**Documentación del Proyecto: Optimización de Rutas para la Recolección de Productos en un Supermercado**

**1. Introducción**

El objetivo principal del proyecto es optimizar la ruta de recolección de productos en un supermercado. Para ello, se ha desarrollado un sistema que integra:

* **Un mapa del supermercado:** Representado en un archivo CSV, donde cada celda indica:
  + **0:** Espacio transitable.
  + **1:** Obstáculo.
  + **2:** Ubicación de gondolas (puntos de almacenamiento de productos).
* **Información de productos:** Almacenada en un archivo JSON (products.json), donde cada entrada contiene:
  + Un gondola\_id.
  + Las coordenadas (x e y) donde se encuentra la gondola.
  + Una lista de productos disponibles en dicha gondola.

Además, se han generado pedidos simulados en otro archivo JSON (pedidos.json) para probar la optimización de rutas, y se ha calculado la distancia entre productos basándose en la distancia entre gondolas.

Originalmente esto iba a ser datos reales extraidos de las practicas profesionales ya que es un Proyecto en el que se trabajo de manera real. La empresa se nego a concede acceso, por lo tanto se crearon scripts de python para generar datos ficticios simulando los datos reales con los que se ha trabajado.

**2. Desarrollo del Proyecto**

**2.1 Cálculo de Distancias entre Gondolas**

La primera fase consistió en calcular las distancias entre cada par de gondolas utilizando el algoritmo A\*.

* **Implementación del A\*:**  
  Se desarrolló una función astar() que utiliza la distancia Manhattan como heurística para encontrar la ruta más corta en el mapa.
* **Problema inicial:**  
  Muchos cálculos devolvían float('inf'), lo que indicaba que, según el algoritmo, algunas gondolas parecían no ser alcanzables.

**2.2 Complicaciones y Soluciones en el Cálculo de Rutas**

**Complicación 1: Restricción de Celdas Transitables**

* **Problema:**  
  La función get\_neighbors() sólo permitía moverse a celdas con valor 0, excluyendo así las celdas marcadas con 2 (las gondolas).
* **Solución:**  
  Se modificó la función para que cualquier celda cuyo valor no sea 1 (obstáculo) se considere transitable, permitiendo así que el algoritmo A\* utilice también celdas con valor 2.

**Complicación 2: Orden de Coordenadas (x e y)**

* **Problema:**  
  Las coordenadas en el CSV se interpretaban de forma que la fila representaba y y la columna x, pero en algunos casos se detectaron desajustes (coordenadas fuera de los límites del mapa).
* **Solución:**  
  Se revisó la conversión de datos y se aseguró que en el JSON se almacenen correctamente:
  + x\_coordinate: corresponde al índice de columna.
  + y\_coordinate: corresponde al índice de fila.

**2.3 Generación de Pedidos Simulados**

Para probar la optimización de rutas, se desarrolló un script que genera 50 pedidos simulados, cada uno conteniendo entre 8 y 40 productos.

* **Procedimiento:**
  + Se leen los productos existentes en products.json.
  + Se seleccionan aleatoriamente productos de la lista, permitiendo duplicados si el pedido excede el número total de productos disponibles.
  + Los pedidos se guardan en un archivo pedidos.json.

**2.4 Cálculo de Distancias entre Productos**

La siguiente fase implicaba calcular las distancias entre productos individuales.

* **Enfoque inicial:**  
  Se planeó utilizar el algoritmo A\* para cada par de productos, tomando como posición la coordenada de la gondola a la que pertenecen.
* **Problema y optimización:**  
  Dado que varios productos pertenecen a la misma gondola, se optó por reutilizar las distancias ya calculadas entre gondolas (almacenadas en gondola\_distances.json):
  + Si dos productos pertenecen a la misma gondola, la distancia es 0.
  + Si provienen de gondolas diferentes, la distancia entre productos es la misma que la distancia entre las respectivas gondolas.
* **Resultado:**  
  Se genera un nuevo archivo product\_distances.json con un formato anidado similar al de las distancias entre gondolas, lo que permite su posterior uso en optimizadores de rutas como el TSP.

**3. Estado Actual y Próximos Pasos**

**Estado Actual**

* **Mapeo y carga de datos:**  
  Se cuenta con el CSV del mapa y el JSON de productos correctamente formateados.
* **Cálculo de distancias entre gondolas:**  
  Resuelto mediante A\* con las correcciones necesarias en la función de vecinos y en la interpretación de coordenadas.
* **Generación de pedidos simulados:**  
  Los pedidos se generan aleatoriamente y se almacenan en pedidos.json.
* **Cálculo de distancias entre productos:**  
  Utilizando las distancias pre-calculadas entre gondolas, se han generado las distancias entre productos.

**Próximos Pasos**

* **Integración con el optimizador de rutas (TSP):**  
  Utilizar product\_distances.json para alimentar un algoritmo que optimice la ruta de recolección según el pedido simulado.
* **Validación y pruebas en entornos reales:**  
  Realizar pruebas con datos reales del supermercado para afinar el modelo.
* **Optimización y mejoras en el rendimiento:**  
  Ajustar el código para mejorar la eficiencia, especialmente en el manejo de grandes cantidades de productos y pedidos.

**4. Conclusiones**

A lo largo del desarrollo del proyecto se han identificado y solucionado varios problemas críticos:

* Se resolvió el problema de celdas no transitables modificando la función get\_neighbors().
* Se ajustó el manejo de coordenadas para asegurar la correcta ubicación de gondolas y, por ende, de productos.
* Se optimizó el cálculo de distancias entre productos utilizando la información pre-calculada de distancias entre gondolas, evitando la necesidad de ejecutar múltiples cálculos de A\*.

Este proceso de iteración y solución de problemas ha permitido avanzar de forma significativa en la creación de un sistema robusto para la optimización de rutas en el supermercado. La integración de estos módulos sentará las bases para la implementación del optimizador de rutas (TSP) y futuras mejoras en el sistema.

Version 1 – MPV

Esta es la version inicial, soluciona el problema, de la manera mas sencilla possible. Utilizando TSP con una heuristica que cojo el nearest neighbor. Encapsulamos esta version como la MPV. Lo bueno esque aunque sencilla, la solucion esta desarollada siguiendo practicas de codigo. Por lo que el codigo es muy modular, organizado y limpio. De tal manera en las proximas versions podremos ir mejorando la solución implementando modulos adicionales sin cambiar la logica general del Proyecto. Podriamos considerar que esta version supone el esqueleto de la solución final.

Version 2

Imlplementado fuerza bruta para n < PROBANDO

Gran diferencia

Using brute-force TSP solver.  
Data saved to /Users/pablo/Documents/TFG/TFG/v2/output/optimized\_route.json  
Route: ['starting\_point', 'product 257', 'product 254', 'product 241', 'product 213', 'product 161', 'product 86', 'product 56', 'product 17', 'product 2', 'product 101', 'product 176', 'finishing\_point']  
Total distance: 280  
Execution time: 25.8410 seconds  
  
  
  
Using nearest neighbor TSP solver.  
Data saved to /Users/pablo/Documents/TFG/TFG/v2/output/optimized\_route.json  
Route: ['starting\_point', 'product 257', 'product 254', 'product 161', 'product 213', 'product 241', 'product 176', 'product 101', 'product 2', 'product 56', 'product 86', 'product 17', 'finishing\_point']  
Total distance: 334  
Execution time: 0.0063 seconds

A screenshot of a blue and yellow diagram

Description automatically generated

REFERENCES

<https://www.geeksforgeeks.org/traveling-salesman-problem-tsp-in-python/>

<https://www.youtube.com/watch?v=cY4HiiFHO1o&ab_channel=WilliamFiset>