Aguayo-Tech Backend

26 de noviembre del 2023
Documentación backend

Carlos Humberto Ávila Sánchez

Sistema de Planificación de Rutas Logísticas

Modelos.

En el diseño del backend de nuestro sistema de gestión de rutas para entregas, hemos adoptado una estructura orientada a objetos mediante la implementación de clases, proporcionando un enfoque modular y organizado para gestionar la complejidad de los datos. Cada entidad clave, como "Camión", "Entrega" y "Ruta", se ha modelado como una clase independiente. Esto no solo facilita la organización y mantenimiento del código, sino que también permite un mayor control sobre la estructura de los datos.

Clase Nombre

La clase Name se utiliza para definir el nombre de una persona, en este caso, el nombre de un cliente.

Atributos:

- first (str): Primer nombre de la persona.
- last (str): Apellido de la persona.

Métodos:

- get_first(): Retorna el primer nombre.
- get_last(): Retorna el apellido.
- set_first(first: str): Establece el primer nombre.
- set_last(last: str): Establece el apellido.
- __eq__(other: object): Compara si dos objetos Name tienen el mismo apellido.
- __ne__(other: object): Compara si dos objetos Name tienen apellidos diferentes.
- __lt__(other: object): Compara si el apellido de un objeto Name es alfabéticamente menor que el de otro.
- __le__(other: object): Compara si el apellido de un objeto Name es alfabéticamente menor o igual que el de otro.
- __gt__(other: object): Compara si el apellido de un objeto Name es alfabéticamente mayor que el de otro.
- __ge__(other: object): Compara si el apellido de un objeto Name es alfabéticamente mayor o igual que el de otro.
- serialize(): Serializa el objeto en formato JSON.
- deserialize(name_json: str): Deserializa un objeto Name a partir de una cadena JSON.
- __str__(): Retorna una representación en cadena del objeto Name.

Constructor:

__init__(first: str = None, last: str = None): Constructor de la clase Name. Puede recibir dos parámetros o inicializar los atributos en None si no se proporcionan.

Clase Coordenada

La clase Coordinate se utiliza para representar direcciones en el sistema.

Atributos:

- x (float): Coordenada en el eje X.
- y (float): Coordenada en el eje Y.

Métodos:

- get_x(): Retorna la coordenada en X.
- get_y(): Retorna la coordenada en Y.
- set_x(x: float): Establece la coordenada en X.
- set_y(y: float): Establece la coordenada en Y.
- get_distance(): Obtiene la distancia respecto al origen.
- get_distance_to(other: object): Obtiene la distancia respecto a otro objeto Coordinate.
- get_point(): Retorna las coordenadas en forma de tupla.
- __eq__(other: object): Evalúa si las distancias respecto al origen son iguales.
- __ne__(other: object): Evalúa si las distancias respecto al origen son distintas.
- __lt__(other: object): Evalúa si la distancia al origen es menor respecto a la del otro objeto.
- __le__(other: object): Evalúa si la distancia al origen es menor o igual respecto a la del otro objeto.
- __gt__(other: object): Evalúa si la distancia al origen es mayor respecto a la del otro objeto.
- __ge__(other: object): Evalúa si la distancia al origen es mayor o igual respecto a la del otro objeto.
- serialize(): Serializa el objeto en formato JSON.
- deserialize(coordinate_json: str): Deserializa el objeto en formato JSON.
- __str__(): Retorna una representación en cadena del objeto Coordinate.

Constructor:

__init__(x: float = None, y: float = None): Constructor de la clase Coordinate. Puede recibir dos parámetros o inicializar los atributos en None si no se proporcionan.

Clase Producto

La clase Product se utiliza para definir productos que se enviarán.

Atributos:

- name (str): Nombre del producto.
- weight (int): Peso que ocupa el producto.

Métodos:

- get_name(): Retorna el nombre del producto.
- get_weight(): Retorna el peso que ocupa el producto.
- set_name(name: str): Establece el nombre del producto.
- set_weight(weight: int): Establece el peso que ocupa el producto.
- __eq__(other: object): Compara si los pesos de dos productos son iguales.
- __ne__(other: object): Compara si los pesos de dos productos son diferentes.
- __lt__(other: object): Compara si el peso del producto es menor que el de otro objeto.
- __le__(other: object): Compara si el peso del producto es menor o igual que el de otro objeto.
- __gt__(other: object): Compara si el peso del producto es mayor que el de otro objeto.
- __ge__(other: object): Compara si el peso del producto es mayor o igual que el de otro objeto.
- serialize(): Serializa el objeto en formato JSON.
- deserialize(product_json: str): Deserializa el objeto a partir de una cadena JSON.
- __str__(): Retorna una representación en cadena del objeto Producto.

Constructor:

__init__(name: str = None, weight: int = None): Constructor de la clase Producto. Puede recibir dos parámetros o inicializar los atributos en None si no se proporcionan.

Clase Vehículo

La clase Vehicle se utiliza para definir los vehículos que se utilizarán en las rutas.

Atributos:

- id (str): Identificador único del vehículo.
- payload (int): Límite de carga del vehículo.
- current_payload (int): Carga actual del vehículo.
- shipments (List[Shipment]): Lista de pedidos asignados al vehículo.

Métodos:

- get_id(): Retorna el ID del vehículo.
- get_payload(): Retorna el límite de carga del vehículo.
- get_shipments(): Retorna la lista de pedidos del vehículo.
- get_current_payload(): Retorna la carga actual del vehículo.
- set_id(id: str): Establece el ID del vehículo.
- set_payload(payload: int): Establece el límite de carga del vehículo.
- set_current_payload(current_payload: int): Establece la carga actual del vehículo.
- set_shipments(shipments: List[Shipment]): Establece la lista de pedidos del vehículo.
- fill_vehicle(shipments: List[Shipment]): Llena el vehículo con la mejor combinación de pedidos y retorna las coordenadas de los pedidos.
- __eq__(other: object): Compara si el límite de carga es igual al de otro vehículo.
- __ne__(other: object): Compara si los límites de carga son diferentes.
- __lt__(other: object): Compara si el límite de carga es menor al de otro vehículo.
- __le__(other: object): Compara si el límite de carga es menor o igual al de otro vehículo.
- __gt__(other: object): Compara si el límite de carga es mayor al de otro vehículo.
- __ge__(other: object): Compara si el límite de carga es mayor o igual al de otro vehículo.
- serialize(): Serializa el objeto en formato JSON.
- deserialize(vehicle_json: str): Deserializa el objeto a partir de una cadena JSON.
- __str__(): Retorna una representación en cadena del objeto Vehicle.

Constructor:

__init__(id: str = None, payload: int = None, shipments: List[Shipment] = None): Constructor de la clase Vehicle. Puede recibir tres parámetros o inicializar los atributos en None si no se proporcionan.

Clase Envío

La clase Shipment se utiliza para definir los envíos que se realizarán.

Atributos:

- client (Client): Cliente asociado al envío.
- product (Product): Producto asociado al envío.

Métodos:

- get_client(): Retorna el cliente del envío.
- get_product(): Retorna el producto del envío.
- set_client(client: Client): Establece el cliente del envío.
- set_product(product: Product): Establece el producto del envío.
- serialize(): Serializa el objeto para poder ser guardado en un archivo JSON.
- deserialize(shipment_dict: dict): Deserializa el objeto a partir de un diccionario obtenido desde un archivo JSON.
- __str__(): Retorna una representación en cadena del objeto Shipment.

Constructor:

__init__(client: Client = Client(), product: Product = None): Constructor de la clase Shipment. Puede recibir dos parámetros o inicializar los atributos en valores predeterminados si no se proporcionan.

Clase Ruta

La clase Route representa una ruta y contiene una lista de pedidos.

Atributos:

- shipments (List[Shipment]): Lista de pedidos asociados a la ruta.

Métodos:

- get_shipments() -> List[Shipment]: Retorna la lista de pedidos de la ruta.
- set_shipments(shipments: List[Shipment]) -> None: Establece la lista de pedidos de la ruta.
- __str__() -> str: Retorna una representación en cadena del objeto Route.

Constructor:

__init__(shipments: List[Shipment] = []) -> None: Constructor de la clase Route. Puede recibir un parámetro o inicializar el atributo en una lista vacía si no se proporciona.

Clase Cliente

La clase Client representa a un cliente identificado por su dirección y nombre.

Atributos:

- name (Name): Nombre del cliente.
- address (Coordinate): Dirección del cliente.

Métodos:

- get_name() -> Name: Retorna el nombre del cliente.
- get_address() -> Coordinate: Retorna la dirección del cliente.
- set_name(name: Name) -> None: Establece el nombre del cliente.
- set_address(address: Coordinate) -> None: Establece la dirección del cliente.
- __eq__(other: object) -> bool: Compara si tanto el nombre como la dirección del cliente son iguales.
- __ne__(other: object) -> bool: Compara si tanto el nombre como la dirección del cliente son distintos.
- __lt__(other: object) -> bool: Compara si el nombre del cliente es menor alfabéticamente que el de otro cliente.
- __le__(other: object) -> bool: Compara si el nombre del cliente es menor o igual alfabéticamente que el de otro cliente.
- __gt__(other: object) -> bool: Compara si el nombre del cliente es mayor alfabéticamente que el de otro cliente.
- __ge__(other: object) -> bool: Compara si el nombre del cliente es mayor o igual alfabéticamente que el de otro cliente.

- serialize() -> str: Serializa el objeto en formato JSON.

deserialize(client_json: str) -> None: Deserializa el objeto a partir de un formato JSON.

- __str__() -> str: Retorna una representación en cadena del objeto Client.

Constructor:

__init__(name: Name = None, address: Coordinate = None) -> None: Constructor de la clase Cliente. Puede recibir dos parámetros o inicializar los atributos en None si no se proporcionan.

Solución planteada.

En nuestra estrategia para la planificación logística, hemos diseñado una solución que aprovecha la presencia de dos vehículos con características distintas. Con el objetivo de optimizar la eficiencia en la entrega de pedidos, hemos dividido el mapa en dos secciones: la cercana y la lejana. Cada vehículo asume la responsabilidad de una sección específica, permitiendo así una asignación más focalizada y eficiente de recursos.

División del Mapa:

Hemos segmentado el mapa en dos áreas, identificadas como la sección cercana y la sección lejana, cada una asignada a un vehículo específico. Esta división se basa en la ubicación geográfica de los puntos de entrega y optimiza la cobertura de cada vehículo.

Representación mediante Coordenadas:

Todas las ubicaciones, ya sean puntos de entrega, el almacén o destinos intermedios, se representan mediante coordenadas en un plano cartesiano. Esto facilita la planificación de rutas y la coordinación espacial de los vehículos.

Responsabilidad de los Vehículos:

Cada vehículo se carga con todos los posibles envíos de su sección asignada mientras no exceda su límite de peso. Esta estrategia maximiza la capacidad de carga de cada vehículo, reduciendo la necesidad de múltiples viajes innecesarios.

Proceso de Entrega:

Una vez que un vehículo ha entregado todos los pedidos en su sección, regresa al almacén.
En caso de que haya más pedidos pendientes, el vehículo se recarga para su próxima salida. Este ciclo continúa hasta que se hayan entregado todos los pedidos planificados.

Esta solución simplificada busca equilibrar la carga de trabajo entre los dos vehículos y optimizar las rutas de entrega. La división del mapa y la coordinación basada en coordenadas facilitan la asignación eficiente de pedidos, minimizando el tiempo y los recursos requeridos para completar las entregas. Esta estrategia prioriza la eficiencia y la coordinación, maximizando la capacidad de carga de cada vehículo en su área designada.

Clasificación de envíos.

Esta solución para la planificación de rutas logísticas se fundamenta en un enfoque voraz para optimizar la asignación de pedidos a vehículos, maximizando la eficiencia del transporte. El procedimiento se centra en la gestión de pedidos, ordenándolos por pesos de menor a mayor, y seleccionando combinaciones de pedidos hasta que la suma de sus pesos alcance o supere el límite establecido para el vehículo.

Procedimiento:

- **Ordenamiento por Peso:**

  Inicia con la lista de pedidos y los ordena en función de sus pesos, de menor a mayor. Este paso establece la base para la estrategia voraz.

- **Selección de Pedidos:**

  Itera sobre la lista ordenada de pedidos, seleccionando la primera combinación de pedidos que cumple con la restricción del peso del vehículo.

- **Asignación de Rutas:**

  La combinación de pedidos seleccionada se asigna como una ruta para el vehículo correspondiente. Este proceso se repite hasta que no hay más pedidos por asignar o se excede el límite de carga del vehículo.

Algoritmo Voraz:

El algoritmo voraz adoptado en este contexto busca una solución óptima local en cada paso, seleccionando la mejor opción disponible en ese momento. El enfoque de seleccionar pedidos por peso permite maximizar la capacidad de carga del vehículo en cada asignación, reduciendo así el número total de rutas necesarias.

Es crucial destacar que este algoritmo se enfoca exclusivamente en determinar los pedidos que formarán parte de una ruta y no aborda la planificación detallada del camino de la ruta en sí. Este enfoque simplificado es efectivo para asignar eficientemente pedidos a vehículos según su capacidad de carga.

Ventajas:

- **Eficiencia:** El enfoque voraz permite una asignación rápida y eficiente de pedidos.
- **Simplicidad:** La simplicidad del algoritmo facilita su implementación y comprensión.
- **Optimización Local:** Cada paso busca la mejor solución local, contribuyendo a la eficiencia general.

Limitaciones:

Optimización Global: No garantiza la solución global óptima, ya que la optimización se realiza en pasos locales.

Cálculo de rutas.

Esta solución para el cálculo de rutas en el contexto logístico utiliza un enfoque voraz y la representación de puntos como nodos en un grafo. El proceso se inicia seleccionando el punto más lejano con respecto al origen, y a partir de este, se crea un grafo donde las aristas representan las distancias entre los puntos. El algoritmo avanza seleccionando el siguiente nodo más cercano en cada paso, garantizando que todos los puntos sean visitados antes de regresar al almacén para obtener una nueva combinación de envíos.

Procedimiento:

- **Selección del Punto de Partida:**

  Utilizando el algoritmo voraz previamente descrito, se selecciona el punto más lejano con respecto al origen como punto de partida para la ruta.

- **Creación del Grafo:**

  Se crea un grafo utilizando todos los puntos obtenidos por el algoritmo voraz como nodos. Las aristas se ponderan con las distancias euclidianas entre los puntos.

- **Asignación de Rutas:**

  Con un punto de partida determinado, se busca el siguiente nodo más cercano en el grafo. Este proceso se repite hasta que todos los puntos sean visitados.

- **Registro de Puntos Visitados:**

  Se mantiene un registro de los puntos visitados y los no visitados para garantizar que cada punto sea alcanzado una vez y solo una vez.

- **Regreso al Almacén:**

  Una vez que todos los puntos han sido visitados, se regresa al almacén para obtener una nueva combinación de envíos y repetir el proceso.

- **Algoritmo de Rutas:**

  Este enfoque utiliza un algoritmo voraz para determinar el punto de partida y seleccionar el próximo punto más cercano en cada paso. El uso de grafos y la representación de distancias entre puntos permite una planificación eficiente del camino de la ruta.

Ventajas:

- **Eficiencia de Distancia:** Utiliza distancias reales entre puntos para optimizar la eficiencia en las rutas.
- **Optimización Local:** El enfoque voraz garantiza soluciones óptimas locales en cada paso.

Limitaciones:

- **Optimización Global:** No asegura la solución global óptima en términos de la planificación total del sistema de rutas.

- **Complejidad del Grafo:** La creación del grafo y el cálculo de distancias pueden volverse intensivos en recursos para conjuntos de datos muy grandes.

Funciones principales

remove_shipments(shipments: List[Shipment], shipments_to_remove: List[Shipment]) -> List[Shipment]

- Elimina los pedidos que ya fueron asignados a un vehículo.

Parámetros:

- shipments: Lista de pedidos.
- shipments_to_remove: Lista de pedidos a eliminar.
-

Retorno:

- Lista de pedidos sin los pedidos que ya fueron asignados a un vehículo.

get_route(root: Coordinate, g: nx.Graph, vehicle: Vehicle, shipments: List[Shipment]) -> List[Coordinate]

- Calcula la ruta más corta entre un punto de partida y los demás puntos.

Parámetros:

- root: Punto de partida.
- g: Grafo con las coordenadas de los puntos.
- vehicle: Vehículo que realizará la ruta.
- shipments: Lista de pedidos.

Retorno:

- Lista de coordenadas con la ruta más corta.

get_next(g: nx.Graph, root: Coordinate) -> Coordinate

- Obtiene el siguiente punto más cercano a la raíz.

Parámetros:

- g: Grafo con las coordenadas de los puntos.
- root: Punto de partida.

Retorno:

- Siguiente punto más cercano a la raíz.

get_routes() -> Tuple[Route, Route]

- Obtiene las rutas de los vehículos ligeros y pesados.

Retorno:

- Ruta del vehículo ligero.
- Ruta del vehículo pesado.

calculate_route(shipments: List[Shipment], vehicle: Vehicle) -> Route

- Calcula la ruta de un vehículo.

Parámetros:

- shipments: Lista de pedidos.
- vehicle: Vehículo que realizará la ruta.

Retorno:

- Ruta del vehículo.