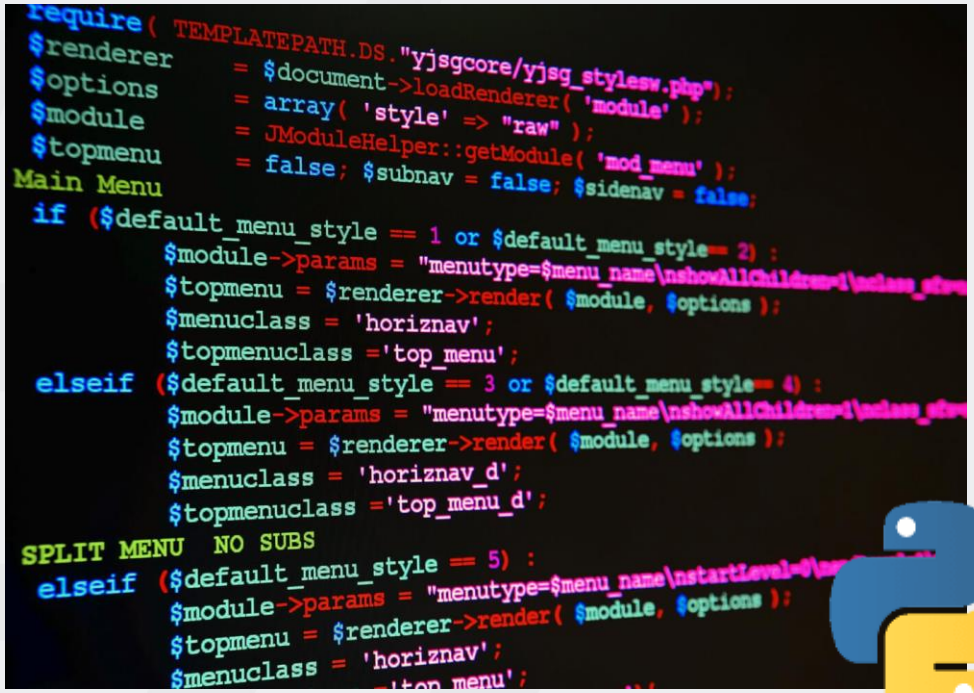


python

Solución al TSP con Python

Ing. Juan Camilo Correa Chica

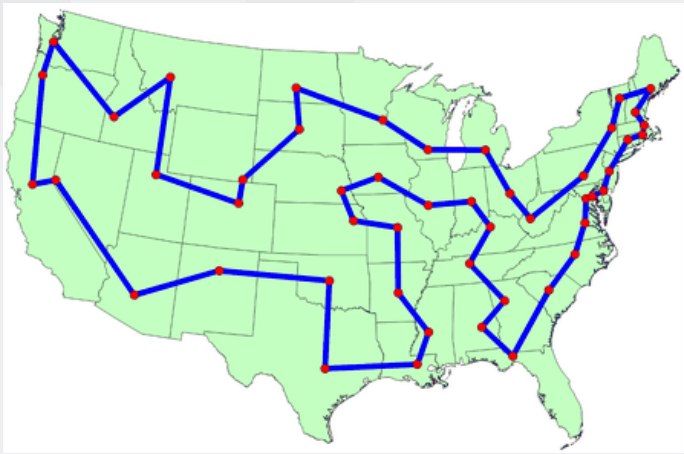
Solución al Travel Salesman Problem con Python

- Travel Salesman Problem - TSP
- Heurística constructiva: Método de inserción
- Heurística de mejoramiento: Movimientos K-opt

Travel Salesman Problem

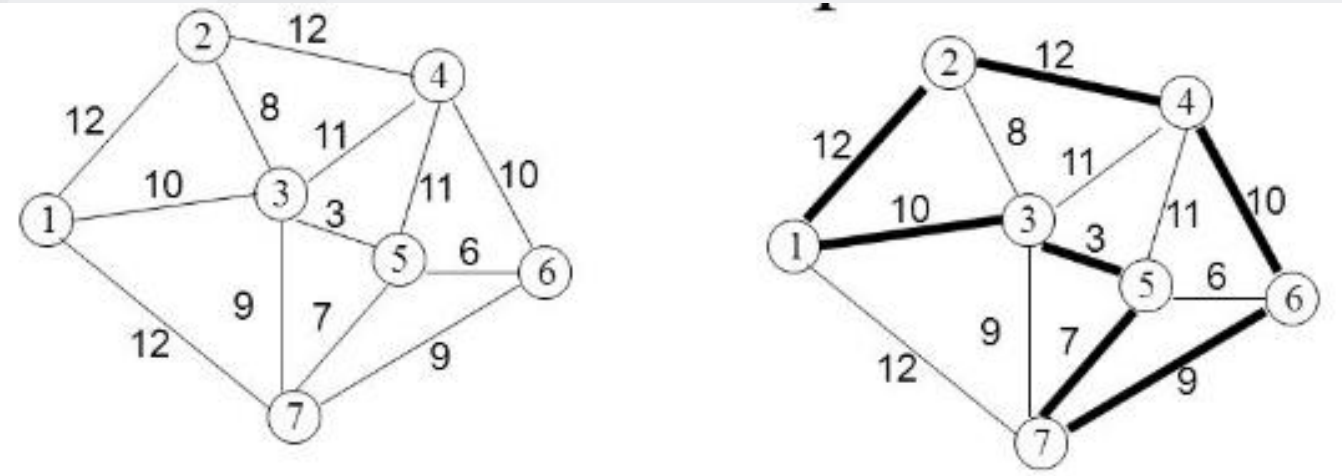
El problema del vendedor viajero (TSP por sus siglas en inglés) es uno de los problemas mejor conocidos en el campo de la investigación de operaciones, ciencias de la computación y matemáticas. La idea básica del problema es la de tratar de encontrar en ciclo más corto en una red tal que todos los nodos de la red sean visitados y que la distancia viajada sea mínima. De manera más técnica se puede enunciar así:

Se tiene una red ***G=[N, A, C]*** que está definida por el número N de nodos, el número A de enlaces entre los nodos y ***C=[c_{ij}]*** la matriz de costos donde c_{ij} es el costo de moverse desde el nodo ***i*** hacia el nodo ***j***. El problema del vendedor viajero se soluciona con un ciclo Hamiltoniano en la red ***G*** con el costo minimo. Un ciclo Hamiltoniano es un ciclo que pasa por cada nodo ***i*** de los ***N*** nodos de la red exactamente una vez.



Travel Salesman Problem

Empezando desde la ciudad 1 (nodo 1), el vendedor debe viajar a todas las otras ciudades (nodos) antes de retornar de nuevo al punto de partida. La distancia entre cada ciudad con respecto a las otras está dada y se asume que es igual independientemente de la dirección. El objetivo es minimizar el costo o distancia total que se debe viajar, un recorrido optimo es aquel en el que no se crean sub-ciclos cerrados y no hay rutas que se entrecrucen.



Travel Salesman Problem

Muchos algoritmos se han propuesto para la solución de este problema. Un proceso de solución típico tiene varias etapas, donde un “tour” inicial se construye y luego se insertan los puntos que quedaron por fuera. Una vez se tienen todos los nodos en el tour se procede a ir mejorando el tour hasta llegar al valor optimo de la solución.

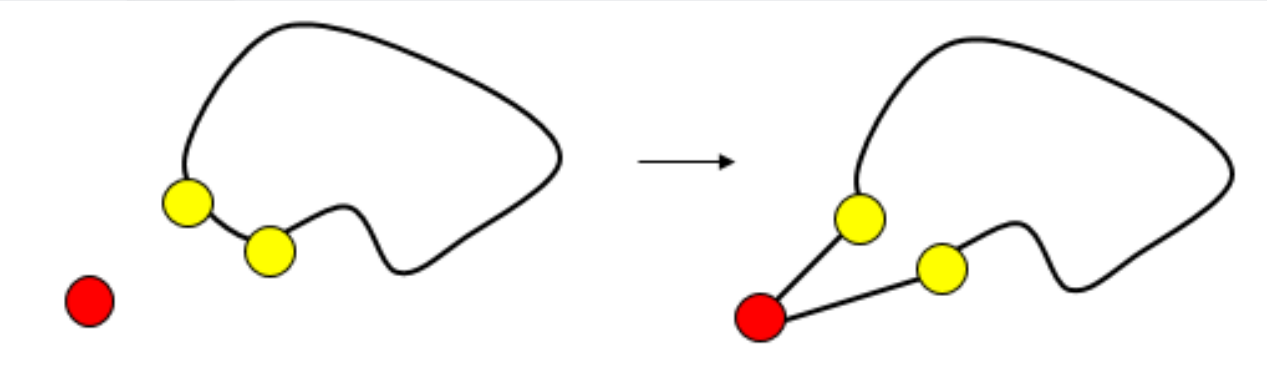
1. Crear un sub-tour inicial: Algoritmos de inserción.
2. Insertar los nodos restantes: Algoritmos de inserción.
3. Mejorar el tour existente: Movimientos 2-opt y 3-opt, y swapping.

Algoritmos de inserción: Inserción del más cercano, inserción del más lejano, inserción del más barato, inserción del más costoso.

Inserción del más cercano: El tour comienza con cualquier nodo de la red, luego se busca el nodo más cercano y se conecta al último nodo que se haya agregado al tour, este paso se repite hasta que todos los nodos se hayan insertado al tour y se conecta el último nodo insertado al primero con el que se comenzó el tour.

Travel Salesman Problem

Inserción del más lejano: El tour comienza con cualquier nodo de la red, luego se busca el nodo más lejano y se conecta al último nodo que se haya agregado al tour, este paso se repite hasta que todos los nodos se hayan insertado al tour y se conecta el último nodo insertado al primero con el que se comenzó el tour.



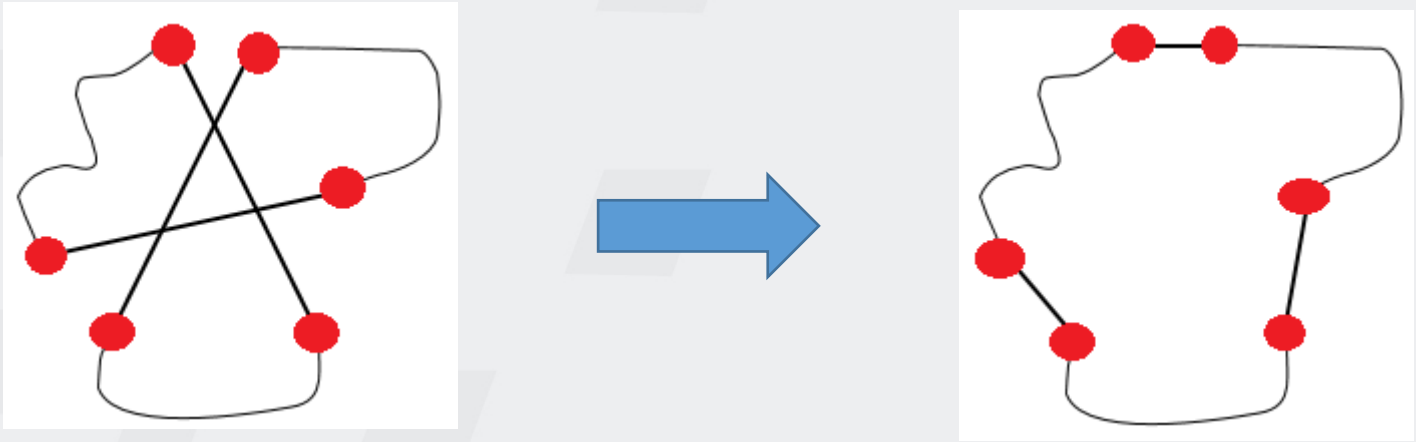
Mejoramiento de la solución

Para mejorar la solución, se pueden emplear movimientos K-opt, con los que se busca minimizar el costo final del tour mediante el rediseño y el intercambio de arcos (vértices) que hacen parte de la solución actual. Los movimientos K-opt ayudan a eliminar arcos que se cruzan y a mejorar la solución ya que encuentra nuevos arcos con costos inferiores a los de la solución original.

Movimiento 2-opt: Se toman dos arcos (vértices), suponga (v, w) y (x, y) y se modifica el tour formando nuevos arcos, sea (v, x) y (w, y) o bien (v, y) y (w, x).

Mejoramiento de la solución

Movimiento 3-opt: En este caso se escogen 3 arcos (vértices) del tour y los nodos pertenecientes a ellos se reconectan para formar nuevos arcos que mejoren el costo de la solución original



Problema

En GitHub se encuentra un archivo de Python (TSP.py) con 3 instancias del problema del vendedor viajero, utilice una heurística constructiva (método de inserción) y una heurística de mejoramiento (movimiento K-opt) para encontrar una solución lo más óptima posible (tour con la menor distancia).

scipy.optimize.minimize

La función scipy.optimize.minimize de la librería scipy solo trabaja en problemas con la siguiente definición:

$$\begin{aligned} &\min F(x) \\ &\text{subject to} \quad C_j(X) = 0, \quad j = 1, \ldots, \text{MEQ} \\ &\qquad\qquad C_j(x) \geq 0, \quad j = \text{MEQ}+1, \ldots, M \\ &\qquad\qquad XL \leq x \leq XU, \; I = 1, \ldots, N. \end{aligned}$$

Es decir, desigualdades del tipo $C_j(x) \geq a$

De modo que las funciones de restricción dela forma:

$$x_1 + x_2 + x_3 \geq 1$$

Se pueden expresar de la siguiente forma:

$$x_1 + x_2 + x_3 - 1 \geq 0$$

scipy.optimize.minimize

Ejemplo: Minimizar el volumen de una caja cúbica, con la restricción de que la superficie de la cara lateral a lo sumo puede ser 10cm^2

```
def calcVolume(x):   #Definicion del volumen de la caja
  length = x[0]
  width = x[1]
  height = x[2]
  volume = length * width * height
  return volume
```

Superficie lateral de la caja:

```
def calcSurface(x):
  length = x[0]
  width = x[1]
  height = x[2]
  surfacearea = 2*(length * width) + 2*(length * height)+
                2*(width * height)
  return surfacearea
```

scipy.optimize.minimize

Definición de la función objetivo:

```
def objective(x):
  return -calcVolume(x) # Maximize instead of minimize
```

Restricción:

```
def constraint(x):
  return 10 - calcSurface(x)

constrts = [{'type':'ineq', 'fun':constraint}]
```

Semilla inicial:

```
long_ini = [1,1,1]
```

Optimización:

```
solBox = optimize.minimize(objective, long_ini,
                           method='SLSQP', constraints=constrts)
```

scipy.optimize.linprog

La función scipy.optimize.linprog permite minimizar una función objetivo sujeta a restricciones expresadas en forma de ecuaciones e inecuaciones (desigualdades) lineales. Adicional a las restricciones que podíamos trabajar con la función minimize, linprog permite trabajar con desigualdades de la forma $C_j(x) \leq a$

De manera generalizada, linprog permite optimizar programas lineales expresados de la siguiente forma:

$$\min f(x)$$
$$s.t\ A\_ub(x) \leq b_{ub}$$
$$A\_eq(x) = b\_eq$$

Ejemplo:

$$\min f = -x_0 + 4x_1$$
$$s.t.\ -3x_0 + x_1 \leq 6$$
$$x_0 + 2x_1 \leq 4$$
$$x_1 \geq -3$$

scipy.optimize.linprog

Para ejecutar el proceso de optimización se requiere de un conjunto de parámetros de entrada para la función linprog, no todos los parámetros son necesarios en todos los tipos de problemas. Los parámetros utilizados con mayor frecuencia son los que permiten introducir los coeficientes de la función objetivo, los coeficientes de las restricciones y los rangos de las variables.

- c : array_like. Coefficients of the linear objective function to be minimized.
- A_ub : 2-D array which, when matrix-multiplied by x, gives the values of the upper-bound inequality constraints at x.
- b_ub : array_like. 1-D array of values representing the upper-bound of each inequality constraint (row) in A_ub.
- A_eq : array_like. 2-D array which, when matrix-multiplied by x, gives the values of the equality constraints at x.
- b_eq : array_like. 1-D array of values representing the RHS of each equality constraint (row) in A_eq.
- bounds : sequence, optional. (min, max) pairs for each element in x, defining the bounds on that parameter. Use None for one of min or max when there is no bound in that direction. By default bounds are (0, None) (non-negative) If a sequence containing a single tuple is provided, then min and max will be applied to all variables in the problem.
- method : str, optional (only). Type of solver. At this time only ‘simplex’ is supported.
- callback : callable, optional. If a callback function is provided, it will be called within each iteration of the simplex algorithm.
- options : dict, optional. A dictionary of solver options.

scipy.optimize.linprog

Luego de ejecutar el proceso de optimización, la función linprog devuelve un objeto de tipo scipy.optimize.OptimizeResult que contiene los siguientes campos:

- x : ndarray. The independent variable vector which optimizes the linear programming problem.
- slack : ndarray. The values of the slack variables. Each slack variable corresponds to an inequality constraint. If the slack is zero, then the corresponding constraint is active.
- success : bool. Returns True if the algorithm succeeded in finding an optimal solution.
- status : int. An integer representing the exit status of the optimization (0: Opimization terminated successfully; 1: Iteration limit reached; 2: Problem appears to be infeasible; 3: Problem appears to be unbounded)
- nit : int. The number of iterations performed.
- message : str. A string descriptor of the exit status of the optimization.

scipy.optimize.linprog

Ejemplo:

$$\min f = -x_0 + 4x_1$$
$$s.t. \; -3x_0 + x_1 \leq 6$$
$$x_0 + 2x_1 \leq 4$$
$$x_1 \geq -3$$

```
from scipy.optimize import linprog as lp

c = [-1, 4]
A = [[-3, 1], [1, 2]]
b = [6, 4]
x0_bounds = (None. None)
x1_bounds = (-3, None)
res = lp(c, A_ub=A, b_ub=b, bounds=(x0_bounds, x1_bounds))
print(res)
```

Ejercicios prácticos utilizando scipy.optimize

1. $\min x_1 x_4(x_1 + x_2 + x_3) + x_3$

$$s.t.\ x_1x_2x_3x_4 \geq 25$$
$$x_1^2+x_2^2 + x_3^2 + x_4^2 = 40$$
$$1 \leq x_1, x_2, x_3, x_4 \leq 5$$

Semilla inicial $x_0 = (1,5,5,1)$

Enlaces útiles Pandas

https://www2.isye.gatech.edu/~mgoetsch/cali/VEHICLE/TSP/TSP003__.HTM
https://docs.mosek.com/6.0/capi/node015.html