UNIT 19
DATA STRUCTURES & ALGORITHMS

YoTeLoLlevo S.L.

18 DE ENERO DE 2025

PABLO NICOLÁS SOTO IRAGO
Computer Science & AI

Contenido

Introducción

La empresa YoTeLoLlevo S.L. quiere modernizar sus operaciones, reemplazando su proceso manual de reparto por un sistema informatizado en Python. Este nuevo sistema incluye la recepción de órdenes, asegurando que las solicitudes provienen de zonas cubiertas, y la planificación de rutas, optimizando el reparto con el algoritmo de Dijkstra. Esto se hace buscando reducir los tiempos de entrega y maximizar la eficiencia del reparto de paquetes, mejorando la experiencia del cliente.

Objetivos

Desarrollar un sistema para la recepción y validación de órdenes

Garantizar que las solicitudes de entrega estén dentro del área cubierta por YoTeLoLlevo S.L., priorizando resultados rápidos y precisos para mantener la competitividad en el mercado.

Optimizar la planificación de rutas

Implementar el algoritmo de Dijkstra para minimizar los tiempos de reparto, mejorando la eficiencia operativa y la satisfacción del cliente.

Diseñar un simulador en Python

Crear una herramienta en Python para simular el servicio de recepción de órdenes, permitiendo medir y validar los tiempos de respuesta del sistema.

Justificar y documentar las decisiones técnicas

Realizar un análisis de las estructuras de datos y algoritmos utilizados, justificar su selección y detallar los pasos seguidos durante la implementación.

Validar las soluciones con datos de ejemplo

Probar el sistema de planificación de rutas y recepción de órdenes con datos simulados (cercanos a la realidad) para asegurar su funcionalidad y efectividad.

Marco teórico

Tipos de datos básicos

En el escenario propuesto, vamos a tener una gran variedad de datos. Según sus valores, usaremos los siguientes tipos de datos:

Numéricos (int y float)

- Usaremos enteros (int) para valores enteros como las coordenadas de los nodos, índices en bucles o en el tamaño de los nodos cuando se dibujan.
- Los valores “float” representan valores positivos o negativos decimales, y se usarán en distancias, pesos o cálculos matemáticos.

Booleanos (bool)

Se emplearán para representar estados lógicos, en este caso de forma implícita, como en condiciones “if”, en excepciones y control de flujo (try:) o en bucles con condiciones de salida (while true:).

Cadenas de caracteres (str)

Las cadenas serán fundamentales para almacenar datos como mensajes y texto estático, los nombres de ubicaciones, las claves de los diccionarios y cualquier texto relacionado con los pedidos y las ubicaciones.

Estructuras de datos

En cuanto a las estructuras de datos, es necesario incluir diferentes formas para organizarlos según sus características y sus valores de manera eficiente:

Diccionarios

Las coordenadas se almacenarán en diccionarios, donde se le asignara una key (nombre) y un valor (tupla de coordenadas x e y), siendo estos:

- Key: ‘Almacen’, ‘Puerta del Sol’, 'Alonso Cano', etc.
- Valores: Las coordenadas (x, y) de los nodos como (20, 20) o (10, 10).

Listas

Se usarán en la distribución y contenido de las zonas accesibles por el servicio de envíos, como ‘Centro’, ‘Chamberí’ o ‘Salamanca’. También se usará en asignar vecinos para dar colores diferentes a los nodos anexos.

Tuplas

De forma parecida a las listas o a los diccionarios, se almacenarán valores en listas de datos, pero en este caso serán datos inmutables, como las coordenadas de los nodos (dentro ya del diccionario coordenadas) o en el cálculo de distancias.

Desarrollo del proyecto

En el proceso de virtualización para la validación de órdenes de entrega y la optimización de rutas en YoTeLoLlevo S.L., se evaluaron diversas alternativas de algoritmos y estructuras de datos, considerando la eficiencia en el tiempo de ejecución, el uso de memoria y la adaptabilidad a las necesidades del sistema.

Validación de órdenes de entrega

Para validar que la dirección de entrega (ubicaciones) pertenecen al área cubierta por la empresa, se analizaron diferentes estructuras de datos. Aunque las listas son más flexibles, su complejidad de búsqueda lineal O(n) podría generar problemas y tiempos de espera grandes cuando el volumen de datos es elevado. Por ello, se optó por utilizar un diccionario, que permite búsquedas eficientes en tiempo constante O (1) en promedio.

En los algoritmos de búsqueda, aunque la búsqueda binaria en listas ordenadas O (logn) podría haber sido una alternativa, pero mantener la lista ordenada constantemente es poco eficiente e incrementaría la complejidad en escenarios dinámicos. El diccionario fue la elección ideal para esta tarea, al combinar búsquedas rápidas con la capacidad de gestionar datos asociados a cada dirección.

Optimización de rutas de entrega

Para la planificación de rutas, se han investigado diversos algoritmos de grafos (Ver en Análisis de algoritmos). He elegido el algoritmo de Dijkstra, por su capacidad para encontrar rutas más cortas desde un origen hacia múltiples destinos, lo que resulta adecuado para la planificación eficiente en un grafo ponderado. En este caso, se modelaron las ubicaciones como nodos y las calles/rutas como aristas, con pesos basados en el tiempo estimado de viaje calculado a partir de las distancias entre coordenadas (parecidas y proporcionales a las reales).

Aunque otros algoritmos, como A*, que ofrecen mejoras en ciertos contextos al utilizar una heurística, en este sistema específico se busca la precisión del cálculo mediante Dijkstra. La complejidad de O (V^2) con listas de adyacencia es aceptable por el tamaño moderado del grafo que representa el área de cobertura.

Análisis de riesgos

El análisis de riesgos para la implementación del sistema automatizado en YoTeLoLlevo S.L. se divide en tres áreas: riesgos técnicos, operativos y de seguridad:

Riesgos Técnicos

Rendimiento del sistema de validación

Si el sistema de validación es lento, podría impactar negativamente en la competitividad de la empresa y perder potenciales clientes.

Mitigación: Utilizar diccionarios en lugar de listas para las búsquedas, lo que garantiza tiempos promedio de búsqueda O (1), es decir, más rápidos. Esto asegura respuestas casi inmediatas incluso con un volumen elevado de datos.

Inexactitud en la optimización de rutas

Las rutas ineficientes aumentarán los tiempos de entrega y los costes operativos. Siempre se busca la eficiencia en el reparto.

Mitigación: Implementar un algoritmo de optimización de rutas (Dijkstra). Esto permite encontrar las rutas más cortas y óptima entre las ubicaciones cubiertas por el servicio, reduciendo tiempos de entrega y costos operativos.

Escalabilidad del sistema

El sistema debe soportar un crecimiento importante de datos sin afectar el rendimiento.

Mitigación: Utilizar diccionarios para modelar el grafo de ubicaciones y conexiones, lo que permite un acceso rápido a las adyacencias de los nodos. Además, el uso del algoritmo de Dijkstra asegura un rendimiento adecuado.

Riesgos Operativos

Capacitación insuficiente

Si los empleados no comprenden el funcionamiento del sistema, podrían cometer errores al procesar órdenes o planificar rutas.

Mitigación: Implementar un plan de capacitación integral, con talleres prácticos y documentación clara para garantizar que el personal domine el sistema.

Resistencia al cambio

Algunos empleados podrían mostrar rechazo hacia la automatización por temor a perder autonomía o empleo.

Mitigación: Mentalizar al personal desde el inicio del proyecto, destacando los beneficios del sistema automatizado, como la reducción de tareas repetitivas.

Riesgos de Seguridad

Vulnerabilidades en datos de clientes

La información sensible suele ser objetivo de ataques cibernéticos.

Mitigación: Implementar medidas de seguridad robustas, como el cifrado de datos en tránsito y en reposo. Como alternativa de seguridad, también se implementarán sistemas de autenticación multifactor para proteger el acceso.

Justificación técnica de herramientas y lenguajes

Para el desarrollo del sistema de YoTeLoLlevo S.L., se ha seleccionado Python como el lenguaje de programación principal. Hacer este programa con Python se basa en varias razones:

Facilidad de desarrollo y prototipado rápido

Python es el lenguaje perfecto para un proyecto como el de YoTeLoLlevo S.L. por su agilidad de programación y sencillez de aprendizaje. Python es óptimo para implementar y ajustar rápidamente las funcionalidades de recepción de órdenes y planificación de rutas.

Soporte para algoritmos y estructuras de datos

El programa utilizará el algoritmo de Dijkstra para trazar la ruta óptima en un grafo que muestra los nodos (ubicaciones) y las conexiones ponderadas entre ellos. Python ofrece herramientas y bibliotecas externas (como networkx) que simplifican la implementación del algoritmo. En este proyecto, se emplean diccionarios para modelar el grafo, tuplas para representar las coordenadas (x e y) y listas para almacenar datos como ubicaciones por zona. Estas estructuras de datos son eficientes y adecuadas para gestionar tanto la validación de órdenes como la planificación de rutas.

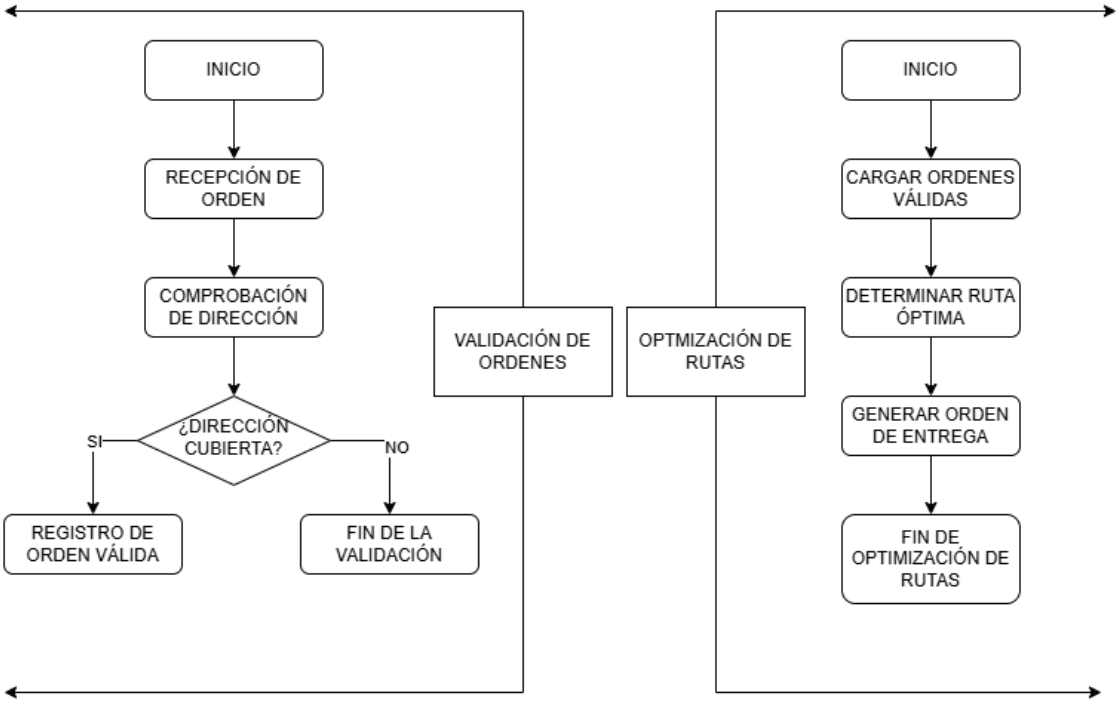
Amplia comunidad y soporte técnico

Python tiene una de las comunidades online más grandes del mundo del software. Esto me asegura acceso a todo tipo de documentación, ejemplos y soluciones, aplicables a este programa.

Escalabilidad y flexibilidad

El diseño de Python permite desarrollar sistemas que puedan escalar con el tiempo. En este proyecto, se busca la fragmentación del programa en funciones y módulos, asegurando que el sistema sea fácil de mantener y ampliar. Python también es lo suficientemente flexible para integrar optimizaciones en el futuro, ya sea con bibliotecas más avanzadas o mediante la implementación de componentes específicos en otros lenguajes como C++.

Diagrama de flujo



Análisis de algoritmos

Algoritmo	Descripción	Complejidad	Ventajas	Desventajas
Ordenación por Burbuja	Compara elementos adyacentes y los intercambia si están en el orden incorrecto. Esto se repite hasta que no haya más intercambios.	Mejor caso: O(n). Promedio y peor caso: $O(n^2)$	Sencillo de implementar y entender.	Su complejidad lo hace poco adecuado para listas largas.
Ordenación por Selección	Encuentra el elemento menor y lo coloca en la primera posición, luego el segundo más pequeño en la segunda, y así hasta el final.	Mejor, promedio y peor caso: $O(n^2)$	Simplifica el ordenamiento al no depender de la lista preordenada.	Su complejidad lo hace poco adecuado para listas largas.
Ordenación por Inserción	Toma cada elemento y lo coloca en su posición correspondiente en la sublista ordenada.	Mejor caso: O(n). Promedio y peor caso: $O(n^2)$	Muy eficiente para listas pequeñas o casi ordenadas.	Su complejidad cuadrática lo hace poco adecuado para listas largas.
Ordenación Shell	Es una mejora de la ordenación por inserción que compara elementos separados por un intervalo, reduciendo el intervalo con el tiempo.	Mejor caso: O(n log n) o O(n^(3/2)), dependiendo de la secuencia de intervalos Peor caso: $O(n^2)$	Más rápido que la ordenación por inserción en listas más grandes.	Puede ser más complejo de implementar debido a la selección de intervalos.
Quicksort	Utiliza el enfoque de "divide y vencerás" para dividir la lista en sublistas en torno a un pivote y ordenarlas recursivamente.	Mejor caso y promedio: O(n log n) Peor caso: $O(n^2)$	Muy rápido en la mayoría de los casos prácticos. Escala bien con gran cantidad de datos.	Puede ser ineficiente en el peor caso si el pivote no se elige bien.
Búsqueda Binaria	Algoritmo de búsqueda, no de ordenación. Requiere que la lista esté previamente ordenada y divide la búsqueda en mitades hasta encontrar el elemento deseado.	Mejor, promedio y peor caso: O(log n)	Muy rápido en listas ordenadas.	No es un algoritmo de ordenación, no ayuda en la optimización de la secuencia de rutas.

Dijkstra	Algoritmo utilizado para encontrar la ruta óptima en un grafo ponderado, ideal para optimizar rutas entre destinos.	Mejor, promedio y peor caso: $O(V^2)$	Eficiente para grafos con rutas definidas, útil en la optimización de rutas y planificación logística.	Puede ser ineficiente en grafos muy grandes si no se optimiza adecuadamente con estructuras como colas de prioridad.

He elegido el algoritmo de Dijkstra, ya que es el adecuado para el programa de planificación de rutas. Este algoritmo:

- Optimiza rutas en un grafo ponderado, encontrando la más corta entre destinos.
- Genera rutas eficientes según la prioridad o tiempo.
- Es ideal para la planificación logística, gestionando rutas entre múltiples destinos.

Análisis de los resultados

El programa permite calcular la ruta óptima entre el almacén de YoTeLoLlevo S.L. y un destino mediante el algoritmo de Dijkstra, visualizando el grafo con las conexiones entre nodos. También valida el destino, calcula el tiempo estimado de entrega y guarda un registro de las órdenes en un archivo:

Entrada y Validación de Destino

El programa permite al usuario introducir un destino. Si el destino no es válido (no está en el conjunto de nodos), el programa lo informa y pide una nueva entrada.

Cálculo de la Ruta Más Corta

Una vez se valida el destino en el sistema de validación de órdenes, el programa calcula la ruta más corta desde 'Almacen' usando el algoritmo de Dijkstra. La ruta se imprime en la consola, y los nodos por los que pasa también se listan. Por último se dibuja el grafo, con las ubicaciones como nodos, y las aristas como las conexiones.

En caso de que no haya una ruta posible entre los nodos, se informa al usuario.

Dibujo del Grafo

Si existe una ruta válida, el grafo con la ruta más corta se dibuja en una ventana emergente utilizando matplotlib. Los nodos están coloreados según su zona (Centro, Salamanca, Chamberí, etc.), y la ruta seleccionada es resaltada.

Tiempo Estimado

El tiempo estimado para ir del 'Almacen' al destino se calcula y se muestra en minutos y segundos.

Almacenamiento de Órdenes

Cada vez que se procesa una nueva orden de entrega, se añade una orden más en el archivo ordenes_envio.txt. Esto puede ser útil para realizar un seguimiento de las entregas.

Pseudocódigo

En anexos, en el archivo 'pseudocodigo.py'.

Código en Python

Implementación del algoritmo de Dijkstra en Python con ayuda de las librerías networkx, matplotlib.pyplot, itertools y math.

A continuación, se presenta el código:

datos.py

En anexos, en el archivo 'datos.py'.

main.py

En anexos, en el archivo 'main.py'.

planificacion_rutas.py

En anexos, en el archivo 'planificacion_rutas.py'.

Descripción del Grafo

El grafo creado en el código representa una red de nodos (ubicaciones de distintos barrios de Madrid) conectados por aristas que indican la distancia (o tiempo estimado de viaje) entre ellos. Los nodos están distribuidos por coordenadas geográficas aproximadas (en diferente escala), y se agregan aristas solo entre nodos que están a una distancia de 3 unidades o menos. Esto refleja una red de calles o puntos de interés conectados por caminos cercanos, lo que es una representación simplificada de la ciudad.

Conclusiones

YoTeLoLlevo S.L. quería informatizar el proceso de reparto, por ello ha desarrollado este proyecto para optimizar su sistema de reparto. Se ha creado un servicio de recepción de órdenes, donde se validan y se garantiza que solo se procesen las direcciones cubiertas por la red de envíos. Además, se ha implementado una solución para la planificación de rutas utilizando el algoritmo de Dijkstra, optimizando los tiempos de reparto.

El programa en Python ha permitido el funcionamiento informatizado del servicio de recepción de órdenes. Los resultados obtenidos en la planificación de rutas han mostrado mejoras significativas en la eficiencia del reparto ya que proporcionan una herramienta visual para el repartidor y el cliente.

Bibliografía

Se ha utilizado IA para el desarrollo de este proyecto.

- Ponce, J. (2021) Algoritmos de búsqueda. Disponible en: https://jahazielponce.com/algoritmos-de-busqueda/ (Accedido: 12 de octubre de 2024).

- Moreno, G. (2023) Búsqueda binaria. Disponible en: https://gabimoreno.soy/busqueda-binaria (Accedido: 12 de octubre de 2024).

- Universidad Nacional del Sur (2017) Algoritmos sobre grafos. Disponible en: https://cs.uns.edu.ar/~prf/teaching/AyC17/downloads/Teoria/Grafos-1x1.pdf (Accedido: 13 de octubre de 2024).

- VGA (s.f.) Algoritmo Dijkstra. Disponible en: http://atlas.uned.es/algoritmos/voraces/dijkstra.html (Accedido: 13 de octubre de 2024).

- Méndez Martínez, L., Rodríguez-Colina, E. y Medina Ramírez, R.C. (2013) TOMA DE DECISIONES BASADAS EN EL ALGORITMO DE DIJKSTRA. Disponible en: https://revistas.udistrital.edu.co/index.php/REDES/article/view/6357/7872 (Accedido: 13 de octubre de 2024).

- Gammafp (2017) Algoritmo A*. Youtube. Disponible en: https://youtu.be/X-5JMScsZ14(Accedido: 14 de octubre de 2024).

- FreeCodeCamp (2023) Algoritmos de ordenación explicados con ejemplos en

JavaScript, Python, Java y C++. Disponible en:

https://www.freecodecamp.org/espanol/news/algoritmos-de-ordenacion-explicados-con-ejemplos-en-javascript-python-java-y-c/ (Accedido: 28 de octubre de 2024).

- Alejandro G. Lillo. (2022). *Qué son los modelos de optimización de rutas*. Disponible en: https://xcloudy.es/que-son-los-modelos-de-optimizacion-de-rutas (Accedido: 14 de diciembre de 2024).

- StudySmarter. (s.f.). Optimización de rutas. Disponible en: https://www.studysmarter.es/resumenes/ingenieria/ingenieria-aeroespacial/optimizacion-de-rutas/ (Accedido: 14 de diciembre de 2024).

- Estefania C. Navone (2022). Algoritmo de la ruta más corta de Dijkstra: introducción gráfica. Disponible en: https://www.freecodecamp.org/espanol/news/algoritmo-de-la-ruta-mas-corta-de-dijkstra-introduccion-grafica/ (Accedido: 15 de diciembre de 2024).

- Koolac (2023) Youtube. Disponible en: https://www.youtube.com/watch?v=rldKl1CNx-A&list=PLGZqdNxqKzfYXTwYAZIlmjnQmrytCSR1J (Accedido: 16 de enero de 2025).

“Todo el trabajo debe estar respaldado por una investigación y referenciado a lo largo del texto mediante el sistema de referencia de Harvard y se deberá incluir una bibliografía en el mismo formato. El uso incorrecto de las referencias puede dar lugar a plagio si no se aplica correctamente.”

Anexos

pseudocodigo.py

```
                        # PSEUDOCÓDIGO

# Algoritmo Dijkstra (calcular_ruta_mas_corta):
# 1. Inicializar un conjunto de nodos no visitados
# 2. Establecer la distancia a todos los nodos como infinita, excepto al
nodo de inicio (establecer distancia a 0)
# 3. Mientras haya nodos no visitados:
#     a. Seleccionar el nodo con la distancia mínima
#     b. Para cada vecino del nodo seleccionado:
#         i. Calcular la distancia desde el nodo actual hasta el vecino
#         ii. Si la distancia calculada es menor que la distancia
previamente registrada, actualizar la distancia
#     c. Marcar el nodo como visitado
# 4. Repetir hasta que todos los nodos hayan sido visitados o se haya
encontrado el nodo de destino
# 5. Reconstruir el camino más corto siguiendo los nodos desde el destino
hasta el origen

# INICIO Validación de Órdenes

# Recibir_orden(orden):
#     Dirección <- Extraer_dirección(orden)
```

```
#     SI Dirección EN área_cubierta:
#         SI Dirección válida:
#             Registrar_orden(orden)
#             Retornar "Orden válida y registrada"
#         SINO:
#             Retornar "Dirección inválida"
#     SINO:
#         Retornar "Dirección fuera de cobertura"

# FIN Validación de Órdenes

# INICIO Optimización de Rutas

# Planificar_ruta(órdenes):
#     grafo <- Crear_grafo(órdenes)  // Construir el grafo de rutas

#     destinos <- Extraer_destinos(órdenes)  // Extraer destinos desde
las órdenes
#     rutas_ordenadas <- Ordenar_por_prioridad(destinos)  // Ordenar
destinos por prioridad (o tiempo)

#     ruta_optimizada <- Generar_rutas_optimizada(grafo,
rutas_ordenadas)  // Generar la ruta óptima utilizando el grafo

#     Retornar ruta_optimizada

# FIN Optimización de Rutas


                # COMPLEJIDAD LINEAL - NOTACION BIG O

# Algoritmo Dijkstra

# Inicializar un conjunto de nodos no
visitados                          # O(V)
# Establecer la distancia a todos los nodos como
infinita               # O(V)
# Mientras haya nodos no
visitados:                                      # O(V * log(V))
#     Seleccionar el nodo con la distancia
mínima                       # O(log(V))
#     Para cada vecino del nodo
seleccionado:                             # O(E)
#         Calcular la distancia desde el nodo actual hasta el
vecino     # O(1) por vecino
#         Si la distancia calculada es menor, actualizar la
distancia    # O(log(V)) por vecino
```

```
#     Marcar el nodo como
visitado                                          # O(1)
# Reconstruir el camino más
corto                                           # O(V)


# Complejidad final para Dijkstra:
# O(V)+O(V)+O(V·log(V))+O(E·log(V))+O(V)=O((V+E)·log(V)).

# Validación de Órdenes

# Extraer dirección de la
orden                                             # O(1) (acceso a un
atributo)
# Verificar si la dirección está en el área
cubierta                     # O(1) (si usamos un diccionario)
# Validar la
dirección                                                    # O(1)
# Registrar la orden (si
válida)                                            # O(1) (registro básico
en un sistema)

# Complejidad final para Validación de Órdenes: O(1).

# Optimización de Rutas

# Construir un grafo basado en las
órdenes                                 # O(V + E) (añadir nodos y aristas)
# Extraer destinos desde las
órdenes                                       # O(n) (recorrer todas las
órdenes)
# Ordenar destinos por
prioridad                                           # O(n \cdot \log(n))
(algoritmo de ordenación estándar)
# Calcular rutas optimizadas utilizando el
grafo                         # O((V + E) \cdot \log(V)) (Dijkstra para
cada destino)

# Complejidad final para Optimización de Rutas:

# O(V+E)+O(n)+O(n·log(n))+O((V+E)·log(V))=O((V+E)·log(V)+n·log(n)).
```

datos.py

```
# Coordenadas de los nodos
coordenadas = {
    'Puerta del Sol': (10, 10), 'Plaza Mayor': (12, 11), 'Calle Arenal':
(11, 9),
```

```
    'Gran Vía': (15, 10), 'Plaza de España': (17, 9), 'Calle Mayor': (13,
12),
    'Plaza de Oriente': (14, 8), 'Ópera': (13, 9), 'Calle Preciados':
(11, 11),
    'Callao': (14, 11), 'Centro' : (12, 9),

    # Barrio Chamberí
    'Quevedo': (20, 15), 'Iglesia': (21, 14), 'Alonso Cano': (23, 14),
    'Ríos Rosas': (25, 13), 'Canal': (20, 17), 'Islas Filipinas': (22,
18),
    'San Bernardo': (19, 13), 'Argüelles': (18, 11), 'Moncloa': (18, 19),
    'Guzmán el Bueno': (21, 16),

    # Barrio Salamanca
    'Velázquez': (28, 20), 'Serrano': (32, 19), 'Goya': (34, 18),
    'Príncipe de Vergara': (35, 17), 'Diego de León': (36, 16),
    'Lista': (33, 19), 'Manuel Becerra': (38, 15), 'Doctor Esquerdo':
(40, 14),
    'Avenida de América': (30, 22), 'Núñez de Balboa': (31, 20),

    # Barrio Chamartín
    'Chamartín': (40, 30), 'Plaza de Castilla': (45, 31), 'Pío XII': (43,
29),
    'Cuzco': (42, 28), 'Santiago Bernabéu': (40, 27), 'Hispanoamérica':
(43, 26),
    'Colombia': (41, 25), 'Concha Espina': (39, 26), 'Duque de Pastrana':
(46, 30),
    'Plaza de Lima': (41, 28),

    # Barrio Retiro
    'Atocha': (10, 30), 'Menéndez Pelayo': (11, 29), 'Pacífico': (12,
28),
    'Conde de Casal': (13, 27), 'Ibiza': (15, 25), 'Sainz de Baranda':
(16, 26),
    'Retiro': (14, 28), 'Prado': (11, 31), 'Neptuno': (12, 32),
'Cibeles': (13, 30)
}

# Zonas
zonas = {
    'Centro': [
        'Puerta del Sol', 'Plaza Mayor', 'Calle Arenal', 'Gran Vía',
'Plaza de España',
        'Calle Mayor', 'Plaza de Oriente', 'Ópera', 'Calle Preciados',
'Callao'
    ],
    'Chamberí': [
```

```
        'Quevedo', 'Iglesia', 'Alonso Cano', 'Ríos Rosas', 'Canal',
'Islas Filipinas',
        'San Bernardo', 'Argüelles', 'Moncloa', 'Guzmán el Bueno'
    ],
    'Salamanca': [
        'Velázquez', 'Serrano', 'Goya', 'Príncipe de Vergara', 'Diego de
León',
        'Lista', 'Manuel Becerra', 'Doctor Esquerdo', 'Avenida de
América', 'Núñez de Balboa'
    ],
    'Chamartín': [
        'Chamartín', 'Plaza de Castilla', 'Pío XII', 'Cuzco', 'Santiago
Bernabéu',
        'Hispanoamérica', 'Colombia', 'Concha Espina', 'Duque de
Pastrana', 'Plaza de Lima'
    ],
    'Retiro': [
        'Atocha', 'Menéndez Pelayo', 'Pacífico', 'Conde de Casal',
'Ibiza',
        'Sainz de Baranda', 'Retiro', 'Prado', 'Neptuno', 'Cibeles'
    ]
}

# Colores
colores = [
    "darkblue",
    "dodgerblue",
    "lightskyblue",
    "lightsteelblue",
    "peachpuff",
    "salmon",
    "coral",
    "orangered",
    "darkorange",
    "brown"
]
```

main.py

```
from planificacion_rutas import crear_grafo, calcular_distancia,
destino_validado as destino
from datos import coordenadas

# Crear el grafo utilizando el callejero proporcionado por
planificacion_rutas.py
G = crear_grafo(coordenadas)

# Función para calcular el tiempo estimado desde un destino al 'Almacen'
```

```
def calcular_tiempo_desde_destino(destino):
    # Verificar si el destino está en el callejero
    if destino not in coordenadas:
        print("El destino "+str(destino)+" no está cubierto por el
servicio.")
        return None

    # Calcular la distancia entre el destino y 'Almacen'
    distancia = calcular_distancia(coordenadas['Almacen'],
coordenadas[destino])

    # Calcular el tiempo estimado (ajustado por distancia)
    tiempo_estimado = distancia * 1.5 if destino in ['Centro',
'Salamanca'] else distancia * 1.2

    # Convertir el tiempo estimado en minutos y segundos
    tiempo_estimado_minutos = int(tiempo_estimado)
    tiempo_estimado_segundos = round((tiempo_estimado -
tiempo_estimado_minutos) * 60)

    return tiempo_estimado_minutos, tiempo_estimado_segundos

# Función para procesar las órdenes
def procesar_orden(destino):

    # Calcular el tiempo desde el destino al 'Almacen'
    tiempo_estimado = calcular_tiempo_desde_destino(destino)

    if tiempo_estimado:
        tiempo_estimado_minutos, tiempo_estimado_segundos =
tiempo_estimado

        print("Hemos generado una orden de entrega.")
        print("Tiempo estimado para ir desde el almacen a
"+str(destino)+": "
              + str(tiempo_estimado_minutos)+" minutos y
"+str(tiempo_estimado_segundos)+" segundos.\n")

# Llamar a la función para procesar la orden con el destino importado de
planificacion_rutas.py
procesar_orden(destino)

def guardar_orden(destino):
    # Leer el archivo para cargar los destinos y sus contadores
    with open("ordenes_envio.txt", "r") as archivo:
        lineas = archivo.readlines()

    # Crear un diccionario para almacenar los destinos y sus contadores
```

```
    destinos_dict = {}
    for linea in lineas:
        # Saltar líneas vacías o líneas con formato incorrecto
        if " : " not in linea:
            continue
        destino_nombre, contador = linea.strip().split(" : ")
        destinos_dict[destino_nombre] = int(contador)

    # Actualizar el contador para el destino
    if destino in destinos_dict:
        destinos_dict[destino] += 1
    else:
        destinos_dict[destino] = 1

    # Escribir de nuevo todos los destinos con los contadores
actualizados
    with open("ordenes_envio.txt", "w") as archivo:
        for destino_nombre, contador in sorted(destinos_dict.items()):
            archivo.write(str(destino_nombre)+" : "+str(contador)+"\n")

# Llamamos a la función para guardar la orden
guardar_orden(destino)
```

planificacion_rutas.py

```
import math
import networkx as nx
import matplotlib.pyplot as plt
from itertools import cycle
from datos import coordenadas, zonas, colores  # Importar datos externos
para nodos, zonas y colores

# Función para normalizar el texto
def normalizar_texto(texto):
    # Tabla de mapeo de caracteres con tildes a caracteres sin tildes
    mapeo_tildes = {
        'á': 'a', 'é': 'e', 'í': 'i', 'ó': 'o', 'ú': 'u',
        'Á': 'a', 'É': 'e', 'Í': 'i', 'Ó': 'o', 'Ú': 'u',
        'ñ': 'n', 'Ñ': 'n', 'ü': 'u', 'Ü': 'u'
    }
    # Reemplaza caracteres con tildes por sus equivalentes sin tildes
    texto_normalizado = ''.join(mapeo_tildes.get(c, c) for c in texto)
    # Elimina espacios innecesarios y convierte a minúsculas
    return texto_normalizado.strip().lower()

# Calcular distancia euclidiana (en línea recta) entre dos nodos
def calcular_distancia(coord1, coord2):
    # Usar la fórmula de distancia euclidiana entre dos coordenadas x e y
```

```
    return math.sqrt((coord1[0] - coord2[0])**2 + (coord1[1] -
coord2[1])**2)

# Crear grafo con networkx
def crear_grafo(coordenadas):
    grafo = nx.Graph()  # Crear un grafo vacío

    # Añadir nodos con posiciones
    for nodo, coord in coordenadas.items():
        grafo.add_node(nodo, pos=coord)  # Agregar cada nodo con su
posición

    # Añadir aristas según distancia
    nodos = list(coordenadas.keys())  # Obtener la lista de nodos
    for i, nodo1 in enumerate(nodos): # Iterar sobre cada nodo (indice,
nodo)
        for j, nodo2 in enumerate(nodos):
            if i != j:  # Evitar agregar aristas de un nodo consigo mismo
                distancia = calcular_distancia(coordenadas[nodo1],
coordenadas[nodo2])
                if distancia <= 3:  # Conectar nodos cercanos (radio
máximo de 3)
                    grafo.add_edge(nodo1, nodo2, weight=round(distancia *
1.2, 2))  # Redondear a 2 decimales
    return grafo

G = crear_grafo(coordenadas)  # Crear el grafo utilizando las coordenadas

# Función que calcula la ruta más corta
def calcular_ruta_mas_corta(grafo, origen, destino):
    # Usamos Dijkstra con networkx para encontrar la ruta más corta
    try:
        return nx.dijkstra_path(grafo, source=origen, target=destino,
weight='weight')
    except nx.NetworkXNoPath:  # Manejar el caso en el que no exista un
camino
        print("No existe una ruta entre el almacen y el destino.")
        return None

# Función para asignar colores a los nodos asegurando que nodos
conectados no tengan el mismo color
def asignar_colores(grafo):
    colores_ciclo = cycle(colores)  # Crear un ciclo infinito sobre la
paleta

    color_nodos = {}
    for nodo in grafo.nodes:
```

```
        vecinos = list(grafo.neighbors(nodo))  # Obtener los vecinos del
nodo
        # Asignar un color único a cada nodo
        for color in colores_ciclo:
            # Verificar que el color no se repita con los vecinos
            if all(color != color_nodos.get(vecino) for vecino in
vecinos):
                color_nodos[nodo] = color  # Asignar color al nodo actual
                break

    return color_nodos

color_nodos = asignar_colores(G)  # Asignar colores a los nodos del grafo

# Función para validar el destino
def validar_destino(destino):
    for nodo in coordenadas.keys():
        if normalizar_texto(nodo) == normalizar_texto(destino):
            return nodo  # Retornar el nodo correspondiente si se
encuentra una coincidencia
    return None  # Retornar None si no se encuentra el destino

# Añadir el nodo central 'Almacen' y conectarlo con el nodo más cercano
de cada zona
def agregar_nodo_almacen(grafo, coordenadas):
    coordenadas['Almacen'] = (20, 20)  # Asignar una posición fija para
el nodo 'Almacen'

    grafo.add_node('Almacen', pos=coordenadas['Almacen'])  # Añadir el
nodo 'Almacen' al grafo
    for zona, nodos_zona in zonas.items():
        print("Procesando zona: "+str(zona))
        # Encontrar el nodo más cercano al 'Almacen' en cada zona
        nodo_cercano = min(nodos_zona, key=lambda nodo:
calcular_distancia(coordenadas['Almacen'], coordenadas[nodo]))
          # Conectar con el nodo más cercano
        grafo.add_edge('Almacen', nodo_cercano,
weight=round(calcular_distancia(coordenadas['Almacen'],
coordenadas[nodo_cercano]), 2))

# Agregar el nodo 'Almacen'
agregar_nodo_almacen(G, coordenadas)

# Función para dibujar el grafo
def dibujar_grafo(grafo, color_nodos, ruta_seleccionada):
    pos = nx.get_node_attributes(grafo, 'pos')  # Obtener posiciones de
los nodos
```

```
    labels = nx.get_edge_attributes(grafo, 'weight')  # Obtener etiquetas
de las aristas
    labels = {k: round(v, 2) for k, v in labels.items()}  # Redondear
distancias a 2 decimales

    # Dibujar nodos con sus colores
    nx.draw_networkx_nodes(grafo, pos, node_size=500,
node_color=[color_nodos.get(nodo, 'grey') for nodo in grafo.nodes])

    # Dibujar las aristas (ruta óptima)
    for u, v in grafo.edges():
        if (u, v) in ruta_seleccionada or (v, u) in ruta_seleccionada:
#ruta_seleccionada en Linea 134
            # Resaltar ruta óptima
            nx.draw_networkx_edges(grafo, pos, edgelist=[(u, v)],
width=5, edge_color='black')
        else:
            # Dibujar con color predeterminado
            nx.draw_networkx_edges(grafo, pos, edgelist=[(u, v)],
width=1, edge_color=grafo[u][v].get('color', 'grey'))

    # Dibujar las etiquetas de los nodos
    nx.draw_networkx_labels(grafo, pos, font_size=10, font_weight='bold')

    # Resaltar la ruta seleccionada
    if ruta_seleccionada:
        path_edges = list(zip(ruta_seleccionada, ruta_seleccionada[1:]))
        nx.draw_networkx_edges(grafo, pos, edgelist=path_edges,
edge_color='black', width=6)

    plt.show()

# Lógica principal para obtener y validar el destino
while True:
    destino = input("Introduce el destino desde el Almacen: ")
    nodo_destino = validar_destino(destino)

    if nodo_destino: # Si el destino es válido
        print("La ruta hacia "+str(nodo_destino)+ " es válida.")
        break
    else:
        print("Destino no válido. Por favor, inténtalo nuevamente.")

# Calcular la ruta más corta
ruta_seleccionada = calcular_ruta_mas_corta(G, 'Almacen', nodo_destino)

# Imprimir la ruta más corta y los nodos por los que pasa
```

```
print("Ruta más corta desde el almacen a "+str(destino)+":
"+str(ruta_seleccionada))
print("Nodos por los que ha pasado la ruta:")

# Imprimir los nodos por los que pasa la ruta
for nodo in ruta_seleccionada:
    print("- "+str(nodo))

# Dibujar el grafo con la ruta seleccionada
if ruta_seleccionada:
    dibujar_grafo(G, color_nodos, ruta_seleccionada)
else:
    print("No se puede dibujar la ruta porque no existe un camino
válido.")

# Variable global para el destino validado
destino_validado = nodo_destino


# El algoritmo de Dijkstra encuentra la ruta más corta entre un nodo de
origen y otros nodos en un grafo ponderado.
# Comienza asignando una distancia de 0 al nodo origen y distancias
infinitas a los demás nodos.
# Luego, selecciona el nodo con la menor distancia no visitado, actualiza
las distancias de sus vecinos y lo marca como visitado.
# Repite este proceso hasta encontrar la ruta más corta hacia el destino.
```

Código en GitHub

https://github.com/PabloNSI/YoTeLoLlevo-SL.git