# Tarea 2
# Recocido simulado con el problema de $k$ vendendores viajeros

Mario Emilio Jiménez Vizcaíno
A01173359@itesm.mx

## 1 INTRODUCCIÓN

El problema del vendedor viajero (o mejor conocido como TSP) es un problema famoso por ser muy fácil de describir y difícil de resolver. [2] El problema puede plantearse de forma sencilla: si un vendedor ambulante desea visitar exactamente una vez cada una de una lista de ciudades y luego regresar a la ciudad inicial, ¿cuál es la ruta más corta que el vendedor puede tomar?

En esta tarea se aborda una variante de este problema, conocida como "el problema de múltiples vendedores viajeros", o por sus siglas en inglés, mTSP. Esta variate es una generalización del problema, en el que se permite utilizar más de un vendedor en la solución. Por sus características, este problema usualmente es más apropiado a aplicaciones en la vida real.[1]

## 2 METODOLOGÍA

Para esta tarea implementé una solución del problema usando el algoritmo de recocido simulado, en el que utilizando parámetros controlados como la temperatura y la longitud de cadena, se corren cadenas de Markov para generar soluciones mejores, o posiblemente no tan mejores, dependiendo de la temperatura del algoritmo.

Algunas de las decisiones que tomé para realizar esta implementación se describen a continuación.

### 2.1 Representación de los individuos

Para representar las $n$ ciudades opté por generar una lista de 0 a $n-1$, en la que cada número indica el índice de la ciudad a la que el viajero se moverá a continuación. Para asegurarme que las listas fueran soluciones válidas al problema consideré dos restricciones principales:

- Que la función de mutación sólo intercambie números de lugar en vez de generar nuevos.
- Que el número de ciclos que formaban los caminos fuera igual al número de vendedores viajeros. Esta validación se lleva a cabo dentro de la función `is_solution_valid`, que llama a la función `calculate_cycle_number_map` para generar una lista de la misma longitud que la solución, pero con el índice del ciclo al que pertenece la ciudad, y después obtener el índice máximo y compararlo con el número de vendedores viajeros.

Un ejemplo de esta representación para un problema de 10 ciudades y 3 vendedores es `[1 9 5 2 7 4 3 6 8 0]`, y el resultado de `calculate_cycle_number_map` con esa solución como argumento es `[0 0 1 1 1 1 1 1 2 0]`, lo que significa que la primera, la segunda y la última ciudad son visitadas por el primer vendedor, el segundo vendedor visita de la ciudad 3 a la 8 y el tercer vendedor visita la novena.

### 2.2 Función de evaluación

La función de evaluación es muy simple: recibe a una solución válida y regresa la distancia necesaria para cubrir los caminos que esta solución representa.

Esta función está compuesta por sólo dos líneas:

(1) `rows = np.arange(solution.shape[0])`
Esta línea genera una lista de números del 0 a $n-1$, siendo $n$ la longitud de la solución

(2) `return np.sum(self.distances[rows, solution])`
Esta línea aprovecha las propiedades de indexación que *numpy* nos provee, ya que la matriz `self.distances`, que contiene las distancias entre dos ciudades, es indexada en el eje $x$ por esa lista de números del 0 a $n-1$, y además indexada en el eje $y$ por la solución, que igualmente es una lista de valores entre 0 y $n$. Finalmente, esta lista de distancias es sumada usando la función `np.sum`, y regresada de la función.

### 2.3 Función de mutación

Mi implementación de la función de mutación, `mutate_solution`, recibe como parámetro opcional una solución $s1$ y regresa otra solución $s2$. En caso de que el parámetro de solución $s1$ no sea enviado, se genera una solución "tonta" creando una secuencia de números entre 0 y $n-1$ (índices de ciudades), en donde $n$ es el número de ciudades y se desordena aleatoriamente.

La función de mutación sólo realiza operaciones de intercambios de números, eligiendo dos índices aleatorios con la función `generate_random_swap` y checando si la solución mutada es válida con la función `is_solution_valid` y que checando que la solución mutada no sea la misma que la solución original (si fue pasada como parámetro).

## 3 RESULTADOS

Para probar la implementación utilicé los siguientes parámetros:

- $n\_cities = 30$, generar las coordenadas de 30 ciudades en un mapa de 0 a 1
- $n\_salesmen = 6$, el número de vendedores viajeros (o ciclos en mi representación)
- $\alpha = 0.8$, el factor por el que se multiplicará la temperatura después de cada cadena de Markov
- $\beta = 1.5$, tiene el mismo propósito que $\alpha$, pero es usada durante el ajuste de la temperatura inicial
- $n\_batches = 200$, el número máximo de cadenas de Markov a ejecutar
- $n\_iterations = 500$, representa el número de iteraciones o soluciones a probar por cada cadena de Markov
- $min\_accepted = 0.8$, la fracción de soluciones aceptadas necesaria para que el proceso de ajuste de la temperatura inicial termine

- *max_batches_with_same_solution* = 20, el número de cadenas de Markov que regresen la misma solución necesario para detener el algoritmo ya que no se detecta un avance

El tiempo total de ejecución fue de 26.06 segundos, ejecutando 96 cadenas de Markov (las últimas 20 con la misma solución, evaluada en 3.53), cada cadena de 500 posibles soluciones.

La solución final fue graficada en el siguiente mapa:

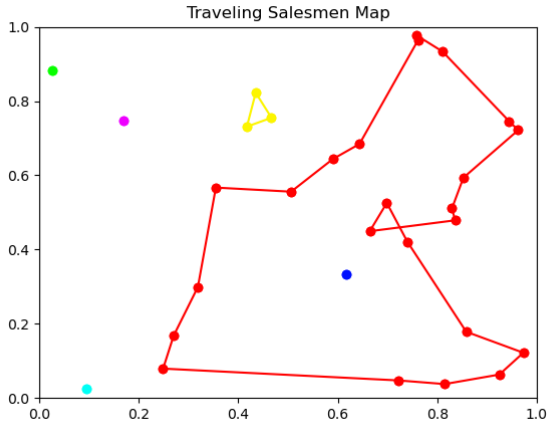


**Figure 1: Mapa de los vendedores viajeros, en donde cada color representa un viajero diferente**

## 3.1 Curva de mejor encontrado

La evolución de la evaluación de la mejor solución forma una curva graficada a continuación:

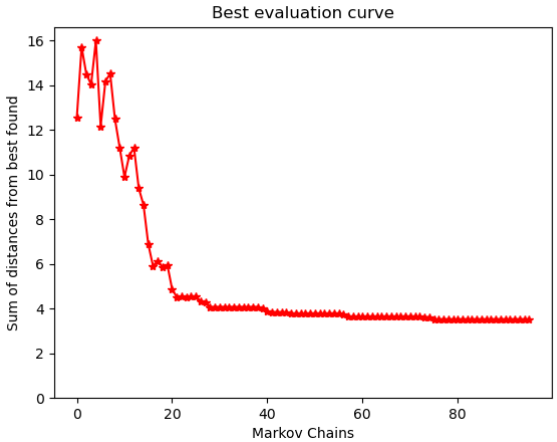


**Figure 2: Curva de mejor encontrado para el mapa anterior**

En el inicio se puede observar que en ocasiones la cadena de Markov acepta soluciones peores (visualizadas como saltos hacia arriba, ya que se elige una solución con una evaluación más alta que la anterior) como consecuencia de que la temperatura es inicializada con el valor de 2.56, pero conforme el algoritmo avanza y la temperatura disminuye se eliminan los saltos hacia arriba.

# 4 CONCLUSIÓN Y RETOS ENCONTRADOS

Considero que la implementación del problema de múltiples vendedores viajeros utilizando el algoritmo de recocido simulado tuvo resultados decentes, a pesar de que se pueden observar algunos cambios "obvios" a simple vista. Definitivamente fue un ejercicio enriquecedor para entender experimentalmente cómo funciona este algoritmo, además de poder mejorar mis habilidades en Python.

Entre los retos que enfrenté, los que considero más interesantes fueron:

- Aprendí sobre el perfilado de programas en Python, lo que fue útil para mejorar la función `calculate_cycle_number_map`, que originalmente estaba implementada con el algoritmo DFS (lineal sobre el tamaño de la solución), y que después descubrí era muy tardada ya que era ejecutada directamente por el intérprete de Python, por lo que la cambié por una implementación que usa `np.min` y que de hecho tiene una complejidad de tiempo cuadrática, pero como está mucho mejor optimizada y a parte la entrada es pequeña, por lo que tiene un desempeño mejor.
- Mejoré mi entendimiento de la librería *matplotlib*, que ya había utilizado para crear gráficas anteriormente, pero esta tarea me permitió experimentar con los mapas de colores, las gráficas múltiples usando la misma figura y el manejo de los datos que esperan las funciones como `plt.plot` y `plt.scatter`.

# REFERENCES

[1] Tolga Bektas. 2006. The multiple traveling salesman problem: an overview of formulations and solution procedures. *omega* 34, 3 (2006), 209–219.

[2] Karla L Hoffman, Manfred Padberg, Giovanni Rinaldi, et al. 2013. Traveling salesman problem. *Encyclopedia of operations research and management science* 1 (2013), 1573–1578.

# A IMPLEMENTACIÓN DEL PROBLEMA DE MÚLTIPLES VENDEDORES VIAJEROS EN PYTHON

```python
import math
from typing import Optional, Tuple

import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics.pairwise import pairwise_distances


class MultipleTravelingSalesmen:
    def __init__(self, coords: np.ndarray, n_salesmen: int = 2,
                 alpha: float = 0.8, beta: float = 1.5, n_batches: int = 200,
                 n_iterations: int = 500, min_accepted: float = 0.8,
                 max_batches_with_same_solution: int = 20):
        self.coords = coords
        self.distances = pairwise_distances(coords)
        self.n_salesmen = n_salesmen
        self.alpha = alpha
        self.beta = beta
        self.n_batches = n_batches
        self.n_iterations = n_iterations
        self.min_accepted = min_accepted
        self.max_batches_with_same_solution = max_batches_with_same_solution

        self.init_temperature()

    def init_temperature(self):
        print("**** Initialize temperature ****")
        self.temperature = 0.1
        solution = self.mutate_solution()
        n_accepted = 0
        iter = 1

        while n_accepted / self.n_iterations < self.min_accepted:
            solution, n_accepted = self.run_batch(solution)
            print("Batch {}:   T {:.2f}  \tacc {:.2f}".format(
                iter, self.temperature, n_accepted / self.n_iterations))
            iter += 1
            if iter == 200:
                raise RuntimeError(
                    "Temp took more than 200 iterations to initialize")
            self.temperature *= self.beta

    def evaluate_solution(self, solution: np.ndarray) -> float:
        rows = np.arange(solution.shape[0])
        return np.sum(self.distances[rows, solution])

    def generate_random_swap(self) -> Tuple[int, int]:
        sample_space = np.arange(self.distances.shape[0])
        samples = np.random.choice(sample_space, 2)
        return (samples[0], samples[1])
```

```python
    def mutate_solution(self, original_solution: Optional[np.ndarray] = None) -> np.ndarray:
        if original_solution is None:
            solution = np.arange(self.distances.shape[0])
            np.random.shuffle(solution)
        else:
            solution = original_solution.copy()

        i, j = self.generate_random_swap()
        solution[i], solution[j] = solution[j], solution[i]

        while (original_solution == solution).all() or not self.is_solution_valid(solution):
            i, j = self.generate_random_swap()
            solution[i], solution[j] = solution[j], solution[i]

        return solution

    def calculate_cycle_number_map(self, solution: np.ndarray) -> np.ndarray:
        solution = solution.copy()
        for _ in range(solution.shape[0]):
            np.minimum(solution, solution[solution], solution)
        _, cycle_num_map = np.unique(solution, return_inverse=True)
        return cycle_num_map

    def is_solution_valid(self, solution: np.ndarray) -> bool:
        # Check number of cycles == number of salesmen
        n_cycles = self.calculate_cycle_number_map(solution).max() + 1
        return n_cycles == self.n_salesmen

    def run_batch(self, starting_solution: np.ndarray) -> Tuple[np.ndarray, int]:
        """ Cadena de Markov """
        best_solution = starting_solution
        best_evaluation = self.evaluate_solution(best_solution)
        n_accepted = 0

        for _ in range(self.n_iterations):
            new_solution = self.mutate_solution(best_solution)
            new_evaluation = self.evaluate_solution(new_solution)
            # Accepted because it's better
            if new_evaluation <= best_evaluation:
                best_solution = new_solution
                best_evaluation = new_evaluation
                n_accepted += 1
                continue

            random_probability = np.random.random()
            delta_eval = abs(new_evaluation - best_evaluation)
            delta_probability = math.exp(- delta_eval / self.temperature)
            # Accepted because of random probability
            if random_probability < delta_probability:
                best_solution = new_solution
                best_evaluation = new_evaluation
                n_accepted += 1

        return (best_solution, n_accepted)
```

```
    def run(self) -> Tuple[np.ndarray, np.ndarray]:
        print("**** Run the algorithm {} times ****".format(self.n_batches))
        best_solution = self.mutate_solution()
        best_eval_history = np.empty(self.n_batches)
        best_eval_history[0] = self.evaluate_solution(best_solution)
        batches_with_same_solution = 0

        for i in range(1, self.n_batches):
            new_solution, n_accepted = self.run_batch(best_solution)
            best_eval_history[i] = self.evaluate_solution(new_solution)
            print("Batch {:4}: T e^{:.3f}\tacc {:.3f} eval {:.7f} ".format(
                i + 1, math.log(self.temperature),
                n_accepted / self.n_iterations, best_eval_history[i]))
            if (best_solution == new_solution).all():
                batches_with_same_solution += 1
            else:
                best_solution = new_solution
                batches_with_same_solution = 0
            if batches_with_same_solution == self.max_batches_with_same_solution:
                print("Early termination because {} consecutive batches returned the same solution"
                      .format(batches_with_same_solution))
                best_eval_history = best_eval_history[:i+1]
                break
            self.temperature *= self.alpha

        return (best_solution, best_eval_history)

    def graph_solution(self, solution: np.ndarray):
        solution_idxs = np.arange(len(solution))
        x1s = self.coords[solution_idxs, 0]
        y1s = self.coords[solution_idxs, 1]
        x2s = self.coords[solution, 0]
        y2s = self.coords[solution, 1]

        cycle_num_map = self.calculate_cycle_number_map(solution)
        n_cycles = cycle_num_map.max() + 1
        colormap = plt.get_cmap("hsv")
        colors = [colormap(i / n_cycles) for i in cycle_num_map]

        plt.title('Traveling Salesmen Map')
        for x1, y1, x2, y2, color in zip(x1s, y1s, x2s, y2s, colors):
            plt.plot([x1, x2], [y1, y2], 'o-', c=color, mfc=color)
        plt.xlim(0, 1)
        plt.ylim(0, 1)

    def graph_evaluation_history(self, evaluation_history: np.ndarray):
        history_size = evaluation_history.shape[0]
        plt.title('Best evaluation curve')
        plt.plot(range(history_size), evaluation_history,
                 color='red', marker='*')
        plt.ylim(bottom=0)
        plt.xlabel('Markov Chains')
        plt.ylabel('Sum of distances from best found')
```

```
162 if __name__ == "__main__":
163     n_cities = 30
164     coords = np.random.rand(n_cities, 2)
165     n_salesmen = 6
166 
167     tsp = MultipleTravelingSalesmen(coords, n_salesmen)
168     best_solution, best_evaluation_history = tsp.run()
169     tsp.graph_solution(best_solution)
170     plt.savefig("map.png")
171     plt.clf()
172     tsp.graph_evaluation_history(best_evaluation_history)
173     plt.savefig("curve.png")
```