UNIVERSIDAD MARIANO GÁLVEZ DE GUATEMALA

CENTRO UNIVERSITARIO DE ESCUINTLA

FACULTAD DE INGENIERÍA

Carrera: Ingeniería en Sistemas de Información

Curso: Programación III

Proyecto Final

ESTUDIANTES:

Suarlyn Rodolfo Juárez Quevedo- 2190-22-11499

Diego Andreé López Baches- 2190-23-626

Jeferson Rolando García Galicia-2190-23-6735

Santiago Vásquez Díaz- 2190-23-9111

LUGAR Y FECHA:

Escuintla, Escuintla 20-07-2024

Sistema de rutas de entrega

Introducción

Nuestro proyecto se basa en la simulación de un sistema de rutas de entrega, en donde el usuario debe de realizar la petición de pedido de algún producto, luego realizar la petición nuestro algoritmo busca la ruta más corta hacía la ubicación del usuario. Optimizando así el tiempo y recursos para poder llegar hacia él.

Problema

Una empresa presenta el problema de que sus rutas de entrega se encuentran algo largas y han obtenido quejas de sus clientes, por lo cual se desea crear un sistema de creación de rutas para la optimización de una empresa encargada de la entrega de paquetes hacia clientes.

Solución:

Se realizó un sistema para poder optimizar las rutas dentro del país, y así realizar tramos más cortos optimizando recursos para la empresa.

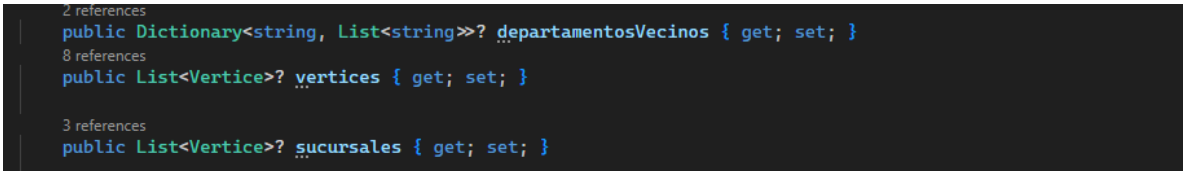
Estructura del proyecto

Clases

- Grafo
- Archivador
- Vértice
- Arista
- CargadorVecinos

Grafo

Esta clase es la encargada de procesar las los puntos que tenemos cargados, conectarlos entre ellos de manera geográfica lógica para posteriormente aplicar el algoritmo y de esa manera encontrar la ruta más corta.



Cuenta con estos tres atributos, necesarios para su funcionamiento.

departamentosVecinos: Almacenamos en un Diccionario los departamentos vecinos, en donde en la key del diccionario es el nombre del departamento y el valor es la lista de los departamentos vecinos de ese departamento.

vertices: Almacenamos una lista de Vertices que utilizaremos par el grafo

sucursales: Almacenamos una lista de vertices en el cual están las sucursales que cuenta la compañía.

Esta clase contiene los siguientes métodos.

- **Constructor**
- **cargarVertices()**
- **cargarVecinos()**
- **cargarSucursales()**
- **conectarGeograficamente(Lista<Vertices> vertices, Diccionario<cadena, Lista<cadena> vecinos)**
- **agregarConexiones(Vertice origen, Lista<Vertice> destinos, Random rand, bool vecinos);**
- **encontrarCamino(int origenID, int destinoID)**

Cada uno se complementa para lograr tener la funcionalidad de un grafo para la solución de nuestro problema.

Constructor:

Este método no requiere de parámetros, solamente lo utilizamos para cargar toda la información necesaria para poder funcionar, como lo es:

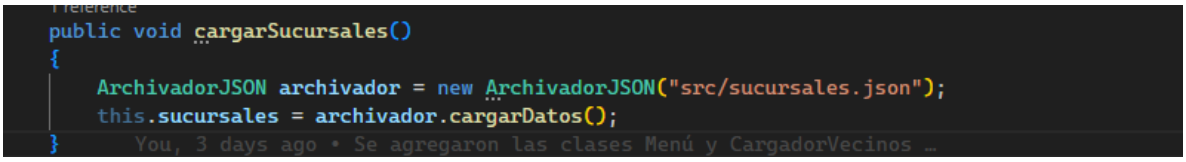
- Cargar los vecinos
- Cargar vértices
- Conectar gráficamente todos los vértices

cargarVertices():

Este método utiliza la clase ArchivadorJSON para poder obtener los datos de nuestro archivo JSON en donde se encuentran los puntos de referencia y convertirlos vértices, para posteriormente cargarlos en nuestro atributo **vertices**

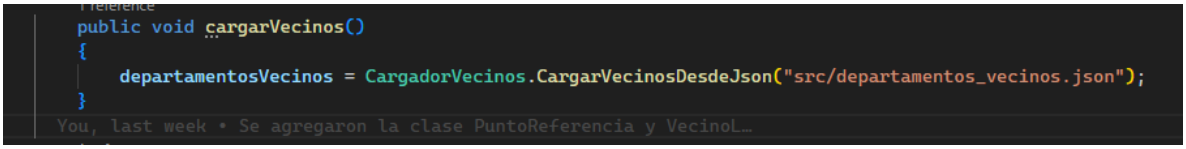
cargarSucursales():

Este método se hace cargo de cargar las sucursales dentro de nuestro atributo sucursales de nuestra clase.



cargarVecinos():

Este método se hace cargo de almacenar en nuestro atributo departamentosVecinos un diccionario con el nombre del departamento y una lista con los nombres de los departamentos vecinos.



La complejidad de la clase entra cuando hay que tener lógica en cada conexión del vértice, ya que necesitamos simular el mapa real de Guatemala. Así que se emplearon los siguientes algoritmos.

agregarConexiones:

Utilizamos este método para realizar la conexión persé en el grafo, creando así las aristas necesarias con el peso correspondiente a su ubicación.

Parámetros:

- **Vertice origen**
- **List<Vertice> destinos**
- **Random rand**
- **Bool vecinos**

Utilizamos un HashSet<int> para almacenar los destinos ya usados, y de esa manera no repetir ninguno.

Luego, recorremos un foreach de cada destino. Si el destino se encuentra en nuestro HashSet, entonces continuamos al siguiente ciclo.

De no ser así, seguimos con el algoritmo.

Agregamos el destino al HashSet, para posteriormente inicializar nuestra variable tiempo que nos será útil para saber que tanto tiempo (que es el peso de nuestra Arista).

Para poder tener congruencia en los pesos de las aristas, nos basamos en un algoritmo sencillo.

Si solamente es vecino, le asignamos un tiempo de 30 a 60 minutos, de forma aleatoria.

En caso de que el origen de ese vecino sea dentro del mismo departamento, entonces le asignamos un tiempo más corto, de 10 a 30 minutos.

De lo contrario, le asignamos un tiempo de 60 a 180 minutos.

Por último, agregamos esa arista al vértice.

```
private void agregarConexiones(Vertice origen, List<Vertice> destinos, Random rand, bool vecinos = false)
{
    HashSet<int> usados = new HashSet<int>();
    foreach (var destino in destinos)
    {
        if (usados.Contains(destino.id)) continue;

        usados.Add(destino.id);
        int tiempo = 0;
        if (vecinos)
        {
            // Si es vecino, asignar un tiempo aleatorio entre 30 y 60 minutos
            tiempo = rand.Next(30, 60);
        }
        else if (origen.departamento == destino.departamento)
        {
            // Si es del mismo departamento, asignar un tiempo aleatorio entre 10 y 30 minutos
            tiempo = rand.Next(10, 30);
        }
        else
        {
            tiempo = rand.Next(60, 180);
        }
        origen.agregarArista(new Arista(origen, destino, tiempo));
    }
```

conectarGeograficamente:

Parámetros:

- **List<Vertice> vertices**
- **Dictionary<string, List<string>> vecinosDepartamentales**

Utilizamos la clase **Random** para la generación de números aleatorios que posteriormente nos será útil.

Recorremos cada vértice, para luego buscar todos los vértices que esten en el mismo departamento y así agregar la conexión a ese vértice con la función **agregarConexiones.**

Debemos de asegurar que estamos conectar los departamentos vecinos para simular de la manera más realista posible.

Nos tenemos que asegurar de conectar cada departamento con su sucursal, así que obtenemos nuestras sucursales desde vértices, y si existe sucursal agregamos esa conexión.

Después comprobamos si existen vecinos, para después buscar dentro de vértices y obtener la lista de vértices vecinos hacia él.

Una vez obtenida esa lista, realizamos un filtro para eliminar cada sucursal de esa lista, para posteriormente agregar la conexión.

```
public void conectarGeograficamente(List<Vertice> vertices, Dictionary<string, List<string>> vecinosDepartamentales)
{
    Random rand = new Random();

    foreach (var v in vertices)
    {
        // Conectar dentro del mismo departamento
        var mismosDepto = vertices.FindAll(d => d.departamento == v.departamento && d.id != v.id);
        agregarConexiones(v, mismosDepto, rand);

        //Asegurarse de conectar la sucursal (si es que existe)
        var sucursal = vertices.FindAll(v2 => v2.departamento == v.departamento && (v2.tipo == "sucursal" || v2.tipo == "sucursal central") && v2.id != v.id);
        if (sucursal != null)
        {
            agregarConexiones(v, sucursal, rand);
        }
        // Conectar con departamentos vecinos si existen
        if (vecinosDepartamentales.TryGetValue(v.departamento, out var vecinos))
        {
            var vecinosDepto = vertices.FindAll(v => vecinos.Contains(v.departamento));
            vecinosDepto.RemoveAll(i => i.tipo == "sucursal" || i.tipo == "sucursal central");
            agregarConexiones(v, vecinosDepto, rand, vecinos: true);
        }

    }
}
```

encontrarCamino:

Este es el método más complejo de nuestra clase, es el algoritmo encargado de encontrar la ruta más corta desde un punto de origen hacia un destino dentro de nuestro grafo.

Parámetros

- **Int origenId**
- **Int destinoId**

Retorno

- **Tupla** (List<Vertice>, int)

En caso de que no haya vertices, retornamos una tupla vac[ia.

De lo contrario, seguimos con nuestro algoritmo.

El algoritmo utilizado es llamado Dijkstra.

Utilizamos 2 variables clave, para almacenar el destino y el origen utilizando solo el ID.

Si no son nulos, continuamos con el algoritmo.

Inicializamos variables necesarias para el algoritmo.

```
var distancias = new Dictionary<Vertice, int>();
var previo = new Dictionary<Vertice, Vertice?>();
var visitados = new HashSet<Vertice>();
var cola = new List<(Vertice vertice, int prioridad)>();
```

distancias: Un diccionario que guarda la distancia mínima conocida desde el origen a cada vértice. Se inicializa con infinito (int.MaxValue) para todos, excepto el origen que se pone en 0.

previo: Un diccionario que guarda el predecesor de cada vértice en el camino más corto encontrado.

visitados: Un conjunto para registrar los vértices ya procesados.

cola: Una lista que funciona como cola de prioridad, donde cada elemento es una tupla con el vértice y su distancia actual desde el origen.

A continuación entramos en un bucle foreach en donde recorremos cada vértice, y le asignamos un lugar en cada diccionario (distancias y previo), necesarias para el algoritmo.

Después, agregamos el origen en el diccionario distancias y se le asigna el valor 0. Además de agregar a la cola el origen, junto con la prioridad de 0.

```
foreach (var v in vertices)
{
    distancias[v] = int.MaxValue;
    previo[v] = null;
}
distancias[origen] = 0;
cola.Add((origen, 0));
```

Ahora, en nuestro bucle while que se ejecutará hasta que la cantidad de elementos en cola sea mayor a 0.

Se ordena la cola por prioridad, que sería la distancia más corta.

Luego, extraemos el vértice con menor distancia guardado en la variable local actual.

Si este ya fue visitado, se salta al siguiente ciclo, de lo contrario se guarda en nuestro HashSet visitados.

Si el vértice actual es el destino, terminamos el bucle.

De lo contrario, para cada arista del vértice actual realizamos lo siguiente:

Calculamos la nueva distancia al vecino sumando la distancia actual y el peso de la arista.

Si esta nueva distancia es menor que la registrada se actualiza la distancia y el predecesor, y se agrega el vecino a la cola.

```
while (cola.Count > 0)
{
    // Extraer el nodo con menor prioridad (distancia)
    cola.Sort((a, b) => a.prioridad.CompareTo(b.prioridad));
    var actual = cola[0].vertice;
    cola.RemoveAt(0);

    if (visitados.Contains(actual)) continue;
    visitados.Add(actual);

    if (actual == destino) break;

    foreach (var arista in actual.aristas)
    {
        var vecino = arista.destino;
        int nuevaDist = distancias[actual] + arista.tiempo_camino;
        if (nuevaDist < distancias[vecino])
        {
            distancias[vecino] = nuevaDist;
            previo[vecino] = actual;
            cola.Add((vecino, nuevaDist));
        }
    }
}
```

Luego, corroboramos el camino de la siguiente manera: Si la distancia al destino sigue siendo infinita (Es decir, el valor máximo de int), es señal de que no hay camino posible, y se imprime un mensaje y retorna 0.

```
if (distancias[destino] == int.MaxValue)
{
    Console.WriteLine("No existe un camino entre los nodos dados.");
    return (new List<Vertice>(), 0);      You, 2 seconds ago • Uncommitted changes
}
```

Después, reconstruimos el camino dentro de una lista de Vértices, en donde con un ciclo for, en donde hacemos un recorrido reverso. Se comienza desde el vértice destino, que mientras v no sea null se agrega v a la lista camino. Luego actualizamos el valor de v al predecesor de v usando el diccionario previo.

Esto se repite hasta que se llega al origen, cuyo predecesor es null.

Luego invertimos la lista del camino para que se muestre desde el origen hasta el destino.

Finalmente, retornamos una tupla con el camino y la distancia total recorrida.

```
// Reconstruir el camino
var camino = new List<Vertice>();
for (var v = destino; v ≠ null; v = previo[v])
    camino.Add(v);
camino.Reverse();
return (camino, distancias[destino]);
```

Ruta

Esta clase estática se encarga de calcular y mostrar la ruta óptima para la entrega de un paquete, desde la sucursal central hasta el destino final del cliente, pasando por la sucursal más cercana.

Métodos:

- MostrarRuta(Grafo grafo, int destino)

Método MostrarRuta(Grafo grafo, int destino)

Este método determina el camino más corto desde la sucursal central hasta el cliente, pasando por la sucursal más cercana al destino. Posteriormente, muestra en consola la información detallada del trayecto, incluyendo el tiempo estimado y cada punto de la ruta.

Parámetros:

- Grafo: Estructura de datos que contiene las sucursales, vértices y aristas de las ubicaciones disponibles.
- int destino: ID del vértice que representa la ubicación del cliente.

Recorre todas las sucursales para encontrar la más cercana al destino del cliente.

Calcula el camino desde la sucursal central hasta esa sucursal cercana (si es necesario).

Concatena ambos tramos del camino para formar una ruta completa.

Calcula el tiempo estimado total del trayecto usando los pesos de las aristas.

Muestra en consola:

El departamento de destino.

La sucursal más cercana.

El tiempo estimado del recorrido (en minutos u horas y minutos).

El detalle paso a paso del recorrido, incluyendo el tipo de punto y el tiempo de traslado entre puntos.

Vertice

Esta clase es necesaria para la creación de nuestra clase Grafos, esta cuenta con:

Atributos:

- Int id
- String departamento
- String tipo
- List<Arista> aristas

Métodos:

- Constructor
- agregarArista

Método constructor

Parámetros:

- **int id**
- **string departamento**
- **string tipo**

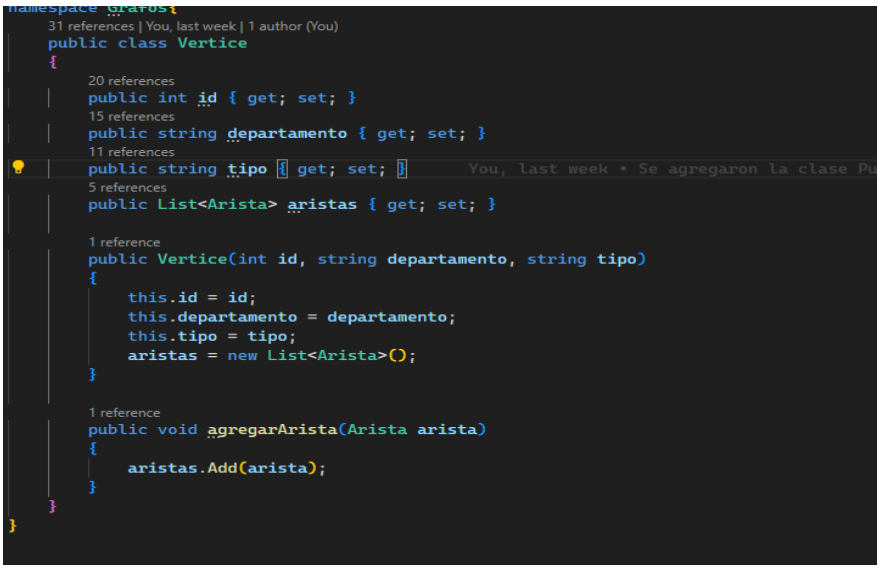
Solamente inicializamos estos atributos para nuestra clase.

Método agregarArista

Parámetros:

- **Arista**

Se utiliza para agregar a la lista de aristas, la arista que se pasa como argumento.



Arista

Esta es otra clase que es necesaria para la creación de los grafos.

Cuenta con:

Atributos:

- Vertice origen
- Vertice destino
- Int tiempo_camino

Métodos:

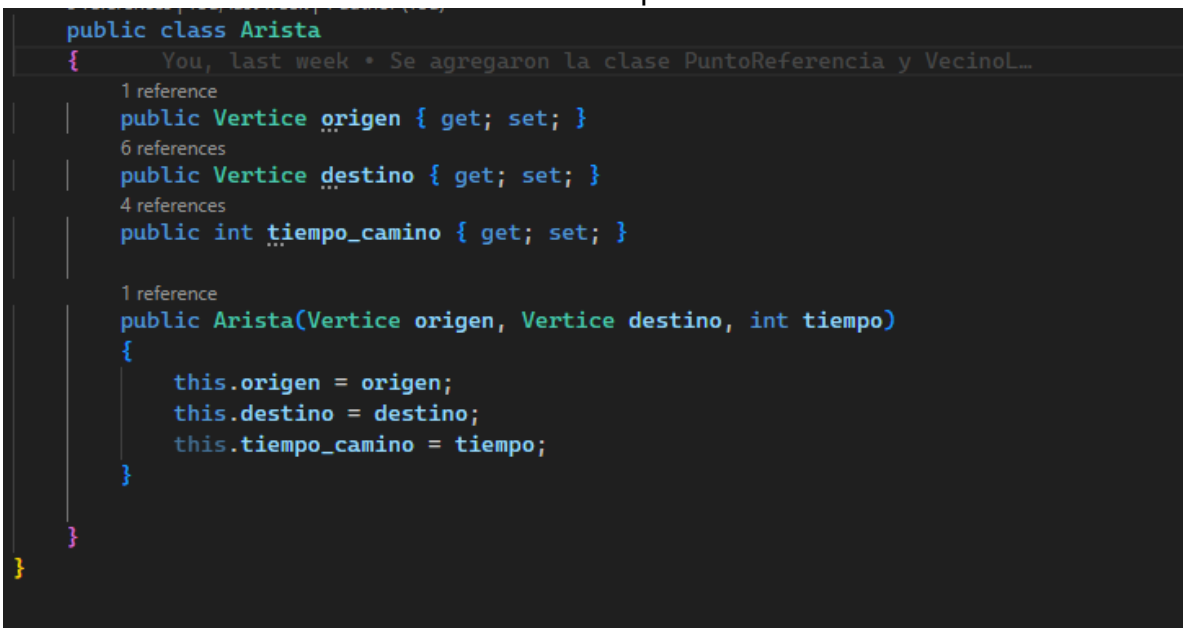
- Constructor

Método constructor

Parámetros

- Vertice origen
- Vertice destino
- Int tiempo

En este constructor inicializamos los valores para nuestra arista.



CargadorVecinos

Define una clase estática llamada CargadorVecinos que contiene un método para cargar información de departamentos vecinos desde un archivo JSON.

Explicación paso a paso

1. **Importación de namespaces**

```
using System.Text.Json;
```

Se importa el espacio de nombres necesario para trabajar con JSON.

2. **Clase estática**

```
public static class CargadorVecinos
```

La clase es estática, por lo que no necesita instanciarse para usar sus métodos.

3. **Método principal**

```
public static Dictionary<string, List<string>>
CargarVecinosDesdeJson(string ruta)
```

   Es un método estático que recibe la ruta de un archivo JSON.

   Devuelve un diccionario donde la clave es el nombre de un departamento y el valor es una lista de nombres de departamentos vecinos.

4. **Lectura y deserialización**

```
string contenido = File.ReadAllText(ruta);
var vecinos = JsonSerializer.Deserialize<Dictionary<string,
List<string>>>(contenido);
```

   Lee todo el contenido del archivo especificado en ruta.

   Deserializa el contenido JSON a un diccionario de tipo Dictionary<string, List<string>>.

5. **Manejo de errores**

```
catch (Exception ex)
{
  Console.WriteLine($"Error cargando vecinos: {ex.Message}");
    return new Dictionary<string, List<string>>();
}
```

   - Si ocurre algún error (por ejemplo, el archivo no existe o el JSON está mal formado), muestra un mensaje en consola y retorna un diccionario vacío.

Clase Departamentos

Es una clase estática, por lo que:

- No se puede instanciar.
- Sus métodos se llaman directamente usando Departamentos.<método>.

```
public static class Departamentos
```

Método: encontrarEnElMismoDepto(string departamento)

Devuelve una lista de todos los vértices que pertenecen al departamento especificado, excepto los que son sucursales ("sucursal" o "sucursal central").

1. Carga los datos desde el archivo puntos_guatemala_500.json.
2. Filtra los vértices que:
    - Tienen el mismo nombre de departamento.
    - No sean sucursales.

```
public static List<Vertice> encontrarEnElMismoDepto(String departamento)
```

Método: encontrarDepartamentos()

Devuelve todos los vértices cuyo tipo es "municipio", utilizados para representar departamentos.

1. Carga todos los datos desde el mismo archivo JSON.
2. Filtra solo los vértices donde tipo == "municipio".

```
public static List<Vertice> encontrarDepartamentos()
```

Clase PuntoReferencia

Es una clase necesaria para la carga de puntos de referencia desde el archivo JSON

Atributos:

- **id: de tipo int.** Representa un identificador numérico único para el punto de referencia.
- **departamento**: de tipo string. Posiblemente indica la ubicación o área geográfica asociada con el punto.
- **tipo: de tipo string.** Podría indicar la categoría o clasificación del punto de referencia (por ejemplo, una estación, una parada, un hito, etc.).

Menu

Esta clase estática representa el sistema de menús en consola utilizado para mostrar opciones, recibir entradas del usuario y simular una animación de carga.

Atributos:

- ConsoleColor colorLetraActual
- Dictionary<int, string> opciones

Métodos:

- cargarOpciones()
- dibujarOpciones()
- cargarAnimación()
- dibujarMenu()
- esperarRespuesta(out int respuesta)
- cambiarColor(ConsoleColor? color = null)
- mostrarDepartamentos()

Método cargarOpciones()

Carga las opciones disponibles en el menú principal.

Agrega las opciones "Realizar pedido" y "Salir" al diccionario de opciones.

Método dibujarOpciones()

Descripción:

Dibuja en consola cada una de las opciones disponibles en el menú, con formato y color.

Método cargarAnimación()

Simula una animación de carga en la consola, mostrando puntos y cambiando de color.

Imprime "Cargando recursos" en pantalla con un efecto visual.

Método dibujarMenu()

Limpia la consola y muestra el título del sistema junto con las opciones disponibles.

Método esperarRespuesta(out int respuesta)

Espera la entrada del usuario desde la consola y la convierte a un entero.

Parámetros:

- out int respuesta: variable de salida donde se almacena la respuesta del usuario.

Retorna:

- El valor ingresado como entero.

Método cambiarColor(ConsoleColor? color = null)

Cambia el color del texto en la consola. Si no se proporciona un color, usa el color predeterminado (colorLetraActual).

Parámetros:

- ConsoleColor? color: color opcional a aplicar.

Método mostrarDepartamentos()

Muestra en consola los departamentos disponibles, obtenidos desde la clase Departamentos.

Imprime la lista de departamentos con su ID y nombre.

Programa Principal (Main)

Este bloque de código representa el ciclo principal del sistema de entrega de paquetería. Se encarga de iniciar el menú, recibir la selección del usuario y ejecutar la funcionalidad correspondiente.

Al iniciar el programa, se crea una instancia de la clase Grafo, que contiene la información necesaria sobre los vértices, las aristas y las ramas. Las opciones del menú se cargan mediante Menu.cargarOpciones ().

A partir de ahí, el sistema entra en un bucle principal (while) donde se muestra el menú en la consola y se espera a que el usuario seleccione una opción mediante Menu.esperarRespuesta(out int opc).

Cuando el usuario elige la opción 1, el sistema le permite seleccionar el departamento al que pertenece. Para ello, se listan todos los departamentos disponibles mediante Menu.mostrarDepartamentos() y se le solicita al usuario que introduzca su ID de ubicación. Una vez hecho esto, se llama al método Route.MostrarRuta(graph, departamentoSeleccionado) para calcular y mostrar la ruta más eficiente para la entrega del paquete.

Si el usuario elige la opción 2, el sistema finaliza la ejecución. Se borra la pantalla, se muestra un mensaje de agradecimiento y el bucle finaliza estableciendo la variable `run` en `false`. Si se selecciona una opción que no aparece en el menú, el sistema simplemente informará que no es válida.

Además, todo el bloque está contenido dentro de una estructura try-catch que gestiona errores, como cuando el usuario introduce datos que no son números. Si se produce un error, se muestra un mensaje amarillo que indica al usuario que debe volver a intentarlo.

Al final de cada ciclo, el sistema espera a que el usuario pulse una tecla antes de continuar, lo que le permite leer con tranquilidad la información presentada antes de volver a mostrar el menú.