ESCUELA POLITÉCNICA NACIONAL
ESCUELA DE FORMACIÓN DE TECNÓLOGOS

ALGORITMOS Y ESTRUCTURAS DE DATOS

ASIGNATURA:	Algoritmos y Estructuras de Datos
PROFESOR:	Ing. Lorena Chulde
FECHA:	04 – 02 - 2026
PERÍODO ACADÉMICO:	2025-B

Sistema Inteligente de Distribución de Paquetes

Nombres de los estudiantes:

Alexander Masapanta
Anguie Fierro

Table of Contents

Objetivo

El objetivo de este proyecto es la optimización de un sistema de distribución de paquetes, aplicando estructuras de datos y algoritmos vistos durante el semestre, como grafos, diccionarios, listas, archivos y colas de prioridad, para la creación del sistema PoliDelivery.Arquitectura del Sistema

ARQUITECTURA DE POLIDELIVERY

El proyecto desarrollado utiliza el lenguaje de programación Python, con el apoyo del entorno de desarrollo Visual Studio Code.

El sistema hace uso de la biblioteca heapq, la cual permite implementar una cola de prioridad, requisito fundamental para la ejecución del algoritmo de Dijkstra.

La arquitectura del sistema es modular, organizada en funciones que separan la lógica de usuarios, rutas, algoritmos y manejo de archivos.

Frontend

Desde el punto de vista visual, el sistema presenta una interfaz por consola, iniciando con un menú principal que permite:

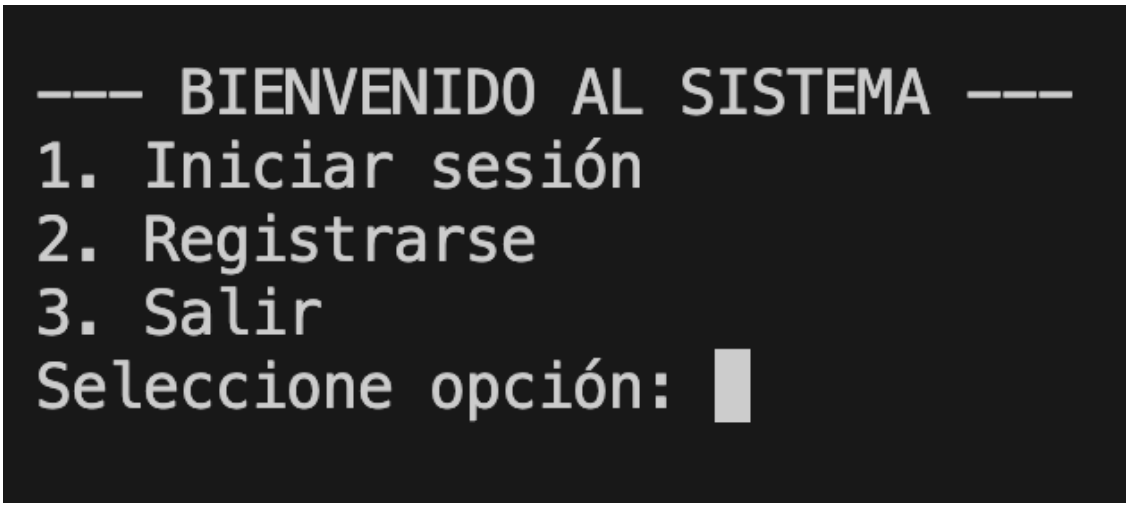

Figure 1Menu

Dependiendo de las credenciales ingresadas, el usuario accede a un menú personalizado según su rol.

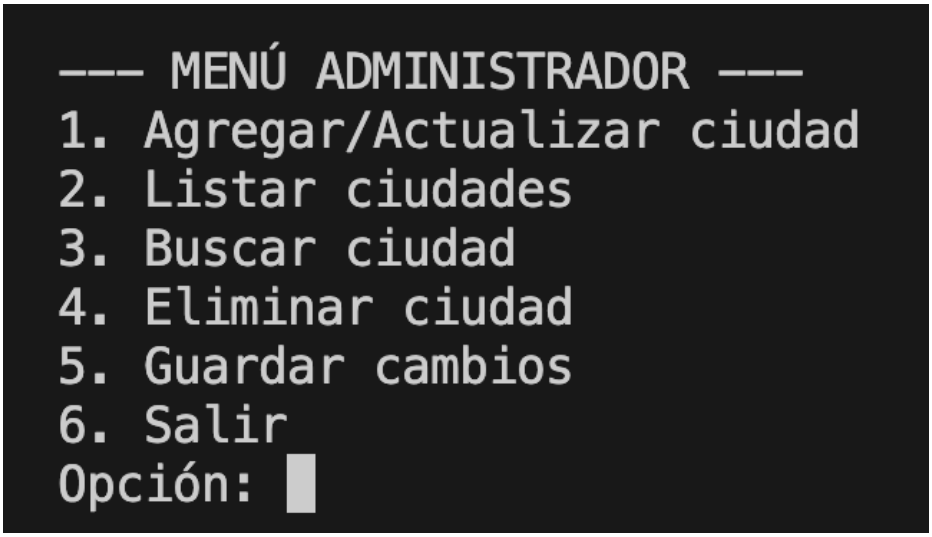

Figure 2M.ADM

Por otro lado temenos el menu de cliente, tiene las opciones de ver un mapa con las rutas El cliente puede:

- Visualizar el mapa de rutas
- Consultar la ruta óptima
- Explorar zonas organizadas por regiones
- Seleccionar centros de distribución
- Guardar su selección de rutas

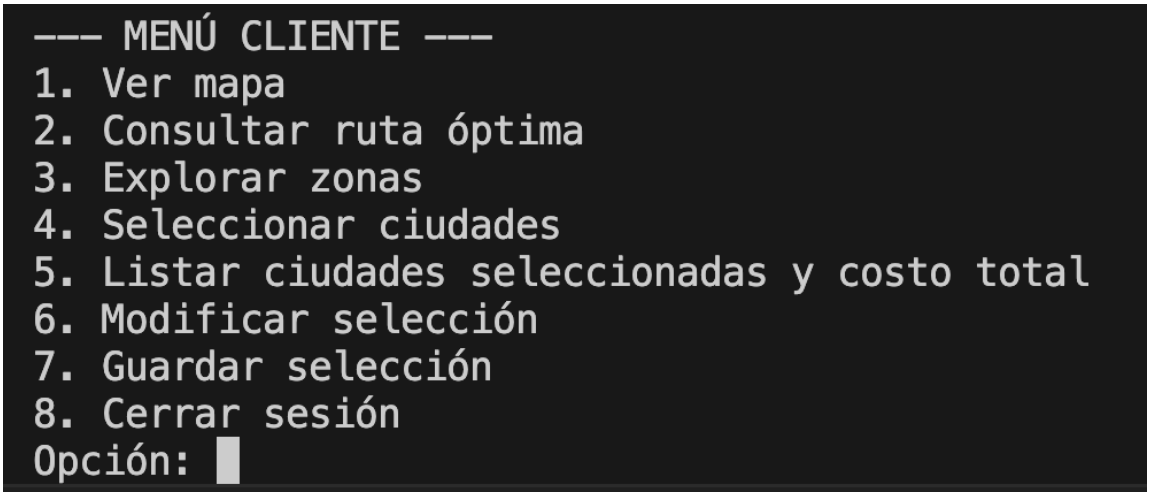

Figure 3M.Cliente

Backend del Sistema

RUTAS

Las rutas disponibles para la distribución se almacenan en el archivo centros.txt, el cual contiene el origen, destino, distancia y costo.
Adicionalmente, se utiliza un diccionario de zonas dividido por regiones y subregiones, que permite mostrar de forma estructurada el alcance del sistema PoliDelivery.

```python
def crear_archivo_centros():
    with open("centros.txt", "w") as archivo:
        archivo.write("Guayaquil,Panecillo, 300.1, 50\n")
        archivo.write("Quito, Cuenca, 360.3, 60\n")
        archivo.write("Napo, Loja, 286.9, 70\n")
        archivo.write("Ibarra, Yahuarcocha, 408.5, 40\n")
        archivo.write("Manta, Tena, 267.7, 65\n")
        archivo.write("Otavalo, Latacunga, 353, 30\n")
        archivo.write("Santo Domingo, Manta, 566, 35\n")
        archivo.write("Carchi, Quito, 149.5, 25\n")
        archivo.write("Puyo, Riobamba, 250.4, 20\n")
        archivo.write("Guayaquil, Machala, 123, 28\n")
```

Figure 4C.Archivo

```python
zonas_de_entrega = {
    "Costa": {
        "Guayas": ["Guayaquil", "Playas"],
        "Manabí": ["Manta", "Puerto López"]
    },
    "Sierra": {
        "Pichincha": ["Quito", "Panecillo"],
        "Tungurahua": ["Baños de Agua Santa"],
        "Ibarra": ["Otavalo", "Yahuarcocha"]
    },
    "Oriente": {
        "Napo": ["Tena"],
        "Pastaza": ["Puyo"]
    }
}
```

Figure 5Zonas

GRAFO

Las rutas se almacenan en un grafo, donde:

- Cada nodo representa un centro de distribución
- Cada arista representa una conexión con su distancia y costo

El grafo se construye a partir de los datos leídos desde el archivo, permitiendo modelar correctamente el sistema de distribución.

```python
def lista_adyacencia_para_dijkstra(rutas):
    grafo_local = {}
    for origen, destino, distancia, costo in rutas:
        if origen not in grafo_local:
            grafo_local[origen] = {}
        if destino not in grafo_local:
            grafo_local[destino] = {}
        grafo_local[origen][destino] = {'distancia': distancia, 'costo': costo}
        grafo_local[destino][origen] = {'distancia': distancia, 'costo': costo}
    return grafo_local
```

Figure 6Rutas

Esta funcion permite agregar una nueva ruta, realizando la misma accion que la funcion anterior ,solo que esta la guarda en la variable local .

```python
def agregar_conexion(origen, destino, distancia, costo):
    if origen not in grafo:
        grafo[origen] = {}
    if destino not in grafo:
        grafo[destino] = {}
    grafo[origen][destino] = {'distancia': distancia, 'costo': costo}
    grafo[destino][origen] = {'distancia': distancia, 'costo': costo}
```

Figure 7Agregar Rutas

Algoritmo de Dijkstra

El algoritmo de Dijkstra permite calcular la ruta de menor costo entre dos centros de distribución , gracias a que:

1. Se inicializan todos los nodos con un costo infinito

2. El nodo de inicio se establece con costo cero

COLA

Se utiliza una cola de prioridad (heap) para seleccionar el nodo con menor costo acumulado

Al finalizar, el algoritmo retorna:

- El costo mínimo
- La distancia total
- La ruta seguida, reconstruida mediante el uso de predecesores

```
def _dijkstra(inicio, tipo_costo='costo'):
    distancias = {nodo: float('inf') for nodo in grafo}
    predecesores = {nodo: None for nodo in grafo}
    distancias[inicio] = 0
    cola_prioridad = [(0, inicio)]
    while cola_prioridad:
        costo_actual, nodo_actual = heapq.heappop(cola_prioridad)
        if costo_actual > distancias[nodo_actual]:
            continue
        for vecino, datos in grafo[nodo_actual].items():
            costo_arista = datos[tipo_costo]
            nuevo_costo = costo_actual + costo_arista
            if nuevo_costo < distancias[vecino]:
                distancias[vecino] = nuevo_costo
                predecesores[vecino] = nodo_actual
                heapq.heappush(cola_prioridad, (nuevo_costo, vecino))
    return distancias, predecesores
```

Figure 8 DIJKSTRA

Reconstrucción de la Ruta

La ruta se reconstruye recorriendo los predecesores desde el nodo destino hasta el nodo origen, invirtiendo el orden para mostrar el camino correcto.

```python

def consultar_ruta_optima(origen, destino):
    distancias_costo, predecesores = _dijkstra(origen, 'costo')
    distancias_dist, _ = _dijkstra(origen, 'distancia')
    if distancias_costo[destino] == float('inf'):
        return [], f"No hay ruta disponible de {origen} a {destino}."
    ruta = []
    actual = destino
    while actual:
        ruta.append(actual)
        actual = predecesores[actual]
    ruta.reverse()
    costo_total = distancias_costo[destino]
    distancia_total = distancias_dist[destino]
    mensaje = (
        "La mejor ruta es:\n"
        f"Ruta: {' -> '.join(ruta)}\n"
        f"Distancia total: {distancia_total:.2f} km\n"
        f"Costo total: ${costo_total:.2f} USD"
    )
    return ruta, mensaje

```

Figure 9 Ruta optima

Algoritmos de Búsqueda y Ordenamiento

Se utilizaron distintos algoritmos de ordenamiento según la funcionalidad requerida en el sistema:

- Ordenamiento Burbuja, aplicado para ordenar los elementos seleccionados por el cliente, ya que se trabaja con listas pequeñas y permite una implementación sencilla.
- Selection Sort, utilizado para mostrar los centros de distribución del sistema de forma ordenada, facilitando su visualización para el administrador.
- Merge Sort, empleado para ordenar los centros de distribución antes de realizar la búsqueda binaria, garantizando resultados correctos y eficientes.
- Quick Sort, implementado para optimizar el rendimiento del sistema al ordenar los centros cuando se manejan mayores volúmenes de datos.

Estos algoritmos permiten mejorar la consulta, organización y visualización de los centros de distribución, contribuyendo a un sistema más eficiente y fácil de usar.

```python
def burbuja(lista):
    n = len(lista)
    for i in range(n):
        for j in range(0, n-i-1):
            if lista[j].lower() > lista[j+1].lower():
                lista[j], lista[j+1] = lista[j+1], lista[j]
    return lista
def seleccion(lista):
    n = len(lista)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if lista[j].lower() < lista[min_idx].lower():
                min_idx = j
        lista[i], lista[min_idx] = lista[min_idx], lista[i]
    return lista
def merge_sort(lista):
    if len(lista) <= 1:
        return lista
    mid = len(lista)//2
    left = merge_sort(lista[:mid])
    right = merge_sort(lista[mid:])
    return merge(left, right)
def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i].lower() <= right[j].lower():
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

Figure 10 Algoritmos

Gestión de Archivos

El sistema utiliza archivos de texto para garantizar la persistencia de datos:

usuarios.txt: almacena los datos de los usuarios registrados

centros.txt: contiene las rutas preestablecidas del sistema

rutas-nombre-cliente.txt: guarda la selección de rutas del cliente

```
def registro_usuario():
    nombre = input("Ingrese su nombre y apellido: ").strip()
    while True:
        identificacion = input("Ingrese su identificación (10 dígitos): ").strip()
        if identificacion.isdigit() and len(identificacion) == 10:
            break
        print("Identificación inválida. Debe tener 10 dígitos.")
    while True:
        edad = input("Ingrese su edad (1-99): ").strip()
        if edad.isdigit() and 1 <= int(edad) <= 99:
            break
        print("Edad inválida.")
    while True:
        correo = input("Ingrese su correo (@gmail.com o @hotmail.com): ").strip()
        if (correo.endswith("@gmail.com") or correo.endswith("@hotmail.com")):
            try:
                with open("usuarios.txt", "r") as f:
                    if correo in f.read():
                        print("Correo ya registrado.")
                        continue
            except FileNotFoundError:
                pass
            break
        print("Correo inválido.")
    contrasena = input("Ingrese contraseña segura: ").strip()
```

Figure 11 Usuarios

Rol del Administrador

El rol administrador permite un manejo dinámico del sistema, brindando la capacidad de:

- Agregar
- Modificar
- Eliminar
- Guardar nuevas rutas

Esto asegura que el sistema pueda adaptarse a cambios futuros.

```
o = input("Origen: ").strip()
d = input("Destino: ").strip()
dist = float(input("Distancia: "))
costo = float(input("Costo: "))
agregar_conexion(o,d,dist,costo)
print(f"Conexión {o} ↔ {d} agregada/actualizada.")
```

Figure 12 ADM 2

```python
def actualizar_o_eliminar_ciudad():
    if not seleccion_cliente:
        print("No hay ciudades seleccionadas.")
        return
    nombre = input("Ciudad a modificar: ").strip()
    if nombre not in seleccion_cliente:
        print("No encontrada.")
        return
    accion = input("Actualizar (A) o Eliminar (E): ").upper()
    if accion == "E":
        seleccion_cliente.remove(nombre)
    elif accion == "A":
        nueva = input("Nuevo nombre: ").strip()
        if nueva in grafo:
            idx = seleccion_cliente.index(nombre)
            seleccion_cliente[idx] = nueva
        else:
            print("Ciudad no registrada.")
```

Figure 13 Modificar

```python
def guardar_centros():
    with open("centros.txt", "w") as f:
        for o, destinos in grafo.items():
            for d, datos in destinos.items():
                f.write(f"{o},{d},{datos['distancia']},{datos['costo']}\n")
    print("Cambios guardados en 'centros.txt'.")
```

Figure 14 Agregar

Conclusions

Este proyecto permitió aplicar los conocimientos adquiridos sobre estructuras de datos y algoritmos, integrándolos en la optimización de rutas. Mediante el uso de grafos, fue posible modelar correctamente los centros de distribución y sus conexiones. La organización jerárquica mediante árboles facilitó la exploración estructurada del sistema, mientras que el uso de archivos permitió que el sistema sea persistente.

En resumen, este proyecto representa una pieza clave en nuestro aprendizaje como desarrolladores, ya que fortalece la experiencia práctica y el razonamiento lógico aplicado a la resolución de problemas reales.