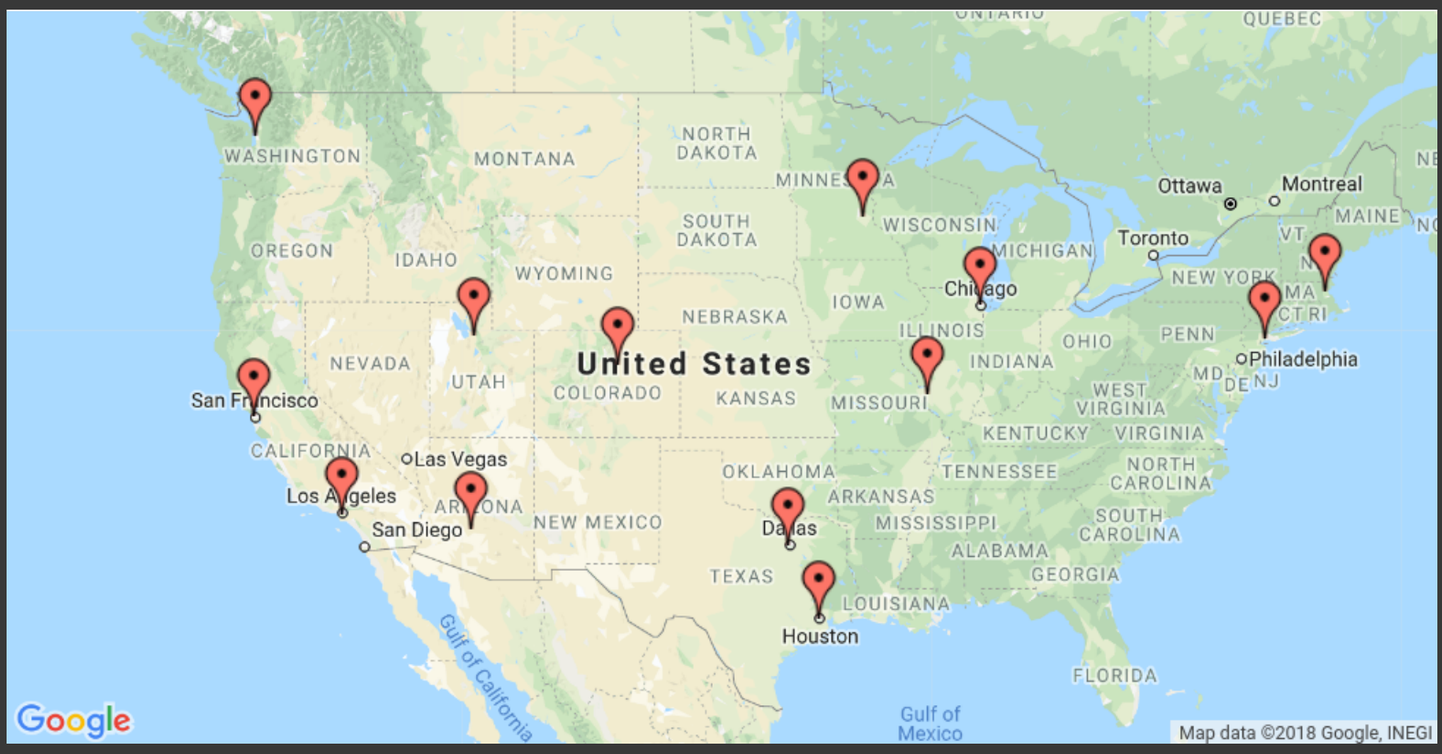
Tarea 9

NLP

Por: Miguel Angel Soto Hernandez

Resolviendo el problema de el agente viajero



Importaciones necesarias

```
!pip install ortools
```

```
Requirement already satisfied: ortools in /usr/local/lib/python3.7/dist-packages (8.2.8710)
Requirement already satisfied: absl-py>=0.11 in /usr/local/lib/python3.7/dist-packages (from orto
Requirement already satisfied: protobuf>=3.14.0 in /usr/local/lib/python3.7/dist-packages (from o
Requirement already satisfied: six in /usr/local/lib/python3.7/dist-packages (from absl-py>=0.11-
```

```
%matplotlib inline
import matplotlib.pyplot as plt
from ortools.constraint_solver import pywrapcp, routing_enums_pb2
```

Creando los datos

```
def crear_datos():
  # Almacena los datos del problema
  data = {}
  data['matriz_distancia'] = [
      [0, 2451, 713, 1018, 1631, 1374, 2408, 213, 2571, 875, 1420, 2145, 1972],
      [2451, 0, 1745, 1524, 831, 1240, 959, 2596, 403, 1589, 1374, 357, 579],
      [713, 1745, 0, 355, 920, 803, 1737, 851, 1858, 262, 940, 1453, 1260],
      [1018, 1524, 355, 0, 700, 862, 1395, 1123, 1584, 466, 1056, 1280, 987],
      [1631, 831, 920, 700, 0, 663, 1021, 1769, 949, 796, 879, 586, 371],
      [1374, 1240, 803, 862, 663, 0, 1681, 1551, 1765, 547, 225, 887, 999],
      [2408, 959, 1737, 1395, 1021, 1681, 0, 2493, 678, 1724, 1891, 1114, 701],
      [213, 2596, 851, 1123, 1769, 1551, 2493, 0, 2699, 1038, 1605, 2300, 2099],
```

```
    [2571, 403, 1858, 1584, 949, 1765, 678, 2699, 0, 1744, 1645, 653, 600],
    [875, 1589, 262, 466, 796, 547, 1724, 1038, 1744, 0, 679, 1272, 1162],
    [1420, 1374, 940, 1056, 879, 225, 1891, 1605, 1645, 679, 0, 1017, 1200],
    [2145, 357, 1453, 1280, 586, 887, 1114, 2300, 653, 1272, 1017, 0, 504],
    [1972, 579, 1260, 987, 371, 999, 701, 2099, 600, 1162, 1200, 504, 0],
]
data['num_vehiculos'] = 1
data['deposito'] = 0
return data
```

La matriz de distancias es una matriz cuya entrada `i`, `j` es la distancia del lugar `i` al lugar `j` en millas, donde los índices de la matriz corresponden a los lugares en el siguiente orden

ID	Ciudad
0	Nueva York
1	Los Ángeles
2	Chicago
3	Minneapolis
4	Denver
5	Dallas
6	Seattle
7	Boston
8	San Francisco
9	Louis
10	Houston
11	Phoenix
12	Salt Lake City

Notas:

- El orden de las ubicaciones en la matriz de distancia es arbitrario, y no está relacionado con el orden de las ubicaciones en cualquier solución del TSP.
- Dado que el solucionador de rutas realiza todos los cálculos con números enteros, la llamada de retorno de la distancia debe devolver una distancia entera para dos ubicaciones cualesquiera. Si alguna de las entradas de data['matriz_distancia'] no es un número entero, deberá redondear las entradas de la matriz o los valores de retorno de la llamada de retorno a números enteros. Véase Escalar la matriz de distancia para un ejemplo que muestra cómo evitar los problemas causados por el error de redondeo.

Los datos también incluyen:

El número de vehículos en el problema, que es 1 porque se trata de un TSP. (Para un problema de rutas de vehículos (VRP), el número de vehículos puede ser mayor que 1). El depósito: la ubicación de inicio y final de la ruta. En este caso, el depósito es 0, que corresponde a Nueva York.

Crear el modelo de enrutamiento

El siguiente código en la sección principal de los programas crea el gestor de índices (manager) y el modelo de enrutamiento (routing). El método `manager.IndexToNode` convierte los índices internos del solucionador (que puedes ignorar con seguridad) en los números de las ubicaciones. Los números de ubicación corresponden a los índices para la matriz de distancia.

Las entradas al `RoutingIndexManager` son:

- El número de filas de la matriz de distancia, que es el número de ubicaciones (incluyendo el depósito).
- El número de vehículos en el problema.
- El nodo correspondiente al depósito.

```
data = crear_datos()
administrador = pywrapcp.RoutingIndexManager(len(data['matriz_distancia']),
                                       data['num_vehiculos'], data['deposito'])
enrutamiento = pywrapcp.RoutingModel(administrador)
```

Crear la llamada de retorno de la distancia

Para utilizar el solucionador de rutas, es necesario crear una llamada de retorno de distancia (o tránsito): una función que toma cualquier par de ubicaciones y devuelve la distancia entre ellas. La forma más sencilla de hacerlo es utilizando la matriz de distancia.

La siguiente función crea la llamada de retorno y la registra con el solucionador como `indice_retorno_transito`

La llamada de retorno acepta dos índices, `from_index` y `to_index`, y devuelve la entrada correspondiente de la matriz de distancia.

```
def llamada_retorno(from_index, to_index):
  # Regresa la distancia entre dos nodos
  # Convertir de la variable de enrutamiento Index a la matriz de distancia NodeIndex.
  desde_nodo = manager.IndexToNode(from_index)
  hasta_nodo = manager.IndexToNode(to_index)
  return data['matriz_distancia'][desde_nodo][hasta_nodo]

indice_retorno_transito = enrutamiento.RegisterTransitCallback(llamada_retorno)
```

Establecer el coste del viaje

arc cost evaluator indica al solucionador cómo calcular el coste del viaje entre dos lugares cualesquiera, es decir, el coste de la arista (o arco) que los une en el gráfico del problema. El siguiente código establece el evaluador de coste de arco.

En este ejemplo, el evaluador del coste del arco es el `indice_retorno_transito`, que es la referencia interna del solucionador a la llamada de distancia. Esto significa que el coste del viaje entre dos lugares cualesquiera es sólo la distancia entre ellos. Sin embargo, en general los costes pueden incluir también otros factores.

También puedes definir múltiples evaluadores de coste de arco que dependan del vehículo que viaja entre las localizaciones, utilizando el método `enrutamiento.SetArcCostEvaluatorOfVehicle()`. Por ejemplo, si los vehículos tienen diferentes velocidades, puedes definir el coste del viaje entre localizaciones como la distancia dividida por la velocidad del vehículo, es decir, el tiempo de viaje

```
enrutamiento.SetArcCostEvaluatorOfAllVehicles(indice_retorno_transito)
```

Establecer los parámetros de búsqueda

El siguiente código establece los parámetros de búsqueda por defecto y un método heurístico para encontrar la primera solución.

El código establece la primera estrategia de solución como `PATH_CHEAPEST_ARC`, que crea una ruta inicial para el solucionador añadiendo repetidamente aristas con el menor peso que no lleven a un nodo visitado previamente (que no sea el depósito).

```
buscar_parametros = pywrapcp.DefaultRoutingSearchParameters()
buscar_parametros.first_solution_strategy = (
    routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC)
```

Añadir la impresora de soluciones

La función que muestra la solución devuelta por el solucionador se muestra a continuación. La función extrae la ruta de la solución y la imprime en la consola.

La función muestra la ruta óptima y su distancia, que viene dada por `ObjectiveValue()`

```
def imprimir_solucion(administrador, enrutamiento, solucion):
  # Imprime la solucion en consola
  print('Objetivo: {} millas'.format(solucion.ObjectiveValue()))
  indice = enrutamiento.Start(0)
  plan_salida = 'Ruta para vehiculo 0:\n'
  distancia_ruta = 0
  solucion_etiquetas = ''
  etiquetas_data = ['New York', 'Los Angeles', 'Chicago', 'Minneapolis', 'Denver',
                  'Dallas', 'Seattle', 'Boston', 'San Francisco', 'St. Louis',
                  'Houston', 'Phoenix', 'Salt Lake City']

  while not enrutamiento.IsEnd(indice):
    plan_salida += f'{administrador.IndexToNode(indice)} -> '
    solucion_etiquetas += f'{etiquetas_data[administrador.IndexToNode(indice)]} -> '
    indice_anterior = indice
    indice = solucion.Value(enrutamiento.NextVar(indice))
    distancia_ruta += enrutamiento.GetArcCostForVehicle(indice_anterior, indice, 0)

  plan_salida += f'{administrador.IndexToNode(indice)}'
  solucion_etiquetas += f'{etiquetas_data[administrador.IndexToNode(indice)]}'
  print(plan_salida)
  print(solucion_etiquetas)
  # plan_salida += 'Distancia de ruta: {} millas\n'.format(distancia_ruta)
  # print(plan_salida)
```

Resolver e imprimir la solución

Por último, puedes llamar al solucionador e imprimir la solución

Esto regresara la solucion y mostrará la ruta optima

```
'''
En este ejemplo, sólo hay una ruta porque es un agente viajero. Pero en problemas
más generales de enrutamiento de vehículos, la solución contiene múltiples rutas.
'''
solucion = enrutamiento.SolveWithParameters(buscar_parametros)
if solucion:
    imprimir_solucion(administrador, enrutamiento, solucion)
```

```
Objetivo: 7569 millas
Ruta para vehiculo 0:
0 -> 7 -> 2 -> 3 -> 6 -> 8 -> 1 -> 11 -> 12 -> 4 -> 5 -> 10 -> 9 -> 0
New York -> Boston -> Chicago -> Minneapolis -> Seattle -> San Francisco -> Los Angeles -> Phoeni
```