Guía de Apoyo, VRP

Gestión de operaciones

Profesores: Andre Carboni, Fernando Ordoñez
Auxiliares: Tomás Baez, Daniela Banda, Antonia Díaz, José Guzmán, Agustín Moreno, Diego Moreno, Antonia Oropesa, Nicolás Pacheco

Esta guía corresponde a un material de apoyo para realizar la Tarea recuperativa del curso. Cualquier consulta respecto a este material pueden preguntar por correo en ucursos a Diego Moreno.

Los contenidos son los siguientes:

Índice de Contenidos

1 VRP clásico **2**

2 VRP modelado con MTZ **4**

3 Heurística Greedy para TSP **6**

4 Técnicas de Mejoramiento **8**

1. VRP clásico

1.1. Definición del Problema

El Problema de Ruteo de Vehículos (VRP, por sus siglas en inglés, Vehicle Routing Problem) es un problema de optimización donde se debe determinar un conjunto de rutas para una flota de vehículos que parte y retorna a un depósito central, de manera que cada cliente sea visitado exactamente una vez, y se minimice el costo total del recorrido (usualmente medido en distancia, tiempo o costo económico).

1.2. Formulación Matemática

Sea $G = (V, A)$ un grafo dirigido, donde $V = \{0, 1, 2, \ldots, n\}$ representa el conjunto de nodos, con el nodo 0 como depósito y los nodos restantes como clientes. Cada arco $(i, j) \in A$ tiene un costo asociado c_{ij}. Se dispone de K vehículos disponibles para realizar las rutas.

1.3. Variables de decisión

Se define el conjunto de nodos $V = \{0, 1, 2, \ldots, n\}$, donde el nodo 0 representa el depósito y los nodos 1 a n representan clientes. Sea $K = \{1, 2, \ldots, m\}$ el conjunto de vehículos disponibles. Para cada par de nodos (i, j) y cada vehículo k se define:

$$x_{ijk} = \begin{cases} 1 & \text{si el vehículo } k \text{ recorre el arco } (i, j) \\ 0 & \text{en caso contrario} \end{cases}$$

1.4. Función objetivo

Minimizar el costo total de los arcos utilizados en todas las rutas de los vehículos:

$$\min \sum_{k \in K} \sum_{i \in V} \sum_{j \in V,\, j \neq i} c_{ij} x_{ijk}$$

1.5. Restricciones

1.5.1. Asignación de clientes

Cada cliente debe ser visitado exactamente una vez por un único vehículo:

$$\sum_{k \in K} \sum_{i \in V \setminus \{j\}} x_{ijk} = 1 \quad \forall j \in V \setminus \{0\}$$

1.5.2. Salida única desde el depósito

Cada vehículo puede salir a lo más una vez desde el depósito:

$$\sum_{j \in V \setminus \{0\}} x_{0jk} \leq 1 \quad \forall k \in K$$

1.5.3. Retorno único al depósito

Cada vehículo puede retornar al depósito a lo más una vez:

$$\sum_{i \in V \setminus \{0\}} x_{i0k} \leq 1 \quad \forall k \in K$$

1.5.4. Conservación de flujo por vehículo

Para cada cliente h, si un vehículo llega a h, debe salir desde h:

$$\sum_{i \in V \setminus \{h\}} x_{ihk} = \sum_{j \in V \setminus \{h\}} x_{hjk} \quad \forall h \in V \setminus \{0\},\ \forall k \in K$$

1.5.5. Eliminación de subtours

Para todo subconjunto $S \subseteq V \setminus \{0\}$, con $|S| \geq 2$:

$$\sum_{k \in K} \sum_{i \in S} \sum_{j \in S,\ j \neq i} x_{ijk} \leq |S| - 1$$

1.6. Naturaleza de las variables

$$x_{ijk} \in \{0, 1\} \quad \forall i, j \in V,\ i \neq j,\ \forall k \in K$$

2. VRP modelado con MTZ

2.1. Descripción general

Esta formulación del VRP es equivalente a la del modelo clásico presentado anteriormente, con la misma función objetivo, las mismas variables de decisión principales x_{ijk}, y las restricciones de asignación, flujo y estructura de ruta.

La única diferencia radica en la forma de evitar los subtours: en lugar de utilizar la formulación clásica de Dantzig–Fulkerson–Johnson (DFJ), se introduce una formulación alternativa basada en el enfoque de Miller–Tucker–Zemlin (MTZ), que emplea variables auxiliares para ordenar los nodos.

2.2. Variables auxiliares

Se introduce una variable continua auxiliar u_i para cada cliente $i \in V \setminus \{0\}$, que representa el orden relativo de visita del nodo en la ruta de su vehículo:

$$u_i \in [1, n] \quad \forall i \in V \setminus \{0\}$$

2.3. Restricción de eliminación de subtours (MTZ)

La formulación MTZ reemplaza las restricciones de subtours clásicas por la siguiente expresión:

$$u_i - u_j + n \cdot x_{ijk} \leq n - 1 \quad \forall i \neq j,\ i, j \in V \setminus \{0\},\ \forall k \in K$$

Donde: - n es el número de clientes (sin contar el depósito). - Si el vehículo k recorre el arco (i, j), entonces $u_j \geq u_i + 1$, forzando un orden creciente en la ruta. - Si $x_{ijk} = 0$, la restricción es automáticamente satisfecha.

2.4. Naturaleza de las variables auxiliares

$$u_i \in \mathbb{R} \quad \forall i \in V \setminus \{0\}$$

2.5. Comparación del tamaño de las restricciones de subtours

Una diferencia clave entre las formulaciones DFJ y MTZ para eliminar subtours radica en la cantidad de restricciones que generan:

- **DFJ (Dantzig–Fulkerson–Johnson):** Este enfoque requiere generar una restricción por cada subconjunto no vacío de clientes $S \subseteq V \setminus \{0\}$ tal que $|S| \geq 2$. En total, esto implica:

  $$\sum_{k=2}^{n} \binom{n}{k} = 2^n - n - 1 \quad \text{restricciones}$$

  donde n es el número de clientes (sin contar el depósito).

  Por ejemplo, si $n = 10$, se generan 1012 restricciones de subtour. Por ello, en la práctica, las restricciones DFJ se agregan dinámicamente como *lazy constraints*.

- **MTZ (Miller–Tucker–Zemlin):** Este enfoque introduce una variable continua u_i por cada cliente i, y genera una restricción para cada par ordenado de clientes distintos $i \neq j$.

  El número total de restricciones MTZ es:

  $$n(n-1)$$

  Por ejemplo, si $n = 10$, se generan 90 restricciones de subtour. Es decir, el enfoque MTZ tiene una cantidad de restricciones polinomial en n, lo que lo hace computacionalmente más manejable en instancias pequeñas o medianas.

3. Heurística Greedy para TSP

3.1. Descripción general

La heurística Greedy (o voraz) es una estrategia simple y rápida para construir una solución factible al Problema del Viajero (TSP), en el que se debe visitar un conjunto de ciudades exactamente una vez y regresar al punto de origen, minimizando la distancia total recorrida.

El enfoque consiste en comenzar desde un nodo inicial (generalmente el nodo 0), y en cada paso seleccionar el nodo más cercano que aún no haya sido visitado. El proceso se repite hasta haber visitado todos los nodos, cerrando finalmente el ciclo con un regreso al nodo de origen.

Existen variaciones del mismo, por ejemplo, al tener una capacidad el Viajero la ruta se finaliza cuando todos los nodos hayan sido satisfechos, o bien, cuando el Viajero no cuente con más oferta.

3.2. Pasos del algoritmo

1. Seleccionar un nodo inicial arbitrario (por ejemplo, el depósito o ciudad 0).
2. Marcarlo como visitado.
3. Mientras existan nodos no visitados:
   - Desde el nodo actual, seleccionar el nodo más cercano que no haya sido visitado.
   - Marcar ese nodo como visitado y moverse hacia él.
4. Volver al nodo inicial para cerrar el ciclo.

3.3. Ventajas y limitaciones

- **Ventajas:**
  - Muy rápida y fácil de implementar.
  - Útil como solución inicial para metaheurísticas (por ejemplo, 2-opt, Tabu Search, etc.).
- **Limitaciones:**
  - No garantiza encontrar la solución óptima.
  - Puede conducir a decisiones miopes: elegir el nodo más cercano no siempre resulta en un tour corto.
  - El rendimiento puede depender del nodo inicial.

3.4. Ejemplo ilustrativo

Supongamos un problema con 5 nodos y la siguiente matriz de distancias simétrica (en km):

$$D = \begin{bmatrix} - & 10 & 15 & 20 & 25 \\ 10 & - & 35 & 25 & 30 \\ 15 & 35 & - & 30 & 20 \\ 20 & 25 & 30 & - & 15 \\ 25 & 30 & 20 & 15 & - \end{bmatrix}$$

Comenzando en el nodo 0:

- Paso 1: desde 0, el más cercano es 1 (10 km).
- Paso 2: desde 1, el más cercano no visitado es 3 (25 km).
- Paso 3: desde 3, el más cercano no visitado es 4 (15 km).
- Paso 4: desde 4, el único no visitado es 2 (20 km).
- Regreso: desde 2 al nodo 0 (15 km).

Tour generado: $0 \to 1 \to 3 \to 4 \to 2 \to 0$ con distancia total de la ruta de: 10+25+15+20+15 = 85 km.

4. Técnicas de Mejoramiento

En esta sección vamos a ver dos técnicas de mejoramiento, 2-opt (k-opt) y relocate.

4.1. 2-opt

4.1.1. Descripción General

La heurística 2-opt es una técnica de búsqueda local ampliamente utilizada para mejorar una solución factible del TSP (como la generada por un algoritmo Greedy). Su objetivo es eliminar cruces innecesarios en las rutas, lo que suele reducir la distancia total del tour.

La idea central es que si dos aristas se cruzan, podemos recortar ese cruce e intercambiar los extremos, generando una ruta más corta. Esta operación se conoce como un 2-opt.

4.1.2. Descripción del procedimiento

Dado un tour inicial que recorre los nodos en cierto orden, la heurística 2-opt:

1. Recorre todos los pares de aristas no adyacentes del tour.
2. Para cada par de aristas $(i, i+1)$ y $(j, j+1)$, con $i < j$, evalúa si al eliminar esas dos aristas y reconectar los extremos opuestos (revirtiendo el segmento entre $i+1$ y j) se obtiene una mejor solución.
3. Si el nuevo tour es más corto, se realiza el cambio (intercambio 2-opt).
4. El proceso se repite hasta que no se encuentran más mejoras (óptimo local).

Supongamos que el tour actual es:

$$0 \to 1 \to 2 \to 3 \to 4 \to 0$$

Y que los arcos $1 \to 2$ y $3 \to 4$ se cruzan al ser dibujados en el plano. La heurística 2-opt detecta este cruce y propone:
- Eliminar: $1 \to 2$ y $3 \to 4$ - Reemplazar por: $1 \to 3$ y $2 \to 4$
Para ello, se invierte el segmento intermedio: $2 \to 3$ pasa a ser $3 \to 2$, resultando en el nuevo tour:

$$0 \to 1 \to 3 \to 2 \to 4 \to 0$$

4.1.3. Ventajas y limitaciones

- **Ventajas:**
  - Rápido y fácil de implementar.
  - Mejora significativamente soluciones iniciales malas como las de Greedy.
- **Limitaciones:**
  - Solo encuentra óptimos locales (puede estancarse).
  - Puede requerir múltiples reinicios o metaheurísticas para escapar de mínimos locales.

4.1.4. Implementación

Al momento de implementar 2-opt pueden programar uno que entregue la primera mejora, así como pueden hacer uno que entregue la mejor mejora. En este caso vamos a mostrar como hacer el primer caso.

Código 1: Heurística 2-opt (primera mejora) aplicada a un TSP, Python

```python
import matplotlib.pyplot as plt
import numpy as np
from scipy.spatial import distance_matrix

# Coordenadas de las ciudades
coords = np.array([
    [0, 0],   # nodo 0
    [1, 5],   # nodo 1
    [7, 2],   # nodo 2
    [5, 6],   # nodo 3
    [2, 1]    # nodo 4
])

# Matriz de distancias
dist_matrix = distance_matrix(coords, coords)

# Tour inicial (visita nodos en orden y regresa al inicio)
initial_tour = list(range(len(coords))) + [0]

# Funciones de 2-opt
def calcular_distancia_total(tour, dist_matrix):
    return sum(dist_matrix[tour[i], tour[i+1]] for i in range(len(tour) - 1))

def dos_opt(tour, dist_matrix):
    mejora = True
    mejor_tour = tour.copy()
    mejor_distancia = calcular_distancia_total(mejor_tour, dist_matrix)

    while mejora:
        mejora = False
        for i in range(1, len(tour) - 2):
            for j in range(i + 1, len(tour) - 1):
                nuevo_tour = mejor_tour[:i] + mejor_tour[i:j+1][::-1] + mejor_tour[j+1:]
                nueva_distancia = calcular_distancia_total(nuevo_tour, dist_matrix)
                if nueva_distancia < mejor_distancia:
                    mejor_tour = nuevo_tour
                    mejor_distancia = nueva_distancia
                    mejora = True
                    break
            if mejora:
                break
    return mejor_tour, mejor_distancia

# Aplicar 2-opt
mejor_tour, mejor_distancia = dos_opt(initial_tour, dist_matrix)
```

En el caso del código anterior se comienza con la ruta de la Figura 1 y resulta en una ruta como la Figura 2.

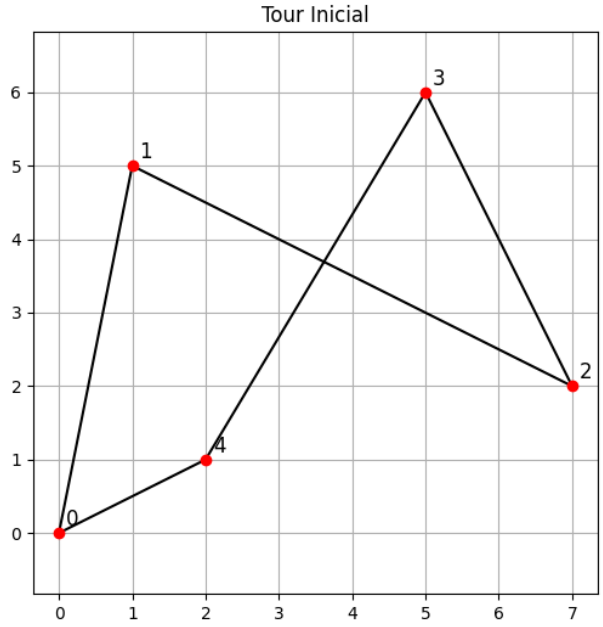

Figura 1: Tour inicial, previo a 2-opt

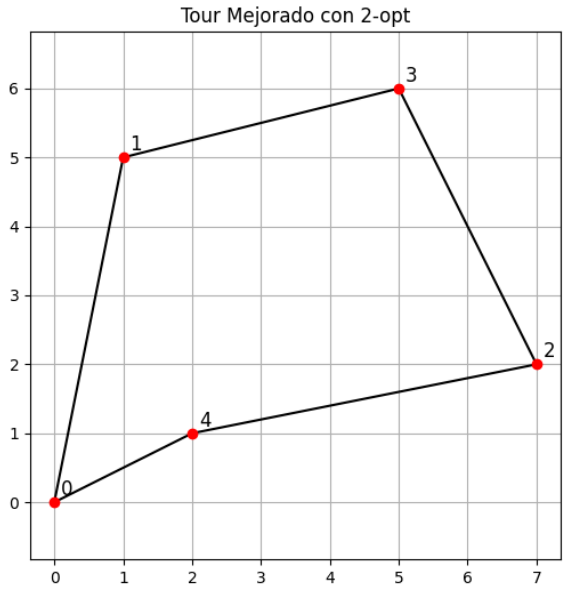

Figura 2: Tour final, mejorado con 2-opt

4.2. Relocate

4.2.1. Descripción General

La heurística **Relocate** consiste en mover un nodo desde su posición actual en una ruta hacia una posición distinta dentro la misma ruta o en otra ruta (en el caso de VRP). Es una técnica de búsqueda local que explora soluciones vecinas con un solo cambio, y se utiliza para reducir la distancia total del recorrido o balancear la carga entre vehículos.

Es particularmente útil después de una solución inicial generada por heurísticas como Greedy, ya que permite afinar el orden de visita sin realizar grandes modificaciones.

4.2.2. Aplicación en TSP

En el caso del TSP, donde existe una sola ruta cerrada, la operación consiste en:

1. Elegir un nodo i del tour (que no sea el depósito si aplica).
2. Eliminarlo de su posición actual.
3. Insertarlo en una nueva posición $j \neq i$, generando una nueva permutación del tour.
4. Evaluar el nuevo costo del tour. Si mejora, se acepta el cambio.

4.2.3. Aplicación en VRP

En el contexto del VRP, la heurística Relocate permite mover un nodo desde una ruta (vehículo) a otra, o dentro de la misma ruta, con el objetivo de mejorar el costo total del sistema, balancear la carga entre vehículos o liberar capacidad en rutas congestionadas.

Este operador puede tomar dos formas:

- **Relocate intra-ruta:** se mueve un nodo a otra posición dentro de la misma ruta.
- **Relocate inter-ruta:** se extrae un nodo desde una ruta R_1 y se inserta en otra ruta R_2 en la posición más conveniente.

Pasos generales del operador inter-ruta:

1. Seleccionar un nodo i de la ruta R_1, excluyendo el depósito.
2. Eliminar el nodo i de su posición actual en R_1.
3. Insertar el nodo i en una posición candidata en otra ruta R_2.
4. Evaluar el cambio: si mejora el costo total del sistema (o cumple un criterio como balance de carga), aceptar el movimiento.

Consideraciones adicionales:

- Si el VRP tiene restricciones de capacidad, se debe verificar que la nueva ruta no exceda el límite al insertar el nodo.

4.2.4. Ventajas

- Rápida de evaluar: sólo afecta un pequeño número de arcos del tour.
- Útil para ajustar la estructura fina del recorrido.
- Puede complementar bien a otras técnicas como 2-opt.

4.2.5. Limitaciones

- No elimina cruces directamente como lo hace 2-opt.
- Puede estancarse en óptimos locales si no se combina con otras estrategias.

4.2.6. Ejemplo ilustrativo intra-ruta

Supongamos el tour:

$$0 \to 1 \to 2 \to 3 \to 4 \to 0$$

Si aplicamos Relocate al nodo 3 y lo insertamos entre 1 y 2, el nuevo tour es:

$$0 \to 1 \to 3 \to 2 \to 4 \to 0$$

La heurística evalúa si este nuevo orden reduce la distancia total y, si es así, lo acepta como una mejora.

4.2.7. Ejemplo ilustrativo inter-ruta

Supongamos que tenemos dos rutas servidas por vehículos distintos:

$$R_1 = [0 \to 1 \to 3 \to 0] \qquad R_2 = [0 \to 2 \to 4 \to 5 \to 0]$$

Aplicamos la heurística Relocate inter-ruta al nodo 3, que actualmente se encuentra en la ruta R_1. El objetivo es insertarlo en una posición de R_2 donde el aumento en costo sea mínimo (idealmente, se reduzca el costo global del sistema).

Supongamos que insertamos el nodo 3 entre los nodos 4 y 5 en R_2. Las rutas resultantes quedan:

$$R_1' = [0 \to 1 \to 0] \qquad R_2' = [0 \to 2 \to 4 \to 3 \to 5 \to 0]$$

Luego se evalúa el impacto total de este movimiento sobre el costo global (distancia total, equilibrio entre rutas, factibilidad de capacidad, etc.). Si el cambio mejora la solución o cumple con los objetivos del sistema, se acepta como una mejora.

4.3. Implementación

Para la implementación se va a trabajar con un VRP con 2 vehículos y solo una iteración de Relocate.

Código 2: Relocate inter-ruta para VRP con dos rutas

```python
import numpy as np
from scipy.spatial import distance_matrix

# Coordenadas de los nodos
coords = np.array([[0, 0], [1, 3], [2, 8], [6, 7], [7, 3], [3, 1]])

# Matriz de distancias euclidianas
dist_matrix = distance_matrix(coords, coords)

# Dos rutas iniciales
ruta1 = [0, 1, 3, 0]
ruta2 = [0, 2, 4, 5, 0]

def calcular_costo_ruta(ruta, dist_matrix):
    return sum(dist_matrix[ruta[i], ruta[i+1]] for i in range(len(ruta)-1))

def calcular_costo_total(rutas, dist_matrix):
    return sum(calcular_costo_ruta(r, dist_matrix) for r in rutas)

def relocate_inter_ruta(ruta1, ruta2, dist_matrix):
    mejor_costo = calcular_costo_total([ruta1, ruta2], dist_matrix)
    mejor_r1, mejor_r2 = ruta1.copy(), ruta2.copy()

    for i in range(1, len(ruta1)-1):  # evitar depósito en extremos
        nodo = ruta1[i]
        r1_sin_nodo = ruta1[:i] + ruta1[i+1:]

        for j in range(1, len(ruta2)):  # posiciones posibles de inserción
            nueva_r2 = ruta2[:j] + [nodo] + ruta2[j:]
            nuevo_costo = calcular_costo_total([r1_sin_nodo, nueva_r2], dist_matrix)

            if nuevo_costo < mejor_costo:
                mejor_costo = nuevo_costo
                mejor_r1 = r1_sin_nodo
                mejor_r2 = nueva_r2

    return mejor_r1, mejor_r2, mejor_costo

# Aplicar relocate
nueva_ruta1, nueva_ruta2, nuevo_costo = relocate_inter_ruta(ruta1, ruta2, dist_matrix)

nueva_ruta1, nueva_ruta2, nuevo_costo
```

Los resultados de esta implementación son los siguientes:

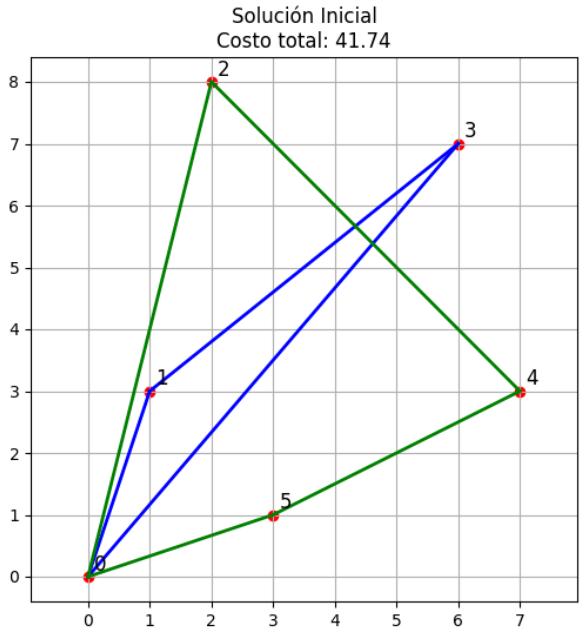

Figura 3

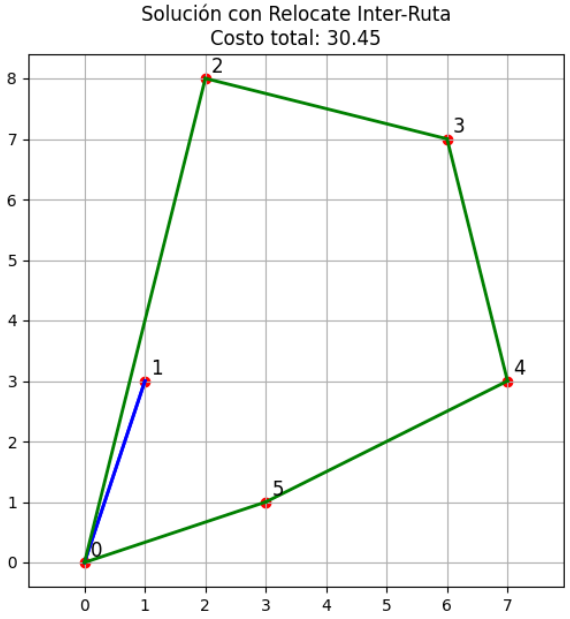

Figura 4

Es posible observar además que, al aplicar nuevamente el operador Relocate, el nodo 1 podría ser transferido desde la ruta azul hacia la ruta verde, lo que permitiría reducir aún más el costo total del sistema. Este tipo de movimientos sucesivos ilustra cómo las mejoras locales pueden acumularse gradualmente, y refuerza la utilidad de aplicar iterativamente operadores simples como Relocate o 2-opt en combinación con criterios de evaluación eficientes.