CASH PICK UP & DELIVERY PROBLEM

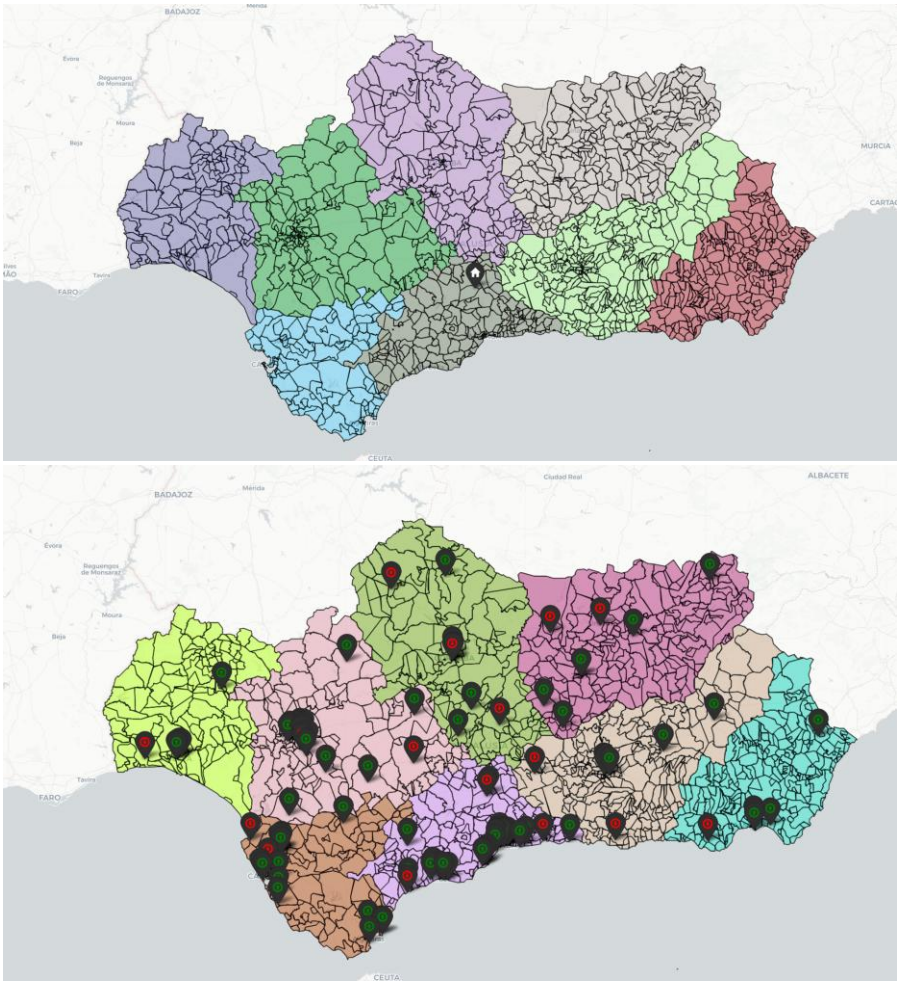
Arturo López – Damas Oliveres

DECIDE4AI

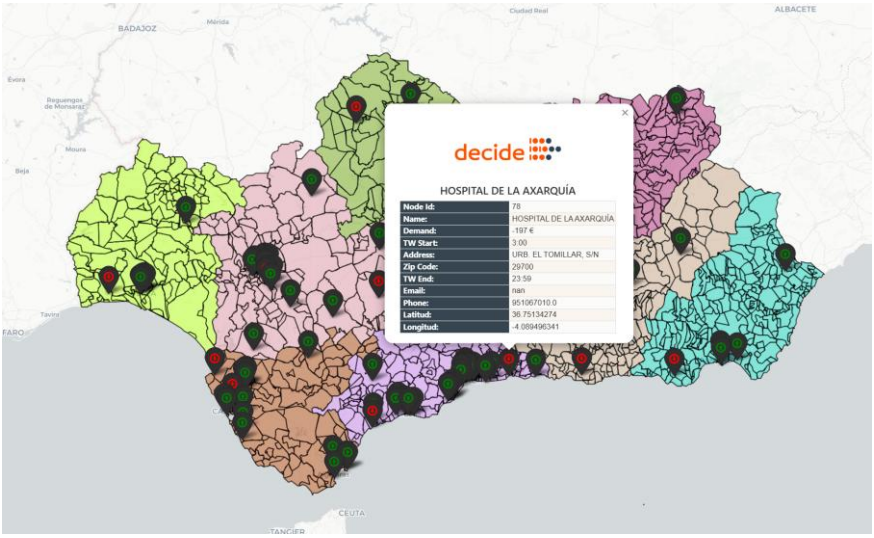
CONTENIDO

DEFINICIÓN DEL PROBLEMA

Nos contacta una empresa para hacer cargas de descargas de efectivo. Se espera repartir dinero entre los hospitales de Andalucía 114 en total, se ha elegido un depósito situado en Antequera, Málaga.



Por cada nodo, disponemos de una información detallada del mismo, los datos principales son la demanda y la ubicación.



CONSIDERACIONES

Se han considerado los siguientes aspectos a la hora de abordar el problema.

- **Máximo Stock:** Parámetro MAX_STOCK de capacidad de almacenaje en el depósito.
- **Stock Actual:** Un 80% del valor máximo de stock.
- **Capacidad de los vehículos:** Se ha programado para poder usar flota heterogénea, pero para la prueba se iguala la capacidad de la flota a VEHICLE_CAPACITY.
- **Máxima distancia:** Se ha establecido para este problema una distancia máxima de MAX_DISTANCE por vehículo.
- **Demanda Transferible:** Se ha permitido utilizar el dinero recogido en las picks_ups para entregarlo en las deliveries del mismo vehículo. Si alguna ruta necesita de dinero extra para hacer las entregas, se calcula el mínimo efectivo con el que debe empezar el camión para satisfacer la demanda. Esto se hace durante la construcción de la ruta, controlando que haya stock disponible.
- **No es necesario utilizar toda la flota** de entrada.

PROBLEMAS ENCONTRADOS MILP

He encontrando bastantes problemas al enfrentarme a los algoritmos exactos. Es la primera vez que los trabajo y he de decir que no estoy contento con el resultado obtenido. He entendido el problema bien y creo que he sido capaz de identificar las variables de decisión, las restricciones del problema y objetivos.

Al empezar a resolver el 'problema real' de los hospitales y he visto representadas las soluciones, me he dado cuenta que mi modelo no funcionaba en absoluto. Se que debo mejorar en los algoritmos exactos, pero no he podido ponerme al día en programación de este tipo con tan poco tiempo para la prueba técnica.

He intentando resolver el problema utilizando la librería 'pulp' y por las imágenes veo que sin éxito: https://coin-or.github.io/pulp/CaseStudies/a_transportation_problem.html#

He decido no tirar la toalla y crear un constructivo heurístico sencillo pero que funciona muy bien para el problema y es muy eficiente. Lo comento en detalle más adelante, pero genera soluciones que son razonables y se ajustan a las restricciones del problema. Aunque no es una solución exacta, el heurístico ha permitido obtener resultados prácticos en un tiempo casi instantáneo.

Los siguientes pasos serían mejorar la solución con 2opt, lin-kernighan, técnicas de inserción de un nodo en una ruta, reducir un vehículo…

Como no es lo que se me ha pedido no sigo implementando, pero es para enseñaros mi manera de trabajar y de pensar. Os dejo el acceso al repositorio por si queréis ver el trabajo realizado: https://github.com/artuloda/cash-pickup-delivery

LIMITACIONES MILP EN PROBLEMAS DE RUTAS

Aunque los problemas de rutas son de naturaleza combinatoria, los algoritmos MILP suelen ser ineficaces para tamaños grandes debido a la necesidad de explorar múltiples combinaciones de decisiones.

- **Complejidad Computacional Alta (NP-Hard)**: Al tener un número elevado de variables (como las variables binarias que indican las rutas de los vehículos) y restricciones (capacidad, distancia, incrementamos el tamaño del problema y puede hacerlo extremadamente difícil de resolver a medida que aumenta el número de vehículos y nodos el espacio de búsqueda crece exponencialmente.
- **Limitación en la Escalabilidad**: A medida que el tamaño del problema aumenta, la solución del MILP puede volverse prohibitiva en términos de tiempo de cálculo.
- **Eliminación de Subtours**: En problemas de rutas, uno de los desafíos típicos es el de los "subtours", que son soluciones no factibles en las que un vehículo se desplaza entre nodos sin visitar todos los nodos. Para eliminarlos, es necesario agregar restricciones adicionales, como las restricciones de eliminación de subrutas (MTZ), que son muy costosas en términos de complejidad computacional.

Resolver el Cash Pickup and Delivery Problem en formato MILP puede ser efectivo para instancias pequeñas y medianas, pero a medida que el tamaño del problema crece, las limitaciones de complejidad y tiempo de computación pueden hacer que MILP no sea la mejor opción. Por eso, suelo recurrir a heurísticas o metodologías híbridas para abordar problemas a gran escala de manera más eficiente.

RESOLUCIÓN PROBLEMA

DATOS ENTRADA:

- Nodos: Lista con los identificadores de los nodos.
- Demandas: Lista con la demanda de cada nodo, positivo pick up, negativo delivery.
- Distancias: Matriz de distancias entre los distintos nodos.
- Nodos no servidos: Conjunto de nodos no entregados.
- Latitudes: Lista con la latitud de cada nodo.
- Longitudes: Lista con la longitud de cada nodo.

RESTRICCIONES:

- Carga: Cada vehículo dispone de un valor de carga máxima permitida.
- Trayecto: Cada vehículo dispone de una distancia máxima permitida.

ALGORITMO PROPUESTO

Se ejecuta el constructivo tantas veces como se haya indicado en el parámetro MAX_ITERATIONS o si se alcanza el tiempo máximo de ejecución indicado en el parámetro MAX_TIME. Para el problema actual de 114 nodos, genera cada solución factible en unos 0,004s de media. Para generar 3000 soluciones factibles emplea 25s. Tras acabar las iteraciones o el tiempo, el algoritmo almacena la mejor solución, crea las métricas, genera el archivo con el detalle de ruta y crea un mapa que representa la solución.

Con el parámetro ALGORITHM_OPTION, seleccionamos entre el algoritmo heurístico y el exacto.

```python
    def construct(self):
        """
        Construct the solutions
        """
        self.context.logger.info("Constructing solutions...")
        start_time = time.time()
        iteration = 0
        while iteration < self.context.parameters.MAX_ITERATIONS and time.time() - start_time < self.context.parameters.MAX_TIME:
            start_time_iteration = time.time()
            # Create a new solution
            if self.context.parameters.ALGORITHM_OPTION == 1:
                solution = Solution(self.context, self.instance)
            else:
                solution = ExactSolution(self.context, self.instance)
            solution.solve()
            self.add_solution(solution)

            # Update best solution
            if solution.fitness < self.best_fitness:
                self.set_best_solution(solution)
            self.context.logger.info(f"Iteration {iteration} - Solution fitness: {solution.fitness}, Time: {time.time() - start_time_iteration:.4f}s")
            iteration += 1
        self.context.logger.info(f"Solution fitness: {self.best_solution.fitness}, Total time: {time.time() - start_time:.2f}s")
```

Para resolver el problema, se ha optado por un constructivo voraz aleatorizado que permita generar y evaluar soluciones de manera rápida.

```python

    def solve(self):
        """
        Solve the cash pickup and delivery problem using a greedy approach.
        """
        # Assign routes using a greedy approach
        for vehicle in range(self.context.parameters.n_vehicles):
            previous_node = 0
            while self.unserved:
                # Find the nearest feasible node
                candidate_nodes = self.find_feasible_nodes(previous_node, vehicle)
                if not candidate_nodes:
                    break  # No more feasible nodes for this vehicle

                # Select the next node to visit
                node, distance = self.select_next_node(candidate_nodes, vehicle)

                # Add node to route
                previous_node = self.add_node_to_route(node, vehicle, distance, self.instance.demands[node])

            # Return to depot
            if self.routes[vehicle]:
                self.total_distance += self.instance.distances[previous_node][0]
                self.current_distance[vehicle] += self.instance.distances[previous_node][0]
                # print(f"Vehicle {vehicle}, nodes: {self.routes[vehicle]}, distance: {self.current_distance[vehicle]},

        # Calculate storage stock and total cost
        self.storage_cost = self.instance.calculate_storage_cost(self.current_stock)
        self.fitness = self.instance.get_solution_value(self.total_distance, self.current_stock, len(self.unserved))
        # self.print_solution()

```

FUNCIONES OBJETIVO Y PENALIZACIONES:

Se ha optado por un enfoque multi-objetivo para resolver este problema. Tenemos tres objetivos principales que varían dinámicamente su importancia a la hora de añadir el siguiente nodo en la ruta.

La función select_next_node tiene como objetivo seleccionar el próximo nodo a visitar para un vehículo, priorizando aquellos que minimizan el costo de almacenamiento y maximizan la finalización del servicio. Al crear las rutas, si el vehículo no ha visitado ningún nodo, se selecciona un nodo al azar de los nodos candidatos utilizando self.random.get_random_choice(candidate_nodes). Esto asegura que el primer nodo sea elegido de manera aleatoria para diversificar las rutas.

Luego, se definen pesos para tres factores que influirán en la selección del siguiente nodo:

- **Peso de la distancia (weight_distance):** Este peso se asigna aleatoriamente entre 0.3 y 0.8 y representa la importancia de minimizar la distancia recorrida por el vehículo.
- **Peso de la penalización por stock (weight_stock_penalty):** Este peso aleatorio entre 0.3 y 0.5, se utiliza para penalizar situaciones donde el stock después de visitar un nodo excede el máximo permitido (self.context.parameters.MAX_STOCK). Esto ayuda a evitar que el vehículo recoja más de lo que puede almacenar, lo que podría incurrir en costos adicionales.
- **Peso dinámico para el retorno al depósito (dynamic_weight_return_to_depot):** Este peso se activa si el vehículo está casi lleno o ha recorrido casi la distancia máxima permitida. Se calcula un aleatorio entre 0.6 y 0.8 para ver si el vehículo está casi lleno. Si es así, se asigna un peso adicional para incentivar la vuelta al depósito, asegurando que el vehículo no se quede sin capacidad o exceda su límite de distancia.

```python
def select_next_node(self, candidate_nodes: list, vehicle: int) -> tuple:
    """
    Select the next node to visit: prioritize nodes that minimize storage cost and maximize service completion

    Args:
        candidate_nodes (list): Candidate nodes
        vehicle (int): Vehicle index
    Returns:
        tuple: Next node and distance
    """

    if len(self.routes[vehicle]) == 0:
        node, distance = self.random.get_random_choice(candidate_nodes)
        return node, distance

    # Define weights for each factor
    weight_distance = self.random.get_random_float(0.3, 0.8)
    weight_stock_penalty = self.random.get_random_float(0.3, 0.5)
    dynamic_weight_return_to_depot = 0

    # Determine if the vehicle is almost full or almost empty
    almost_full_vehicle_multiplier = self.random.get_random_float(0.6, 0.8)
    capacity_threshold = self.context.parameters.VEHICLE_CAPACITY * almost_full_vehicle_multiplier
    millage_threshold = self.context.parameters.MAX_DISTANCE * almost_full_vehicle_multiplier
    if self.current_capacity[vehicle] >= capacity_threshold and self.current_distance[vehicle] >= millage_threshold:
        dynamic_weight_return_to_depot = self.random.get_random_float(0.6, 0.8)

    # Normalize the weights
    total_weight = weight_distance + weight_stock_penalty + dynamic_weight_return_to_depot
    weight_distance_normalized = weight_distance / total_weight
    weight_stock_penalty_normalized = weight_stock_penalty / total_weight
    dynamic_weight_return_to_depot_normalized = dynamic_weight_return_to_depot / total_weight

    def evaluate_candidate(node_distance_tuple):
        node, distance = node_distance_tuple  # Node and distance
        stock_after_visit = self.current_stock + self.instance.demands[node]  # Stock after visit
        stock_penalty = max(0, stock_after_visit - self.context.parameters.MAX_STOCK)  # Stock penalty

        # Combine factors with weights
        return (
            weight_distance_normalized * distance +
            weight_stock_penalty_normalized * stock_penalty +
            dynamic_weight_return_to_depot_normalized * distance
        )

    node, distance = min(candidate_nodes, key=evaluate_candidate)
    return node, distance
```

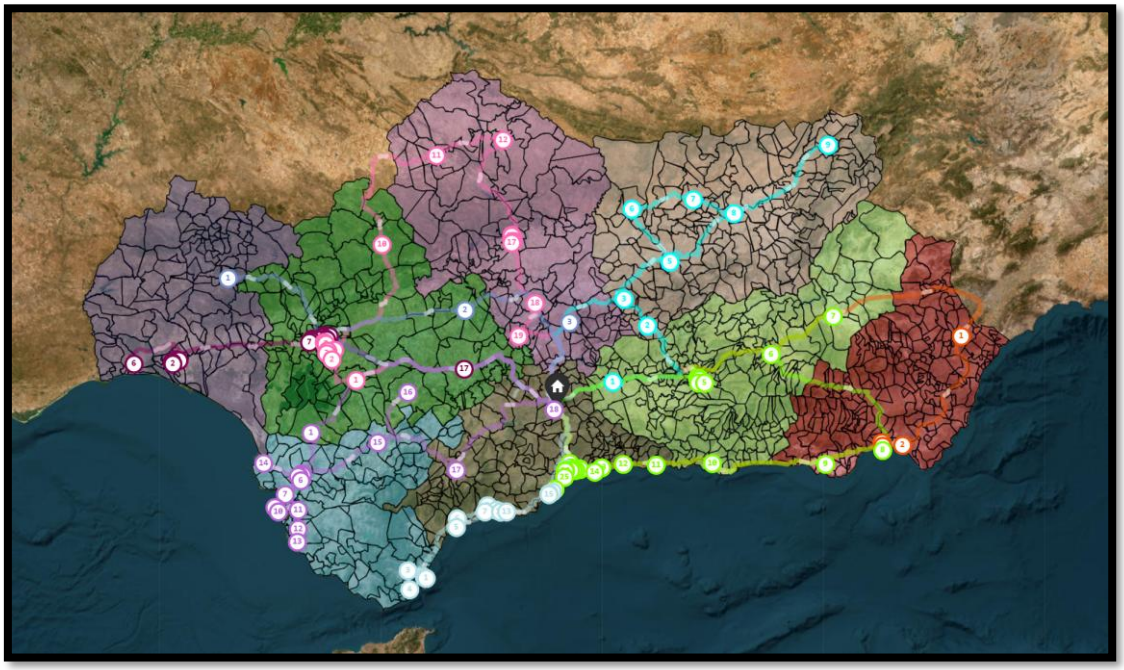
Estos pesos se normalizan para que su suma sea uno, asegurando que cada factor tenga una influencia proporcional en la decisión final. La función interna evaluate_candidate evalúa cada nodo candidato calculando una puntuación basada en la combinación ponderada de los tres factores mencionados. La puntuación se calcula como:

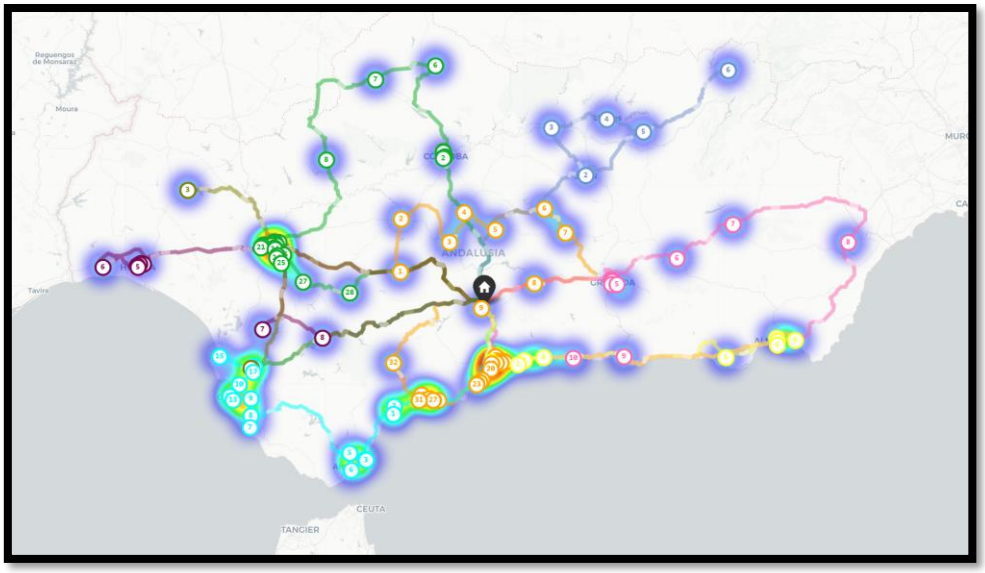
- **Distancia normalizada:** Multiplica la distancia al nodo por su peso normalizado.
- **Penalización por stock normalizada:** Calcula la penalización por exceso de stock y la multiplica por su peso normalizado.
- **Peso dinámico normalizado:** Multiplica la distancia por el peso dinámico normalizado, incentivando el retorno al depósito si es necesario.

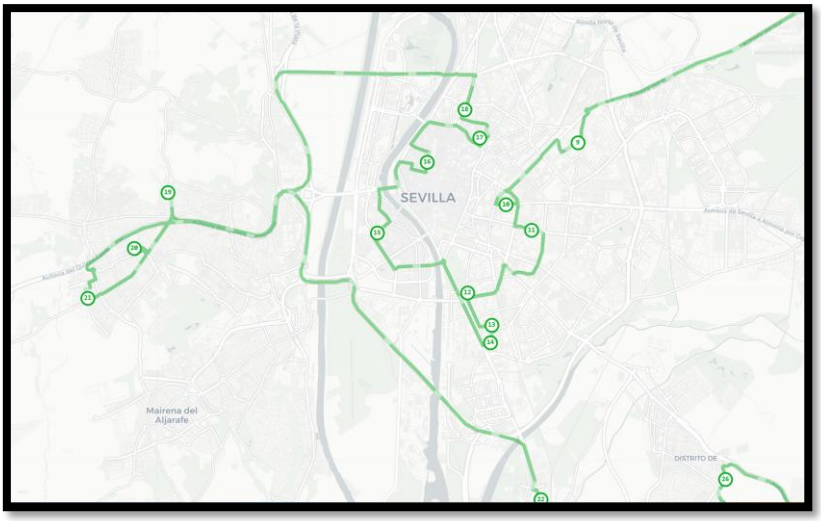
Finalmente, se selecciona el nodo con la puntuación más baja usando **min(candidate_nodes, key=evaluate_candidate)**. Esto asegura que el nodo seleccionado sea el más eficiente en términos de distancia, capacidad de almacenamiento y necesidad de retorno al depósito.

RESULTADOS Y EXPERIMENTOS

Se ha ejecutado el problema con distintas configuraciones y el algoritmo responde bien ante todas las situaciones y configuraciones. Adjunto algunas imágenes del resultado.




Reguengos de Monsaraz
Moura
Tavira
ANDALUCIA
CEUTA
TANGIER


SEVILLA
Mairena del Aljarafe
DISTRITO DE


BADAJOZ
Reguengos de Monsaraz
Moura
Tavira
FARO
ANDALUCIA
MURCIA
CARTAGENA
CEUTA
TANGIER
MOSTAGANEM
openstreetmap
Esri Satellite
Esri World Topographic Map
Carto BD Positron
Zip Codes
Heat Map
Nodes
Route 1 - Load: -459€ Stops: 4
Route 2 - Load: 114€ Stops: 17
Route 3 - Load: 1374€ Stops: 18
Route 4 - Load: 850€ Stops: 9
Route 5 - Load: -2643€ Stops: 19
Route 6 - Load: -76€ Stops: 3
Route 7 - Load: 1623€ Stops: 27
Route 8 - Load: 1399€ Stops: 17

EXPERIMENTOS

Se han realizado 4 experimentos con diferentes instancias para ver el rendimiento del algoritmo con 3000 llamadas al constructivo de manera secuencial y un MAX_STOCK de 15.000€. No se ha aplicado paralelismo.

- Prueba 1: 10 nodos, 1 vehículo, capacidad máxima 2000€, MAX_DISTANCE de 500km.
- Prueba 2: 30 nodos, 10 vehículos, capacidad máxima 2000€, MAX_DISTANCE de 500km.
- Prueba 3: 114 nodos, 20 vehículos, capacidad máxima 2000€, MAX_DISTANCE de 500km.
- Prueba 4: 114 nodos, 20 vehículos, capacidad máxima 2000€, MAX_DISTANCE de 350km.

Los archivos de salida se adjuntarán junto a este documento.