

PONTIFICIA UNIVERSIDAD JAVERIANA



PROYECTO

ENTREGA FINAL

JOHN CORREDOR

ESTRUCTURAS DE DATOS

NICOLAS ALGARRA

DANIEL GONZALEZ

JULIANA PACHECO

BOGOTA, COLOMBIA

2024

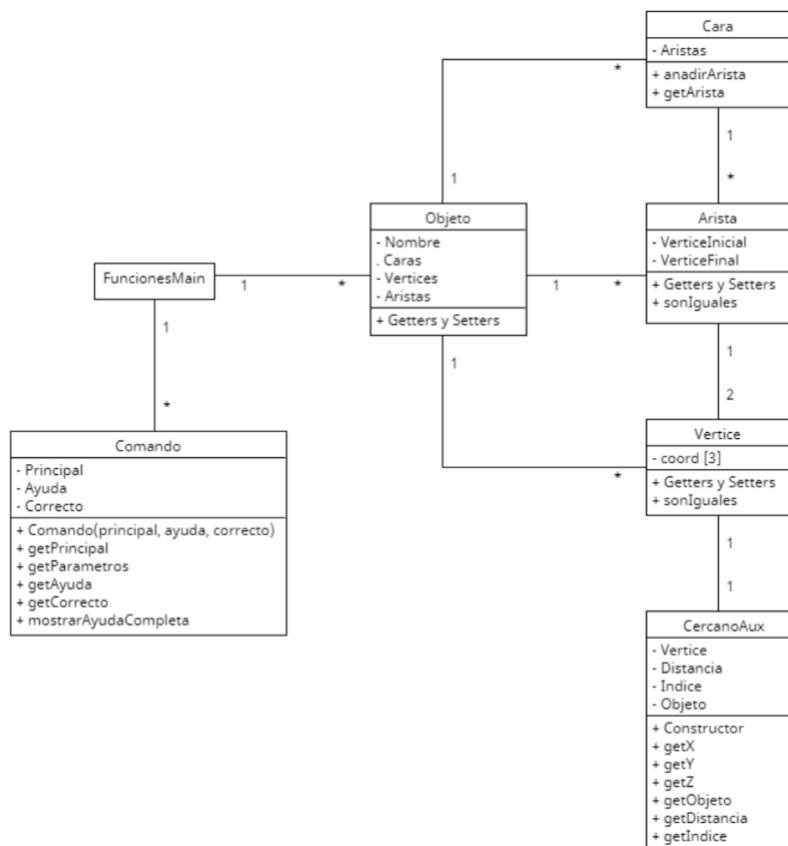
# ESTRUCTURAS DE DATOS

## DOCUMENTACION PROYECTO FINAL

### I) CORRECCIONES

No se realizaron correcciones a funcionalidades anteriores ya que todas estas funcionaban a la perfección y cumplían todos los requisitos pedidos para el proyecto, por esta razón en las entregas anteriores no se nos realizó ningún comentario negativo con respecto a este punto.

### II) DIAGRAMA DE FLUJO



### III) DISEÑO

- Funcionalidad del sistema

Este programa está diseñado para gestionar y manipular estructuras geométricas tridimensionales (objetos), representadas mediante vértices, aristas y caras. El sistema permite cargar, modificar, visualizar y operar sobre estos objetos a través de un conjunto de comandos. Está orientado al manejo de figuras tridimensionales, permitiendo la creación de envolventes y la comparación entre vértices.

#### *Gestión de Objetos Geométricos*

- El programa permite crear y gestionar objetos tridimensionales definidos por vértices, aristas y caras.
- Cada objeto tiene un nombre único que lo identifica y puede contener múltiples caras, compuestas por vértices y aristas.
- Los objetos se almacenan en una lista dinámica en memoria, lo que permite su manipulación continua durante la ejecución del programa.

#### *Cargar y Procesar Comandos*

- El sistema acepta comandos interactivos que permiten realizar diversas operaciones sobre los objetos geométricos.
- Los comandos iniciales se cargan en una lista utilizando la función `CargarComandos()`, que los lee desde un archivo o los inicializa en memoria.
- Durante la ejecución, el sistema procesa los comandos de forma continua mediante la función `CargarComando()`, que identifica y ejecuta la acción deseada.
- Entre las operaciones disponibles están:
  - Listar objetos geométricos.
  - Proporcionar ayuda sobre el uso de comandos.
  - Manipular componentes geométricos como vértices y aristas.

#### *Operaciones sobre Componentes Geométricos*

- Los vértices, aristas y caras de los objetos tridimensionales pueden ser manipulados mediante comandos específicos:
  - **Vértices:**
    - Consultar o modificar las coordenadas (X, Y, Z).
    - Calcular la distancia euclidiana entre dos vértices.

- **Caras:**
  - Añadir nuevas caras a un objeto.
  - Eliminar o listar las caras existentes.
- **Aristas:**
  - Definir conexiones entre vértices.
  - Manipular las aristas que componen las caras de los objetos.

### *Algoritmos de Procesamiento*

- El sistema incluye algoritmos auxiliares para facilitar cálculos geométricos y operaciones con los datos:
  - **getMinVertice:** Busca el vértice con el valor mínimo en una lista, útil para cálculos como la generación de cajas envolventes.
  - **Carga desde archivos:** Los datos de los objetos pueden inicializarse desde archivos externos con la función cargar(), que traduce los datos en estructuras utilizables.

### *Crear Cajas Envolventes*

- El sistema puede generar envolventes o cajas delimitadoras para los objetos tridimensionales utilizando los vértices mínimo y máximo.
- Estas cajas son útiles para simplificar cálculos geométricos como detección de colisiones o delimitación de áreas de influencia.

### *Interactividad Continua*

- El programa opera de forma interactiva mediante un ciclo infinito (while (true)), esperando la entrada del usuario y procesando comandos en tiempo real.
- Esto permite mantener la lista de objetos y sus componentes actualizados constantemente, facilitando la interacción y las modificaciones.

### *Ejecución del Sistema*

- La ejecución comienza con la función main(), que realiza la configuración inicial del entorno:
  - Configura la localización mediante setlocales() para manejar correctamente los caracteres en español.
  - Carga los comandos iniciales utilizando la función CargarComandos().
- Una vez configurado, el sistema entra en un ciclo continuo para procesar comandos, interactuar con el usuario y manipular los objetos geométricos en tiempo real.

## ➤ **ARISTA**

El TAD Arista representa una conexión entre dos vértices en un grafo. Cada arista tiene un vértice inicial y un vértice final. La clase incluye métodos para establecer y obtener los vértices asociados, así como para verificar si dos aristas son iguales, considerando tanto grafos dirigidos como no dirigidos.

### ***Entradas y Salidas***

- Constructor: Arista (const Vertice& VerticeA, const Vertice& VerticeB)
  - Entrada: Dos objetos de la clase Vertice que representan los extremos inicial y final de la arista.
  - Salida: Una nueva instancia de la clase Arista con los vértices proporcionados.
- Métodos de acceso y modificación de vértices (get y set):
  - Entrada:
    - Para obtener: Ninguna.
    - Para establecer: Un objeto de la clase Vertice.
  - Salida:
    - Para obtener: Devuelve una referencia constante al vértice inicial o final de la arista.
    - Para establecer: Modifica el vértice inicial o final de la arista.
  - Descripción: Los métodos de acceso (getVerticeInicial y getVerticeFinal) permiten obtener las referencias a los vértices asociados a la arista. Los métodos de modificación (setVerticeInicial y setVerticeFinal) permiten cambiar los vértices conectados por la arista.
- Método: sonIguales(const Arista& otraArista) const
  - Entrada: Un objeto de la clase Arista para comparar con la arista actual.
  - Salida: Devuelve un valor booleano indicando si las dos aristas conectan los mismos vértices. La comparación es simétrica.

### ***Condiciones***

- El constructor asume que los vértices proporcionados son válidos y existen dentro del grafo.
- El método sonIguales es conmutativo, es decir, devuelve true si las aristas conectan los mismos vértices, independientemente del orden.
- Los métodos de acceso (get) deben ser utilizados antes de modificar los vértices para obtener sus valores originales.

## ➤ **VERTICE**

El TAD Vertice representa un punto en el espacio tridimensional, definido por sus coordenadas (X, Y, Z) y un índice único. La clase incluye métodos para establecer y obtener las coordenadas, comparar vértices y calcular la distancia euclidiana entre puntos en el espacio.

### *Entradas y Salidas*

- **Constructor:** Vertice(float x, float y, float z, int nuevoIndice)
  - **Entrada:**
    - Tres valores flotantes que representan las coordenadas (X, Y, Z) del vértice.
    - Un valor entero que identifica el índice del vértice.
  - **Salida:** Una nueva instancia de la clase Vertice con las coordenadas y el índice proporcionados.
- **Métodos de acceso y modificación de coordenadas (get y set):**
  - **Entrada:**
    - Para obtener: Ninguna.
    - Para establecer: Un valor flotante que representa una coordenada específica (X, Y o Z).
  - **Salida:**
    - Para obtener: Devuelve el valor de una coordenada (X, Y o Z) o el índice del vértice.
    - Para establecer: Modifica el valor de una coordenada específica del vértice.
  - **Descripción:** Los métodos de acceso (getX, getY, getZ, getIndice) permiten consultar las coordenadas o el índice del vértice. Los métodos de modificación (setX, setY, setZ) permiten actualizar las coordenadas del vértice.
- **Método:** sonIguales(const Vertice &otro) const
  - **Entrada:** Un objeto de la clase Vertice para comparar con el vértice actual.
  - **Salida:** Devuelve un valor booleano indicando si las coordenadas de ambos vértices son idénticas.
- **Método:** DistanciaEuclidiana(float x, float y, float z) const
  - **Entrada:** Tres valores flotantes que representan un punto en el espacio tridimensional.
  - **Salida:** Devuelve un valor doble (double) que representa la distancia euclidiana entre el vértice actual y el punto proporcionado.

### *Condiciones*

- El constructor asume que las coordenadas proporcionadas son válidas dentro del contexto geométrico del sistema.
- El método sonIguales solo considera igualdad en coordenadas; el índice no es relevante para esta comparación.

- El método DistanciaEuclidiana calcula la distancia como la raíz cuadrada de la suma de los cuadrados de las diferencias entre las coordenadas correspondientes.

### ➤ **CARA**

El TAD Cara representa una cara en un objeto tridimensional, definida por una lista de aristas que forman su contorno. La clase permite gestionar las aristas asociadas a una cara, añadiendo nuevas aristas y accediendo a las existentes.

#### ***Entradas y Salidas***

- **Constructor:** Cara()
  - **Entrada:** Ninguna.
  - **Salida:** Una nueva instancia de la clase Cara inicializada sin aristas.
- **Método:** anadirArista(Arista& arista)
  - **Entrada:** Un objeto de la clase Arista que representa una arista a añadir.
  - **Salida:** Añade la arista proporcionada a la lista de aristas de la cara.
- **Método:** getAristas()
  - **Entrada:** Ninguna.
  - **Salida:** Devuelve una referencia constante a la lista de aristas que componen la cara.

#### ***Condiciones***

- La lista de aristas debe representar un contorno cerrado para definir una cara válida en un objeto tridimensional.
- El método anadirArista asume que la arista proporcionada es válida y pertenece al mismo objeto tridimensional que la cara.
- La lista de aristas devuelta por getAristas no debe modificarse directamente; se espera que las operaciones sobre la lista se realicen mediante los métodos de la clase Cara.

### ➤ **CERCANOAUX**

El TAD CercanoAux es una clase diseñada para almacenar información sobre un vértice cercano en un espacio tridimensional. Esta clase incluye referencias al vértice, su distancia respecto a un punto dado, un índice único y el nombre del objeto al que pertenece. Proporciona métodos para acceder a los atributos del vértice y a los datos relacionados con la cercanía.

## ***Entradas y Salidas***

- **Constructor:** CercanoAux(Vertex &Nuevovertice, double Nuevadistancia, int Nuevoindice, std::string nuevoObjeto)
  - **Entrada:**
    - Un objeto Vertex que representa el vértice cercano.
    - Un valor de tipo double que indica la distancia desde el vértice a un punto de referencia.
    - Un valor entero que representa el índice único del vértice.
    - Una cadena std::string que identifica el nombre del objeto al que pertenece el vértice.
  - **Salida:** Una instancia de la clase CercanoAux con los datos proporcionados.
- **Métodos de acceso (get):**
  - **Entrada:** Ninguna.
  - **Salida:**
    - getX(), getY(), getZ(): Devuelven las coordenadas del vértice (X, Y, Z).
    - getDistancia(): Devuelve la distancia almacenada entre el vértice y el punto de referencia.
    - getIndice(): Devuelve el índice único asociado al vértice.
    - getObjeto(): Devuelve el nombre del objeto al que pertenece el vértice.
    - getVertex(): Devuelve un puntero al objeto Vertex asociado.

## ***Condiciones***

- El vértice proporcionado debe ser válido y debe pertenecer a un objeto tridimensional registrado en el sistema.
- La distancia debe ser un valor calculado correctamente para garantizar la precisión de las operaciones geométricas.
- Los métodos de acceso (get) se utilizan exclusivamente para obtener información sin modificar los atributos del vértice o sus datos asociados.

## ➤ **COMANDO**

El TAD Comando representa un comando interactivo que puede ser ejecutado en el sistema. Cada comando tiene un identificador principal, una lista de mensajes de ayuda y una lista de formas correctas de uso. La clase proporciona métodos para acceder a estos atributos y para generar un mensaje completo de ayuda que combina las formas correctas de uso con sus respectivas descripciones.



### *Entradas y Salidas*

- **Constructor:** Comando (std::string &Nuevoprincipal, std::list<std::string> nuevaAyuda, std::list<std::string> Nuevocorrecto)
  - **Entrada:**
    - Una cadena que representa el nombre principal del comando.
    - Una lista de cadenas que contiene los mensajes de ayuda del comando.
    - Una lista de cadenas que define las formas correctas de uso del comando.
  - **Salida:** Una nueva instancia de la clase Comando con los atributos proporcionados.
- **Métodos de acceso (get):**
  - **Entrada:** Ninguna.
  - **Salida:**
    - getPrincipal(): Devuelve el identificador principal del comando.
    - getAyuda(): Devuelve la lista de mensajes de ayuda asociados al comando.
    - getCorrecto(): Devuelve la lista de formas correctas de uso del comando.
- **Método:** mostrarAyudaCompleta()
  - **Entrada:** Ninguna.
  - **Salida:** Devuelve una cadena que combina las formas correctas de uso y sus respectivas descripciones en un formato legible.

### *Condiciones*

- El constructor asume que las listas de ayuda y formas correctas tienen el mismo tamaño y que los elementos corresponden en el mismo orden.
- El método mostrarAyudaCompleta itera sobre las listas de ayuda y formas correctas simultáneamente, concatenando sus contenidos en un formato adecuado para mostrar al usuario.
- Los métodos de acceso (get) deben utilizarse para obtener información sin modificar los atributos del comando.

### ➤ **OBJETO**

El TAD Objeto representa una entidad tridimensional compuesta por vértices, aristas y caras. Esta clase permite gestionar los componentes geométricos del objeto, incluyendo la adición y recuperación de caras, vértices y aristas, así como la modificación de su nombre.

### *Entradas y Salidas*

- **Constructor:** Objeto(std::string &nombre)

- **Entrada:** Una cadena std::string que define el nombre único del objeto.
- **Salida:** Una instancia de la clase Objeto con el nombre especificado.
- **Métodos de acceso y modificación (get y set):**
  - **Entrada:**
    - Para obtener: Ninguna.
    - Para establecer: Una cadena para el nombre, o listas de componentes geométricos (caras, vértices o aristas).
  - **Salida:**
    - Para obtener:
      - getNombre(): Devuelve el nombre del objeto.
      - getCaras(), getVertices(), getAristas(): Devuelven referencias a las listas de caras, vértices y aristas del objeto.
    - Para establecer:
      - setNombre(): Cambia el nombre del objeto.
      - setVertices(), setAristas(): Reemplazan las listas actuales de vértices o aristas con las nuevas listas proporcionadas.
- **Métodos para agregar componentes geométricos:**
  - agregarCara(Cara &cara)
    - Entrada: Un objeto Cara que representa una cara a añadir.
    - Salida: Añade la cara a la lista de caras del objeto.
  - agregarArista(Arista arista)
    - Entrada: Un objeto Arista que representa una arista a añadir.
    - Salida: Añade la arista a la lista de aristas del objeto.
  - agregarVertice(Vertice vertice)
    - Entrada: Un objeto Vertice que representa un vértice a añadir.
    - Salida: Añade el vértice a la lista de vértices del objeto.

### ***Condiciones***

- El constructor asume que el nombre proporcionado es único y válido dentro del contexto del sistema.
- Los métodos de acceso (get) permiten obtener referencias a las listas de componentes geométricos, que no deben ser modificadas directamente fuera de la clase.
- Los métodos para agregar componentes (agregarCara, agregarArista, agregarVertice) asumen que los componentes proporcionados son válidos y coherentes con la estructura geométrica del objeto.

### ➤ **AUXDIJKSTRA**

El TAD AuxDijkstra es una clase auxiliar diseñada para facilitar la implementación del algoritmo de Dijkstra en grafos. La clase representa un nodo o vértice durante el cálculo del camino más corto, almacenando información sobre su costo acumulado, su predecesor en el camino, si ha sido visitado y los destinos posibles.

### ***Entradas y Salidas***

- **Constructor:** AuxDijkstra(Vertex \*ReferenciaElemento)
  - **Entrada:** Un puntero al objeto Vertex que representa el nodo asociado.
  - **Salida:** Una instancia de la clase AuxDijkstra inicializada con:
    - Camino = -1 (predecesor no definido).
    - Costo = infinito (indicando que aún no se ha calculado un camino).
    - Visitado = false (nodo no visitado).
- **Métodos de acceso y modificación (get y set):**
  - **Entrada:**
    - Para obtener: Ninguna.
    - Para establecer: Valores correspondientes a los atributos de la clase.
  - **Salida:**
    - getElemento(): Devuelve un puntero al vértice asociado.
    - getCamino(): Devuelve el índice del predecesor en el camino más corto.
    - getCosto(): Devuelve el costo acumulado hasta este vértice.
    - isVisitado(): Devuelve un booleano indicando si el nodo ha sido visitado.
    - getDestinos(): Devuelve una referencia constante a la lista de índices de los nodos destino.
    - setCamino(int camino): Establece el índice del predecesor.
    - setCosto(double nuevoCosto): Actualiza el costo acumulado.
    - setVisitado(bool nuevoVisitado): Marca el nodo como visitado o no.
    - addDestino(int destino): Añade un índice de nodo destino a la lista.

### ***Condiciones***

- El constructor asume que el puntero ReferenciaElemento apunta a un objeto Vertex válido dentro del contexto del grafo.
- Los métodos de acceso (get) solo recuperan información sin modificar los atributos de la clase.
- Los métodos de modificación (set y addDestino) aseguran la consistencia de los datos necesarios para el algoritmo de Dijkstra.
- Los destinos añadidos mediante addDestino deben corresponder a índices válidos dentro del grafo.

## ➤ FUNCIONESMAIN

El archivo FuncionesMain.cpp contiene funciones globales que implementan la lógica principal para cargar comandos, procesar objetos, y manejar operaciones auxiliares en el sistema. Estas funciones permiten la interacción con los datos almacenados y facilitan la ejecución de operaciones sobre ellos.

### Funciones Principales

#### 1. CargarComandos()

- a. **Descripción:** Lee comandos desde un archivo de texto (Ayuda.txt) y los carga en una lista de objetos Comando.
- b. **Entradas:** Ninguna.
- c. **Salidas:** Devuelve una lista (std::list<Comando>) con los comandos cargados.
- d. **Condiciones:**
  - i. El archivo Ayuda.txt debe existir y ser accesible.
  - ii. El formato del archivo debe separar los datos por punto y coma (;).

#### 2. CargarComando(std::list<Comando> comandos, std::list<Objeto> &objetos)

- a. **Descripción:** Procesa un comando ingresado por el usuario, ejecutando acciones sobre la lista de objetos o mostrando mensajes de ayuda.
- b. **Entradas:**
  - i. Una lista de comandos (std::list<Comando>).
  - ii. Una referencia a una lista de objetos (std::list<Objeto>).
- c. **Salidas:** Ninguna (realiza acciones directamente sobre las listas).
- d. **Condiciones:**
  - i. La lista de comandos debe contener al menos un comando válido.
  - ii. Los comandos ingresados por el usuario deben coincidir con los disponibles en la lista.

#### 3. BuscarComandoAyuda(std::string ingresado, std::list<Comando> comandos)

- a. **Descripción:** Busca un comando específico en la lista y devuelve su información de ayuda.
- b. **Entradas:**
  - i. Un string con el comando a buscar.
  - ii. Una lista de comandos (std::list<Comando>).
- c. **Salidas:** Imprime la ayuda del comando en caso de encontrarlo.
- d. **Condiciones:**
  - i. La entrada debe coincidir exactamente con un comando de la lista.

#### 4. cargar(std::string NombreArchivo, std::list<Objeto> &Objetos)

- a. **Descripción:** Carga un archivo con datos sobre objetos tridimensionales y los agrega a una lista.
  - b. **Entradas:**
    - i. Un string con el nombre del archivo a leer.
    - ii. Una referencia a la lista de objetos (std::list<Objeto>).
  - c. **Salidas:** Devuelve un valor booleano indicando si la operación fue exitosa.
  - d. **Condiciones:**
    - i. El archivo debe estar en el formato correcto para ser procesado.
5. **listado**(std::list<Objeto> &Objetos)
- a. **Descripción:** Muestra en pantalla la lista de objetos cargados.
  - b. **Entradas:**
    - i. Una referencia a la lista de objetos (std::list<Objeto>).
  - c. **Salidas:** Imprime los nombres de los objetos en la consola.
  - d. **Condiciones:**
    - i. La lista de objetos no debe estar vacía para que se pueda mostrar información útil.
6. **getMinVertice**(std::list<Vertice> vertices)
- a. **Descripción:** Encuentra el vértice con la menor coordenada en una lista de vértices.
  - b. **Entradas:**
    - i. Una lista de vértices (std::list<Vertice>).
  - c. **Salidas:** Devuelve el vértice con el valor más bajo.
  - d. **Condiciones:**
    - i. La lista debe contener al menos un vértice.

## ➤ FUNCION MAIN ( )

El archivo main.cpp contiene el punto de entrada principal del programa. Configura el entorno y administra la ejecución para procesar comandos interactivos que operan sobre una lista de objetos tridimensionales. Este archivo conecta y coordina las funcionalidades de las clases y funciones implementadas en el proyecto.

## Funcionalidad del Programa

### 1. Configuración del Entorno:

- a. La función setlocale(LC\_ALL, "es\_ES.UTF-8") configura el entorno para manejar caracteres en español, garantizando que las cadenas y salidas del sistema funcionen correctamente con la codificación regional.

### 2. Carga Inicial de Comandos:

- a. Se llama a la función `CargarComandos()` para inicializar una lista (`std::list<Comando>`) con los comandos disponibles, cargándolos desde un archivo externo.
- 3. Ejecución del Ciclo Principal:**
- a. Se inicializa una lista vacía de objetos (`std::list<Objeto>`).
  - b. El programa entra en un ciclo infinito (`while (true)`) que:
    - i. Espera entradas del usuario.
    - ii. Procesa los comandos ingresados utilizando la función `CargarComando(comandos, objetos)`.
    - iii. Permite realizar operaciones sobre la lista de objetos (como agregar, listar o manipular componentes geométricos).

### Condiciones

- Archivo de comandos: El archivo que contiene los comandos debe existir y estar correctamente formateado para ser cargado por `CargarComandos()`.
- Entradas del usuario: Se espera que los comandos ingresados sean válidos y estén definidos en la lista cargada inicialmente.
- Ciclo infinito: El programa continuará ejecutándose indefinidamente hasta que sea finalizado manualmente.

## IV) COMANDOS DE COMPILACIÓN

El siguiente código se usa para compilar el proyecto desde la terminal

```
> g++ -o Proyecto main.cpp libs/*.cpp  
> ./Proyecto
```

## V) PLAN DE PRUEBAS

Dentro de este apartado se va a diseñar el plan de pruebas destinado a la función *ruta\_corta*, para esto se cargó dentro del programa el objeto llamado Mesh\_0, después de esto se va a probar el código con la siguiente tabla de pruebas:

#	Código	Resultado esperado
1	ruta_corta	Comando escrito incorrectamente
2	ruta_corta 1	Comando escrito incorrectamente
3	ruta_corta 1 Mesh_0	Comando escrito incorrectamente
4	ruta_corta 1 2 objeto	El objeto no existe en el programa
5	ruta_corta 1 1 Mesh_0	Los índices son iguales
6	ruta_corta 1 15 Mesh_0	Alguno de los índices no está en el objeto
7	ruta_corta 1 2 Mesh_0	La ruta más corta que conecta los vértices 1 y 2 del objeto Mesh_0. pasa por: 1-2; con una longitud de 8,246211

Resultados:

- **Prueba #1:**

```
$ruta_corta
El comando esta escrito de manera erronea, escriba ayuda ruta_corta para más información
```

Como se puede ver, el resultado en el programa cumple con el resultado esperado en el plan de pruebas

- **Prueba #2:**

```
$ruta_corta 1
El comando esta escrito de manera erronea, escriba ayuda ruta_corta para más información
```

Como se puede ver, el resultado en el programa cumple con el resultado esperado en el plan de pruebas

- **Prueba #3:**

```
$ruta_corta 1 Mesh_0
El comando esta escrito de manera erronea, escriba ayuda ruta_corta para más información
```

Como se puede ver, el resultado en el programa cumple con el resultado esperado en el plan de pruebas

- **Prueba #4:**

```
$ruta_corta 1 2 objeto  
El objeto objeto no ha sido cargado en memoria
```

Como se puede ver, el resultado en el programa cumple con el resultado esperado en el plan de pruebas

- **Prueba #5:**

```
$ruta_corta 1 1 Mesh_0  
Los indices de los vertices dados son iguales
```

Como se puede ver, el resultado en el programa cumple con el resultado esperado en el plan de pruebas

- **Prueba #6:**

```
$ruta_corta 1 15 Mesh_0  
Algunos de los indices de vertices estan fuera de rango para el objeto Mesh_0
```

Como se puede ver, el resultado en el programa cumple con el resultado esperado en el plan de pruebas

- **Prueba #7:**

```
$ruta_corta 1 2 Mesh_0  
La ruta más corta que conecta los vertices 1 y 2 del objeto Mesh_0. pasa por: 1-2; con una longitud de 8,246211.
```

Como se puede ver, el resultado en el programa cumple con el resultado esperado en el plan de pruebas