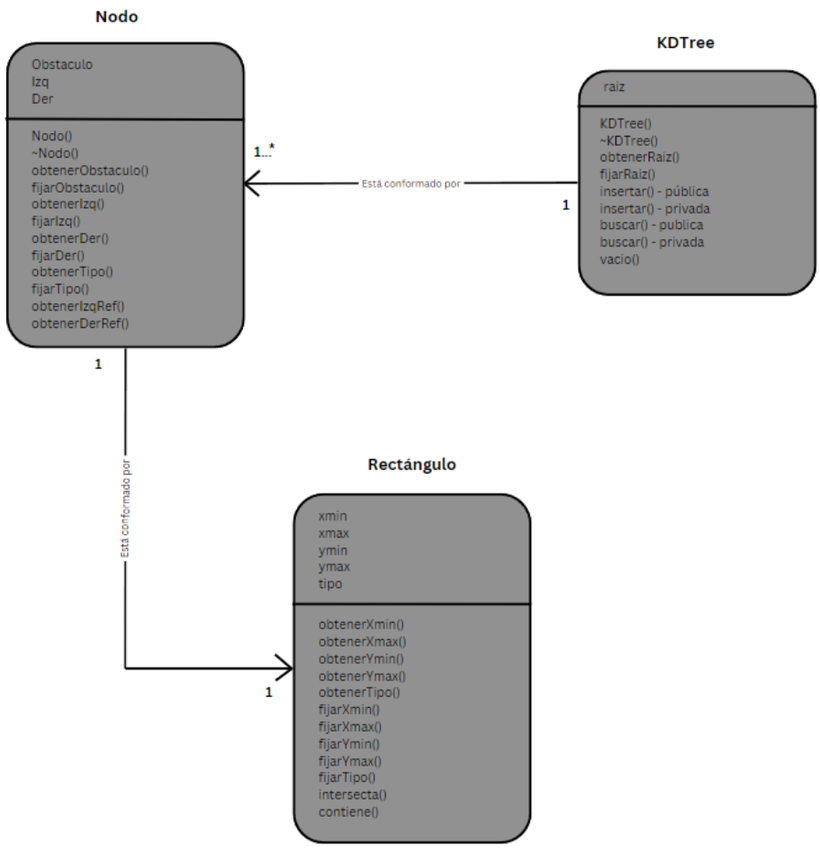
4. Desarrollo componente 2

Objetivo: Utilizar una estructura de datos jerárquica que permita almacenar datos geométricos para analizar los puntos de interés sobre el cielo marciano y así facilitar en el futuro la planificación automática de desplazamientos. Los comandos que se deben implementar como parte de este componente son:

- ubicar_elementos
- en_cuadrante coordX1 coordX2 coordY1 coordY2

4.1. Declaración de TAD's

Para el desarrollo del segundo componente del proyecto se definieron los siguientes TAD's:



a. Rectángulo

Datos mínimos

- xmin, entero, representa el valor mínimo que toma el rectángulo en el eje X del plano cartesiano
- xmax, entero, representa el valor máximo que toma el rectángulo en el eje X del plano cartesiano
- ymin, entero, representa el valor mínimo que toma el rectángulo en el eje Y del plano cartesiano
- ymax, entero, representa el valor máximo que toma el rectángulo en el eje Y del plano cartesiano
- tipo, cadena de caracteres, contiene el nombre del tipo de elemento a partir del cuál se va a crear el rectángulo, puede ser roca, cráter, montículo, duna o cuadrante

Operaciones

- fijarXmin(), fija un nueva coordenada mínima en el eje X para el rectángulo
- fijarXmax(), fija un nueva coordenada máxima en el eje X para el rectángulo
- fijarYmin(), fija un nueva coordenada mínima en el eje Y para el rectángulo
- fijarYmax(), fija un nueva coordenada máxima en el eje Y para el rectángulo
- fijarTipo(), fija un nuevo tipo a partir del cuál se crea el rectángulo
- obtenerXmin(), retorna la coordenada mínima actual del rectángulo en el eje X
- obtenerXmax(), retorna la coordenada máxima actual del rectángulo en el eje X
- obtenerYmin(), retorna la coordenada mínima actual del rectángulo en el eje Y
- obtenerYmax(), retorna la coordenada máxima actual del rectángulo en el eje Y
- obtenerTipo(), retorna el tipo de elemento a partir del cuál se creó el rectángulo
- intersecta(), evalúa si el rectángulo se intersecta con un rectángulo que recibe como parámetro. En caso afirmativo retorna True, de lo contrario retorna False.
- contiene(), evalúa si el rectángulo se encuentra contenido dentro de un rectángulo que recibe como parámetro. En caso afirmativo retorna True, de lo contrario retorna False.

b. Nodo

Datos mínimos

- Obstáculo, apuntador a un rectángulo, representa el elemento u obstáculo que se encuentra ubicado en el nodo
- Izq, apuntador a un nodo, representa el hijo izquierdo del nodo
- Der, apuntador a un nodo, representa el hijo derecho del nodo

Operaciones

- Nodo(), constructor, inicializa los valores del nodo asignando un apuntador a rectángulo que recibe como parámetro para el atributo “obstáculo”, y NULL para los atributos “izq” y “der”
- ~Nodo(), destructor, elimina los apuntadores “obstáculo”, “izq” y “der”, que corresponden a los atributos del nodo
- fijarObstaculo(), fija un nuevo valor para el elemento u obstáculo que se encuentra ubicado en el nodo
- fijarIzq(), fija un nuevo valor para el hijo izquierdo del nodo
- fijarDer(), fija un nuevo valor para el hijo derecho del nodo
- obtenerObstaculo(), retorna el apuntador que corresponde al elemento u obstáculo que se encuentra ubicado en el nodo actualmente
- obtenerIzq(), retorna el apuntador que corresponde al nodo hijo izquierdo del nodo actual
- ObtenerDer(), retorna el apuntador que corresponde al nodo hijo derecho del nodo actual
- ObtenerIzqRef(), retorna la referencia en memoria que corresponde al apuntador al nodo hijo izquierdo del nodo actual
- ObtenerDerRef(), retorna la referencia en memoria que corresponde al apuntador al nodo hijo derecho del nodo actual

c. KDTree

Datos mínimos

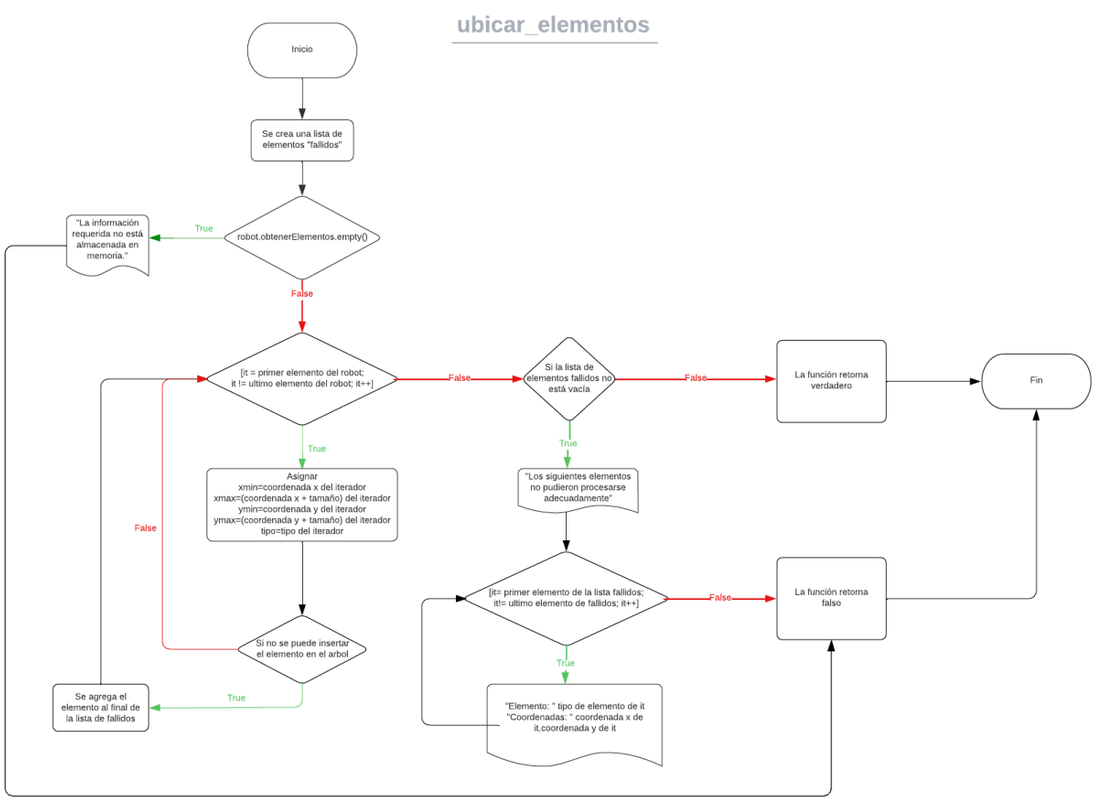
- Raíz, apuntador de tipo Nodo, representa el nodo a partir del cual se crear el árbol KDTree

Operaciones

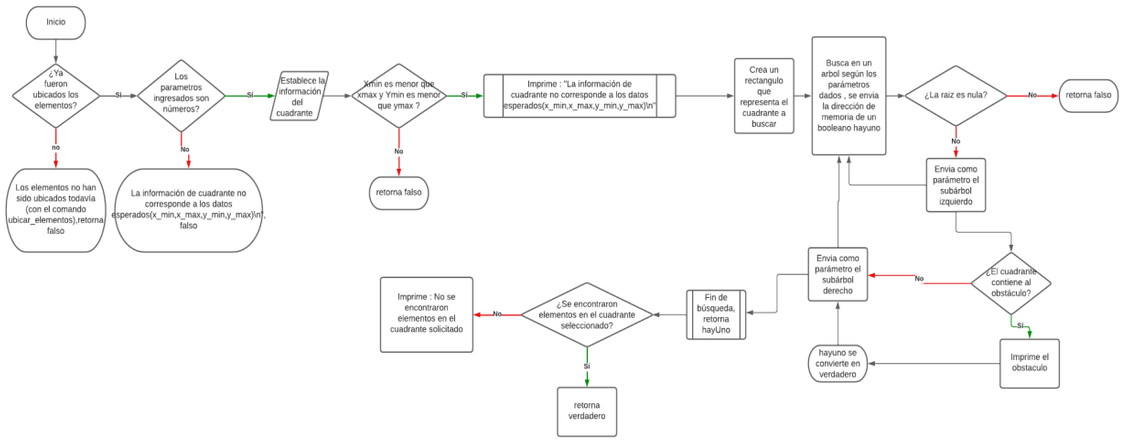
- KDTree(), constructor, inicializa la raíz del árbol KDTree en NULL
- ~KDTree(), destructor, elimina el apuntador que corresponde a la raíz del árbol KDTree
- fijarRaiz(), fija un nuevo valor para la raíz del árbol KDTree
- obtenerRaiz(), retorna el apuntador a la raíz actual del árbol KDTree
- insertar(), función pública que utiliza la función privada "insertar" con el fin de agregar un nuevo nodo en el árbol
- insertar(), función privada que evalúa si es posible insertar un nuevo nodo en el árbol de acuerdo a sus coordenadas. En caso positivo inserta un nuevo nodo en el árbol KDTree y retorna True. En caso negativo retorna False.
- buscar(), función pública que utiliza la función privada "buscar" para recorrer el árbol KDTree y evaluar si un elemento dado de tipo rectángulo se encuentra intersectado con otro elemento dado de tipo rectángulo.
- buscar(), función privada que recorre el árbol KDTree y evalúa si un elemento dado de tipo rectángulo se encuentra intersectado con otro elemento dado de tipo rectángulo. En caso afirmativo este elemento se imprime.
- vacio(), evalúa si la raíz del árbol KDTree es nula, con el fin de determinar si el árbol está vacío

4.2. Diagramas de Flujo

a. Ubicar elementos



b. En_cuadrante



4.3. Plan de pruebas

Con el fin de realizar el plan de pruebas de las nuevas funciones implementadas en el proyecto (“ubicar_elementos” y “en_cuadrante”), se cargarán 5 elementos del archivo de prueba ‘elementosp.txt’ para verificar el correcto funcionamiento del código. Cabe resaltar que los 5 elementos son insertados utilizando el siguiente formato:

Tipo de elemento|Tamaño|Distancia|Centro_x|Centro_y

A continuación se listan los elementos que se encuentran el archivo “elementosp.txt”

- crater|2|metros|2.0|2.0
- crater|2|metros|3.0|3.0
- monticulo|2|metros|6.0|8.0
- crater|5|metros|12.50|7.5
- duna|4|metros|18.00|14.00

A continuación se listan los cuadrantes que se utilizarán en conjunto con el comando “en_cuadrante”:.

- en_cuadrante 0 100 0 100
- en_cuadrante 6 18 2 14
- en_cuadrante 0 5 0 4
- en_cuadrante 50 60 50 60

Finalmente, se presenta el plan de pruebas para cada uno de los cuadrantes descritos anteriormente:

Plan de pruebas			
Descripción de caso	**Valores de entrada**	**Resultado esperado**	**Resultado obtenido**
Ingresar cuadrante	xmim= 0 xmax=100 ymin=0 ymax=100	Elemento: Cráter Coordenadas: (2,2),(4,4) Elemento: Cráter Coordenadas: (12,7),(17,12) Elemento: Montículo Coordenadas: (6,8),(8,10) Elemento:duna Coordenadas:	Elemento: Cráter Coordenadas: (2,2),(4,4) Elemento: Cráter Coordenadas: (12,7),(17,12) Elemento: Montículo Coordenadas: (6,8),(8,10) Elemento:duna Coordenadas:

		(18,14),(22,18)	(18,14),(22,18)

Como se puede apreciar en la siguiente imagen, el elemento cráter con coordenada (3,3) no se pudo guardar dado que se intersecta con el elemento cráter con coordenada (2,2)

```
PS C:\Users\valeo\Downloads\EntregaCuriosity\ProyectoEntrega2> g++ -std=c++11 -o prog *.cpp
PS C:\Users\valeo\Downloads\EntregaCuriosity\ProyectoEntrega2> ./prog
-$ cargar_elementos elementosp.txt
Leyendo archivo 'elementosp.txt':
5 elementos cargados correctamente desde: elementosp.txt
-$ ubicar_elementos
Los siguientes elementos no pudieron procesarse adecuadamente:
Elemento: crater Coordenadas: (3,3)
-$ en_cuadrante 0 100 0 100
Los elementos ubicados en el cuadrante solicitado son:
Elemento:crater
Coordenadas: (2,2),(4,4)
Elemento:crater
Coordenadas: (12,7),(17,12)
Elemento:monticulo
Coordenadas: (6,8),(8,10)
Elemento:duna
Coordenadas: (18,14),(22,18)
```

Plan de pruebas			
Descripción de caso	**Valores de entrada**	**Resultado esperado**	**Resultado obtenido**
1.Ingresar cuadrante	xmim= 6 xmax=18 ymin=2 ymax=14	Elemento: Cráter Coordenadas: (12,7),(17,12) Elemento: Montículo Coordenadas: (6,8),(8,10) Elemento:duna Coordenadas: (18,14),(22,18)	Elemento: Cráter Coordenadas: (12,7),(17,12) Elemento: Montículo Coordenadas: (6,8),(8,10) Elemento:duna Coordenadas: (18,14),(22,18)

A continuación se presenta el resultado de la prueba al ejecutar el programa:

```
-$ en_cuadrante 6 18 2 14
Los elementos ubicados en el cuadrante solicitado son:
Elemento:crater
Coordenadas: (12,7),(17,12)
Elemento:monticulo
Coordenadas: (6,8),(8,10)
Elemento:duna
Coordenadas: (18,14),(22,18)
```

Plan de pruebas			
Descripción de caso	**Valores de entrada**	**Resultado esperado**	**Resultado obtenido**
1.Ingresar cuadrante	xmim= 0 xmax=5 ymin=0	Elemento: Cráter Coordenadas: (2,2),(4,4)	Elemento: Cráter Coordenadas: (2,2),(4,4)

	ymax=4		

A continuación se presenta el resultado de la prueba al ejecutar el programa:

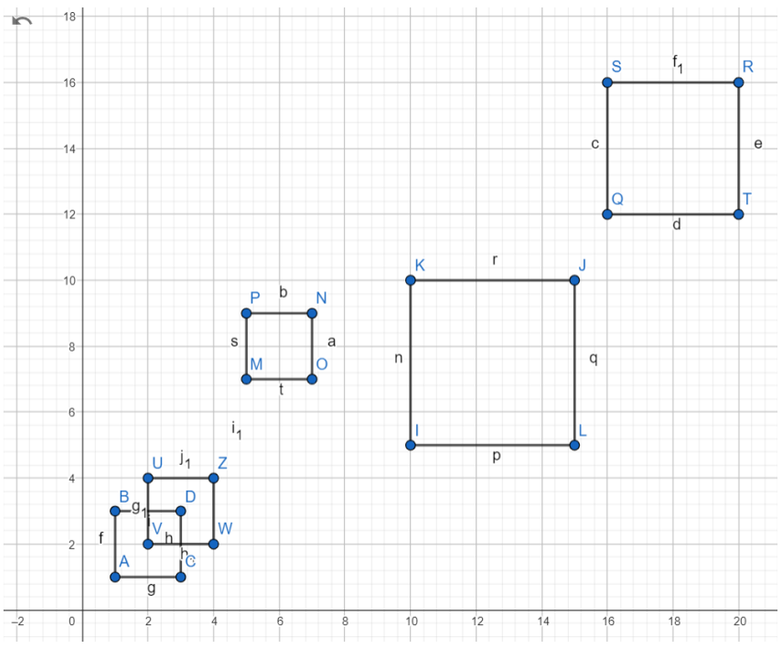
```
-$ en_cuadrante 0 5 0 4
Los elementos ubicados en el cuadrante solicitado son:
Elemento:crater
Coordenadas: (2,2),(4,4)
```

Plan de pruebas			
Descripción de caso	**Valores de entrada**	**Resultado esperado**	**Resultado obtenido**
1.Ingresar cuadrante	xmim= 50 xmax=60 ymin=50 ymax=60	*No se encontraron elementos en el cuadrante solicitado	*No se encontraron elementos en el cuadrante solicitado

A continuación se presenta el resultado de la prueba al ejecutar el programa:

```
-$ en_cuadrante 50 60 50 60
Los elementos ubicados en el cuadrante solicitado son:
*No se encontraron elementos en el cuadrante solicitado*
```

A continuación se adjunta una imagen de los elementos que fueron ingresados al programa, la cual sirvió de apoyo visual para así obtener una mejor idea la representación de los obstáculos en el plano.



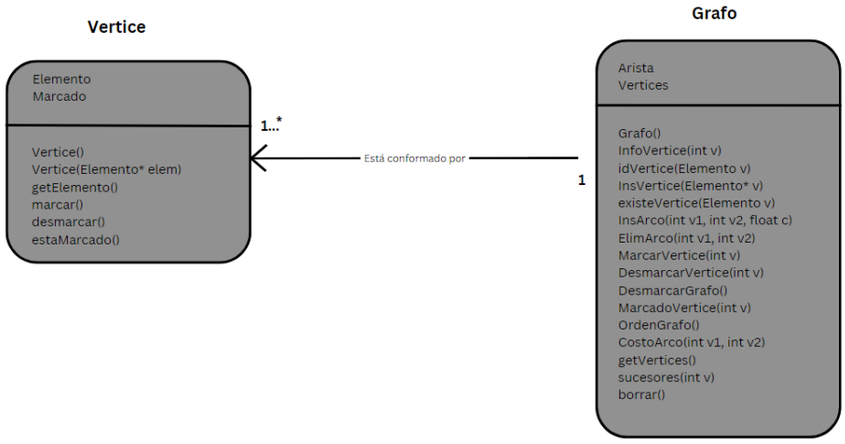
5. Desarrollo componente 3

Objetivo: A partir de la información de puntos de interés, utilizar representaciones en grafos para crear representaciones geográficas (mapas) que permitan posteriormente identificar regiones de interés sobre el suelo marciano para aterrizajes y exploración. Los comandos que se deben desarrollar como parte de este componente son:

- crear_mapa coeficiente_conectividad
- ruta_mas_larga

5.1. Declaración de TAD's

Para el desarrollo del segundo componente del proyecto se definieron los siguientes TAD's:



a. Vértice

Datos mínimos

- elemento, apuntador de tipo "elemento", que corresponde al elemento que contiene el vértice del grafo
- marcado, booleano que indica si el vértice se encuentra marcado o no

Operaciones

- Vertice(), constructor por defecto del vértice, no realiza ninguna operación
- Vertice(Elemento* elem), constructor sobrecargado que recibe como parámetro un elemento y lo asigna al atributo "elemento" del vértice.
- getElemento(), retorna el atributo de tipo elemento que se encuentra almacenado en el vértice
- marcar(), asigna un valor de verdadero para el atributo "marcado" del vértice
- desmarcar(), asigna un valor de faso para el atributo "marcado" del vértice
- estaMarcado(), retorna el atributo "marcado" del vértice

b. Grafo

Datos mínimos

- Arista, lista de tipo "pair" (de dos elementos) que contiene en la primera posición el coste o distancia del vértice actual con su vértice vecino, y en la segunda posición el id del vértice vecino al cual corresponde la distancia antes mencionada

- vertices, lista de vértices que contiene cada uno de los vértices del grafo

Operaciones

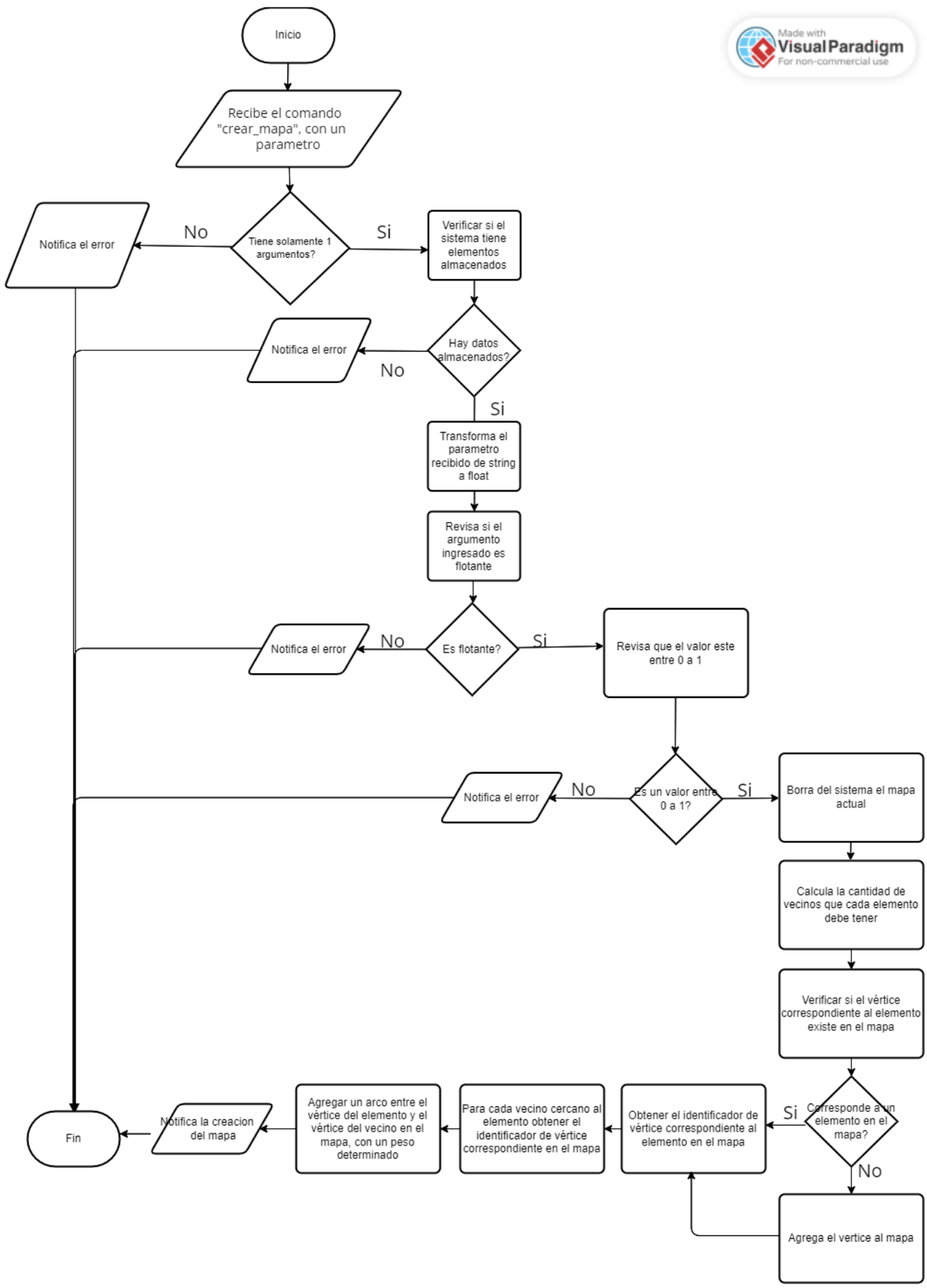
- Grafo(), constructor por defecto del grafo, no realiza ninguna operación
- InfoVertice(int v), recibe un entero que corresponde al ID de un vértice, lo busca en la lista de vértices, y una vez lo encuentra retorna el atributo de tipo “elemento” que se encuentra almacenado actualmente en el vértice
- idVertice(Elemento v), recibe un parámetro de tipo “Elemento” y busca en la lista de vértices el vértice que contenga este elemento con la finalidad de retornar su ID
- InsVertice(Elemento* v), crea un nuevo vértice a partir del parámetro de tipo “elemento” que recibe, es decir, llama al constructor sobrecargado del vértice. Una vez el vétice está creado lo inserta en la lista de vértices
- existeVertice(Elemento v), recibe un parámetro de tipo “Elemento” y busca en la lista de vértices para comprobar si ya existe alguno que contenga este elemento, en caso de que sí se retorna un valo de “verdadero”, en caso contrario se retorna un valor de “falso”
- InsArco(int v1, int v2, float c), crea un nuevo elemento de tipo “Arista” buscando en la lista de aristas hasta encontrar la posición que corresponde al parámetro “v1”, y allí inserta el parámetro “v2” en la primera posición y “c” en la segunda, siendo respectivamente el vértice de destino y la distancia del vértice de origen al de destino.
- ElimArco(int v1, int v2), elimina la arista que conecta al vértice con ID “v1” y el vértice con ID “v2”
- MarcarVertice(int v), llama a la función “marcar()” del vértice que posea el ID que recibe como parámetro con el nombre de “v”, con la finalidad de que su atributo “marcado” ahora tenga un valor de verdadero
- DesmarcarVertice(int v) llama a la función “desmarcar()” del vértice y se utiliza para desmarcar un vértice específico en el grafo.
- DesmarcarGrafo(), se utiliza para desmarcar todos los vértices del grafo.
- MarcadoVertice(int v), es un método de la clase Grafo que se utiliza para verificar si un vértice específico está marcado en el grafo.
- OrdenGrafo(), es un método constante de la clase Grafo que devuelve el número de vértices en el grafo.
- CostoArco(int v1, int v2), se utiliza para determinar el costo de un arco entre dos vértices dentro del grafo.
- getVertices(), devuelve una lista de punteros a elementos (Elemento*), representando los elementos asociados a cada vértice del grafo.
- sucesores(int v), devuelve una lista de enteros que representa los sucesores de un vértice específico en el grafo.
- borrar(), se encarga de eliminar todas las aristas y vértices existentes en el grafo, dejándolo vacío.

5.2. Diagramas de Flujo

a. Crear_mapa

Nombre:	comando **crear_mapa**
Objetivo en Contexto	El objetivo de este comando es utilizar la información de puntos de interés almacenada en memoria para ubicarlos en un grafo teniendo en cuenta el coeficiente de conectividad dado. Se busca que cada elemento esté conectado a los demás elementos más cercanos a él, utilizando la distancia euclidiana como medida de cercanía y peso de la conexión. El comando tiene como objetivo generar un mapa donde cada elemento tenga n vecinos, determinado por el coeficiente de conectividad especificado.
ENTRADAS	Se recibe el comando de crear_mapa y el coeficiente de conectividad que se usara
Pre-Condiciones *(Escenario de éxito)*	El comando “crear_mapa” debe tener exactamente dos palabras, donde la primera debe ser “crear_mapa” y la segunda debe ser el valor del coeficiente de conectividad.
SALIDAS	<u>Si el comando es correcto:</u> Si hay elementos cargados, y se ingresaron los datos de forma correcta, la funcion crearia el mapa e imprimiria”El mapa se ha generado exitosamente. Cada elemento tiene " << vecinos << " vecinos.” <u>Si el comando es incorrecto(Se le pasa un numero diferente de un parametro):</u> Se termina el programa y se muestra en pantalla: “.Se requiere el coeficiente conectividad" <u>Si no hay elementos:</u> Se termina el programa y se muestra en pantalla: “La informacion requerida no esta almacenada en memoria." <u>Error con el coeficiente de conectividad:</u> Sucede cuando el coeficiente de conectividad no es un valor entre 0 y 1, y imprime:”El coeficiente de conectividad debe tener un valor entre 0 y 1.” <u>Si el argumento no es flotante:</u> Se imprime en pantalla “El coeficiente de conectividad debe tener un valor entre 0 y 1”
Post-Condiciones	Se imprime que la funcion se ejecuto correctamente, y se crea el mapa de elementos

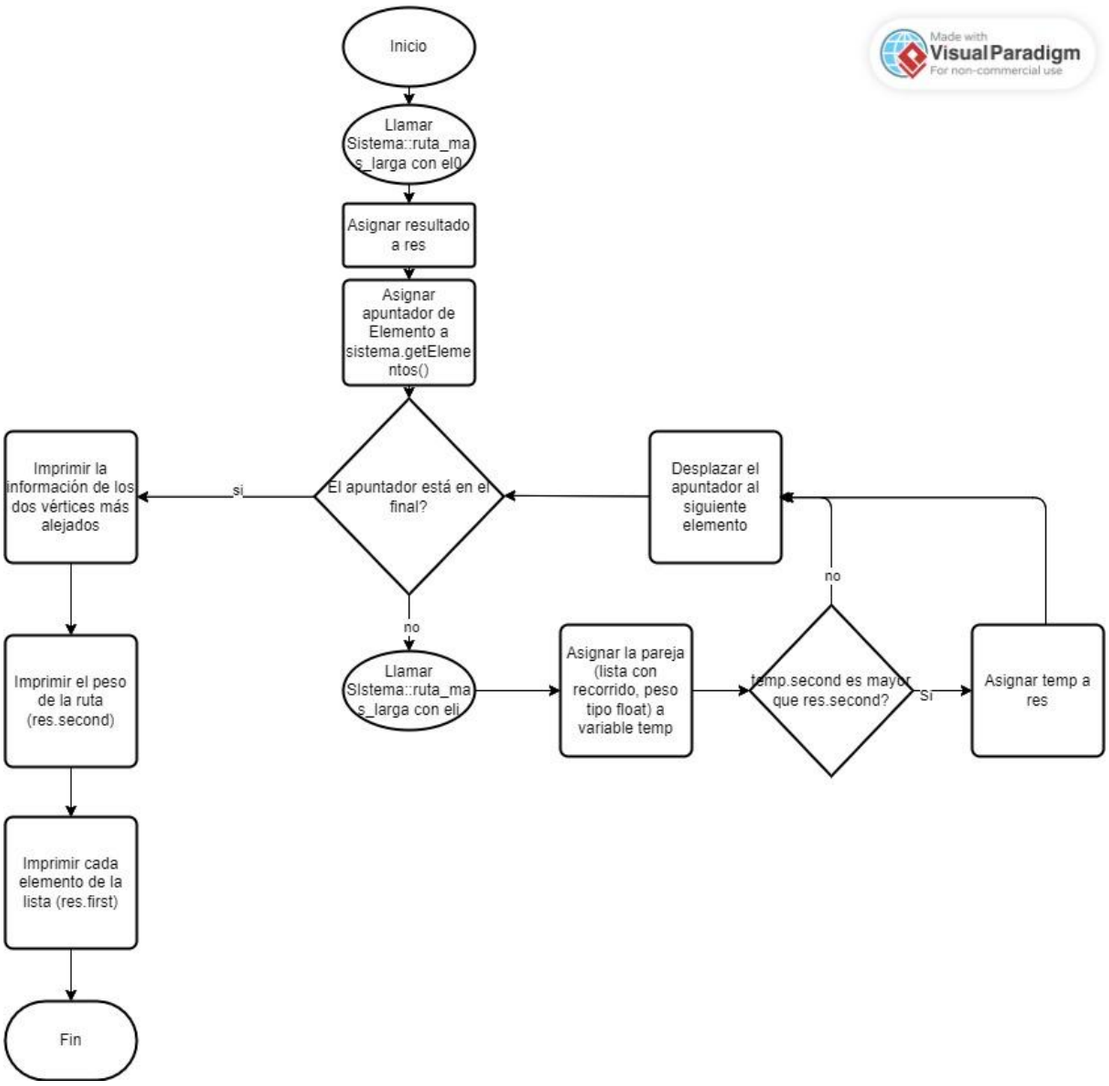
(Escenario de éxito)	



b. Ruta_mas_larga

Nombre:	comando **ruta_mas_larga**
Objetivo en Contexto	El objetivo de este comando es encontrar la ruta más larga existente dentro de un grafo. Esta, siendo la ruta entre dos vértices, que pase por otros k vértices y que el peso de esas k+1 aristas, sea el mayor posible.
ENTRADAS	Se recibe el comando de ruta_mas_larga
Pre-Condiciones *(Escenario de éxito)*	El comando "ruta_mas_larga" requiere que el comando "crear_mapa" haya sido ejecutado con éxito.
SALIDAS	<u>Si el comando es correcto:</u> En una situación adecuada, el programa imprime los siguientes resultados en consola: "Los puntos de interés más alejados entre si son: <<punto 1>> y <<punto 2>>." "La ruta que los conecta tiene una longitud total de <<longitud>> y pasa por los siguientes elementos:" "<<Lista de elementos que recorre>>"" <u>Si el comando recibe algún parámetro:</u> Si el comando es recibido con algún parámetro extra (diferente del nombre del comando" el programa arroja una excepción de tiempo de ejecución e imprime: "No se requieren argumentos" <u>Si el comando "crear_mapa" no ha sido ejecutado con éxito:</u> Si el programa detecta que el mapa está vacío, por ende no ha sido ejecutado el comando crear_mapa, arroja una excepción de tiempo de ejecución e imprime: "El mapa no ha sido generado todavía (con el comando crear_mapa)" <u>Si algún elemento recibido no existe en el mapa:</u> Si el programa detecta que alguno de los vértices no está incluído en el mapa, arroja una excepción de tiempo de ejecución e imprime: "El elemento no se encuentra en el mapa"

Post-Condiciones *(Escenario de éxito)*	Se imprime: “Los puntos de interés más alejados entre si son: <<punto 1>> y <<punto 2>>.” “La ruta que los conecta tiene una longitud total de <<longitud>> y pasa por los siguientes elementos:” “<<Lista de elementos que recorre>>”” No se retorna nada ni se altera espacio en memoria



5.3. Plan de pruebas

Con el fin de realizar el plan de pruebas de las nuevas funciones implementadas en el proyecto (“crear_mapa” y “ruta_mas_larga”), se cargarán 5 elementos del archivo de prueba ‘elementosp.txt’ para verificar el correcto funcionamiento del código. Cabe resaltar que los 5 elementos son insertados usando el siguiente formato:

Tipo de elemento|Tamaño|Distancia|CentroX|CentroY

A continuación se listan los elementos que se encuentran en el archivo “elementosp.txt”:

- crater|1|metros|2.0|2.0
- crater|1|metros|3.0|3.0
- monticulo|1|metros|6.0|8.0
- crater|2.5|metros|12.50|7.5
- duna|2|metros|18.00|14.00

A continuación se listan los coeficientes que se utilizarán en conjunto con el comando “crear_mapa”:

- crear_mapa 0.1
- crear_mapa 1
- crear_mapa -1
- crear_mapa 0.4

A continuación se presenta la tabla con las pruebas realizadas y su respectivo resultado esperado:

Descripción del caso	**Valores de entrada**	**Resultado Esperado**	**Resultado Obtenido**
Sin crear Mapa	ninguno	*El mapa no ha sido generado todavía (con el comando crear_mapa)	*El mapa no ha sido generado todavía (con el comando crear_mapa)
Coeficiente 0.1	0.1	Los puntos de interes mas alejados entre si son: “duna 2 metros 18 14” y “crater 1 metros 2 2”	Los puntos de interes mas alejados entre si son: “duna 2 metros 18 14” y “crater 1 metros 2 2”
Coeficiente 1	1	Los puntos de interes mas alejados entre si son: “crater 2 metros 12 7” y “crater 1 metros 3 3”	Los puntos de interes mas alejados entre si son: “crater 2 metros 12 7” y “crater 1 metros 3 3”
Coeficiente -1	-1	*El coeficiente de conectividad debe tener un valor entre 0 y 1	*El coeficiente de conectividad debe tener un valor entre 0 y 1
Coeficiente 0.4	0.4	Los puntos de interes mas alejados entre si son: “crater 1 metros 3 3” y “duna 2 metros 18 14”	Los puntos de interes mas alejados entre si son: “crater 1 metros 3 3” y “duna 2 metros 18 14”

Finalmente, se presenta el resultado de la ejecución del programa al realizar el plan de pruebas para cada uno de los coeficientes:

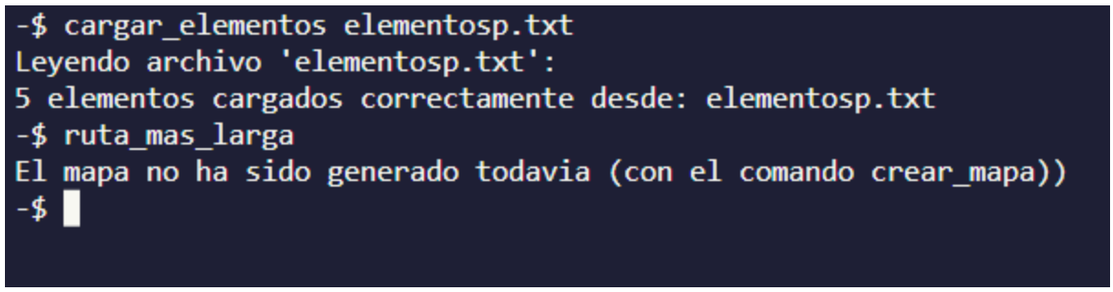


Figura 1. Prueba sin crear Mapa

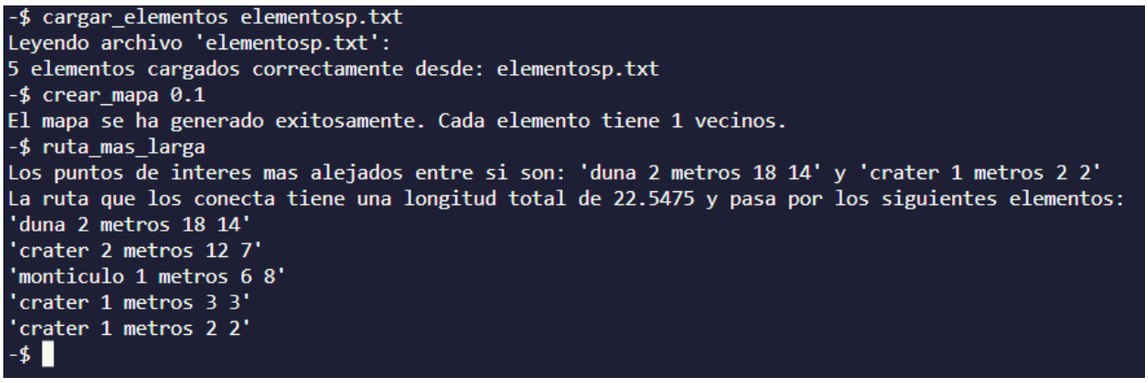


Figura 2. Prueba coeficiente 0.1

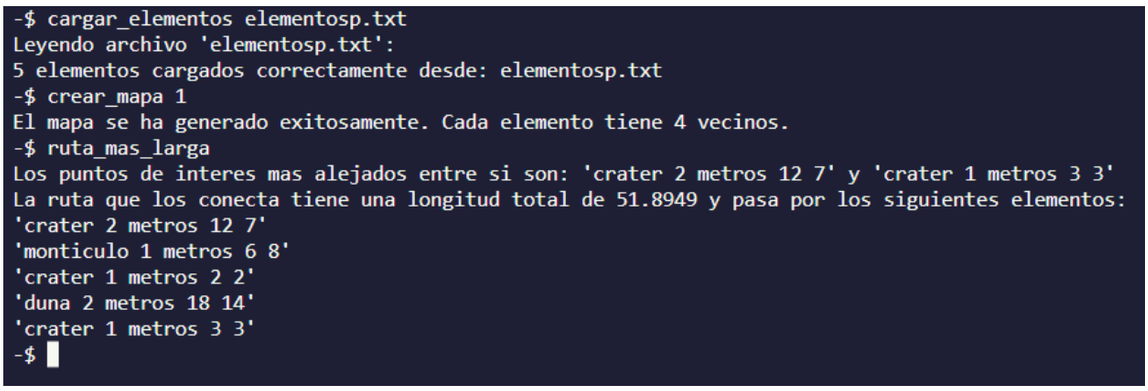


Figura 3. Prueba coeficiente 1

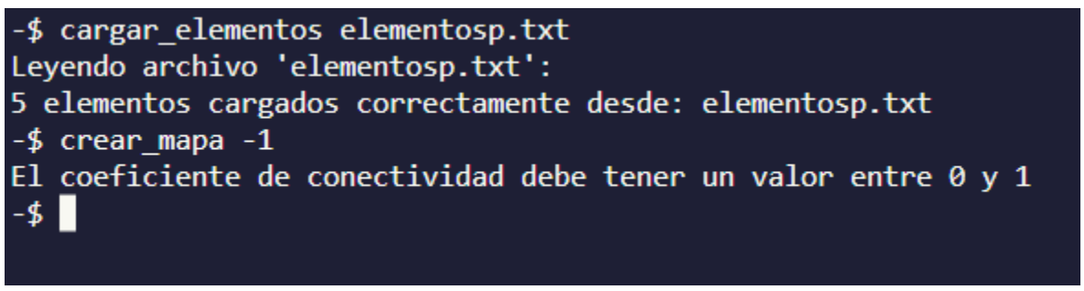


Figura 4. Prueba coeficiente -1

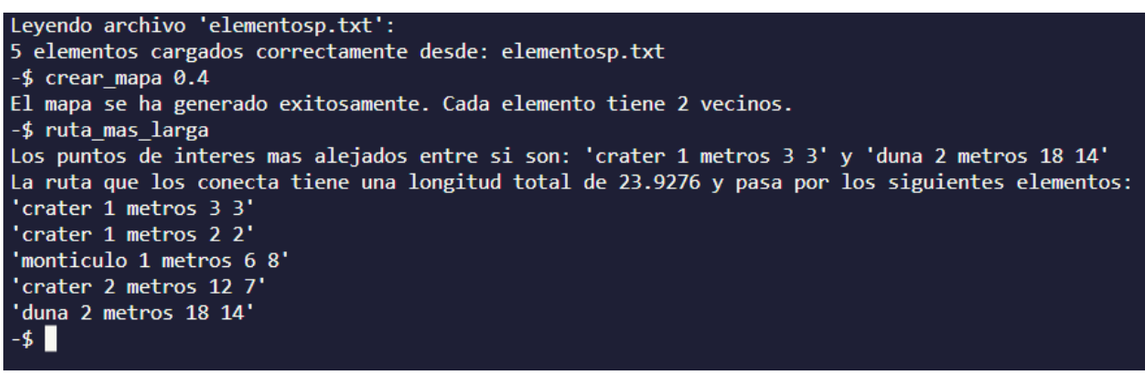


Figura 5. Prueba coeficiente 0.4