Nombre De La Práctica	**ALGORITMO DE CLARK Y WRIG**			**No.**	**1**
Asignatura:	**Programación lógica y funcional**	**Carrera:**	**Ingeniería En Sistemas Computacionales**	**Duración De La Práctica (Hrs)**	**1 hr**
Nombre	**Alexander Martínez Cisneros**	**Grupo**	**3701**		

I. Competencia(s) específica(s):

II. Lugar de realización de la práctica (laboratorio, taller, aula u otro):

- Aula
- Casa

III. Material empleado:

- Laptop
- Lenguaje Python
- Editor de Código Como Visual Studio Code

IV. Desarrollo de la práctica:

ALGORITMO DE CLARK Y WRIG

En temas anteriores se introdujo el problema VRP y se analizó con algunos tipos de restricciones que podían condicionar el problema. Uno de los algoritmos más conocidos para resolver el problema VRP es el algoritmo de **Clark y Wrig** también conocido como **Algoritmos de los ahorros** (Saving Algorithm). Este algoritmo usa heurística, la cual se analizará a continuación, para ir construyendo sistemáticamente las rutas de cada vehículo. Evidentemente, hay que adaptar el algoritmo a las restricciones impuestas por el problema que se quiere resolver: ¿Se tiene un número limitado de vehículos?, ¿Hay un límite de kilómetros para los vehículos?, ¿Hay que cumplir horarios de reparto?, las restricciones son innumerables.

Para atacar el problema se puede partir de los siguientes supuestos

- Se dispone de un único almacén, donde parten y regresan todos los vehículos tras realizar su reparto.
- Por simplicidad, se va a suponer que no se tiene limitación en el número de vehículos disponibles
- Tampoco hay límite de kilómetros para los vehículos
- Todos los vehículos tienen un límite de carga igual que no puede sobrepasarse
- Cada cliente hace un pedido con peso concreto y puede ser diferente al resto

Para mantener la generalidad, no se introducirá más restricciones espaciales y temporales. En cualquier caso, Introducir otro tipo de algoritmo. Hay que hacer notar que este algoritmo puede adaptarse perfectamente bien a la optimización de otros problemas similares, como el reparto de prensa, la recolección de basura o planificación de transporte público a la hora de crear líneas de autobús.

La búsqueda utiliza el algoritmo de los ahorros, que se basa en el hecho de que, dado los clientes servidos por dos vehículos diferentes, si se forza a que un vehículo sirva a los dos, se obtendrá un ahorro tanto en diferencia como en vehículos.

Si se considera N clientes que son servidos por N vehículos, es decir cada cliente es servido por un solo vehículo. En este caso la distancia total recorrida por todos los vehículos, es igual a la suma de

las distancias desde el almacén hasta cada cliente y multiplicada por dos, ya que el vehículo debe ir y volver al almacén suponiendo que el camino de ida tenga la misma distancia que el de vuelta. matemáticamente se puede expresar como:

$$Dist = 2 * \sum_{i=1}^{n} d(almacén, cliente\ i)$$

Donde la función **d(almacén, cliente i)** representa la distancia entre el almacén y el **i-esimo** cliente. Aunque es una solución valida, evidentemente esta solución es poco práctica. No es confiable usar un vehículo para cada cliente. Si se hace que uno de los vehículos sirva a dos clientes (se le llamara **cliente i** y **cliente j)**, se habrá ahorrado de entrada un vehículo, pero también se habrá ahorrado en distancia recorrida.

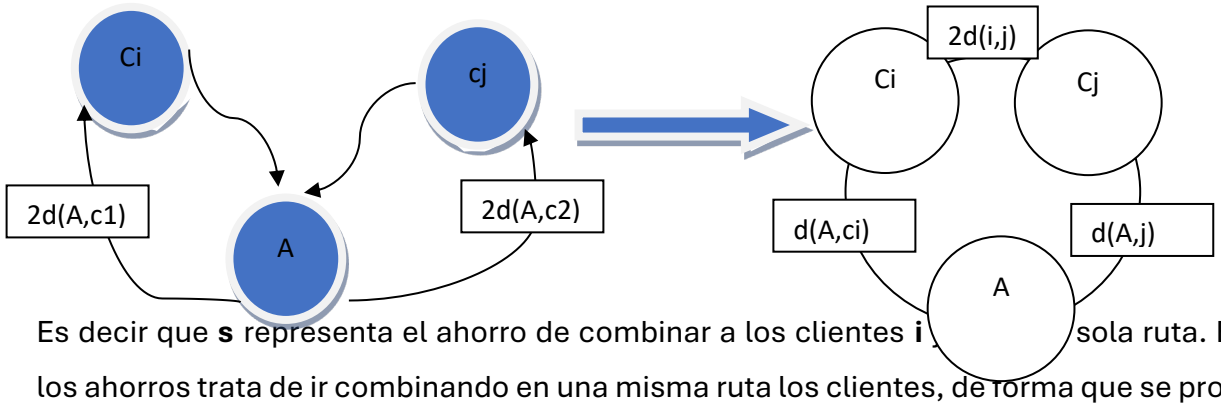
A ese ahorro se le llamará **s** y podrá ser calculado con la siguiente formula (siendo **A** el almacén)

$$S(i,j) = 2d(A,i) + 2d(A,j) - [d(A,i) + d(A,j)]$$

Nota: La covarianza nos dice que tan cerca están los datos(clientes), el punto más cercano entre dos puntos en la recta, si baja la pendiente mejor.

Simplificando queda la siguiente manera:

Ahora se puede ver de forma más intuitiva en el siguiente grafico:



Es decir que **s** representa el ahorro de combinar a los clientes **i** y **j** en una sola ruta. El algoritmo de los ahorros trata de ir combinando en una misma ruta los clientes, de forma que se produzca el mejor ahorro posible. Siempre y cuando no se viole ninguna restricción (por ejemplo, que el vehículo sobrepase su máximo de carga).

Supóngase que se dan las siguientes distancias.

- d(A,i)=10 km
- d(A,j)=15 km
- d(i,j)=12 km

El ahorro obtenido de unir a ambos clientes en una sola ruta es

$$\boldsymbol{s(i,j) = d(A,i) + d(A,j) - d(i,j) = 10 + 15 - 12 = 13}$$

Estos son los pasos que sigue el algoritmo de los ahorros para generar las rutas:

Paso 1 Calcular el ahorro que esta dado **s(i,j) = d(A,i) + d(A,j) – d(i,j)** para cada par de clientes

Paso 2 Ordenar los ahorros de mayor a menor en la lista que se llamara, **lista de ahorros**

Paso 3 Recordar la lista de ahorros. Por cada ahorro en la lista, si no se viola ninguna restricción.

- Si ni **i** ni **j** pertenecen a una ruta, crear una ruta con **i** y con **j**.
- Si **i** o **j** están en ruta (pero no los dos) y además es el primero o el ultimo cliente de la ruta son exteriores (son exteriores), se añade el cliente a la ruta de forma que **i** y **j**, queden juntos.
- Si quedan vehículos sin asignar a una ruta, se crea una o varias (según las restricciones) que los incluya.

En el siguiente programa implementa el algoritmo de los ahorros, sobre la red de carreteras del ejemplo que se ha venido usando en los temas anteriores. En un diccionario llamado coord. Se introduce las coordenadas de las ciudades. Las distancias entre las ciudades se calculan con la función **distancia ( )**. Por simplificar se usará las distancias planas en línea recta que hay entre dos puntos del plano. Se tendrá que desempolvar a Pitágoras y se usará la formula:

$$d = \sqrt{(x1 + x2)^2 + (y1 - y2)^2}$$

Cada ciudad tiene una demanda n que se almacenara en el diccionario llamado pedidos, se supone que la carga máxima de un vehículo es de 40 cajas y se sitúa el almacén en las coordenadas (19.984146, -99.519127), en Jiloyork (Pueblo Mágico).

```python
# VRP
import math
from operator import itemgetter

def distancia(coord1, coord2):
    lat1 = coord1[0]
    lon1 = coord1[1]
    lat2 = coord2[0]
    lon2 = coord2[1]
    return math.sqrt((lat1 - lat2) ** 2 + (lon1 - lon2) ** 2)

def en_ruta(rutas, c):
    ruta = None
    for r in rutas:
        if c in r:
            ruta = r
    return ruta

def peso_ruta(ruta):
    total = 0
    for c in ruta:
        total = total + pedidos[c]
    return total

def vrp_voraz():
    # calcular los ahorros
    s = {}
    for c1 in coord:
        for c2 in coord:
            if c1 != c2:
                if not (c2, c1) in s:
                    d_c1_c2 = distancia(coord[c1], coord[c2])
                    d_c1_almacen = distancia(coord[c1], almacen)
                    d_c2_almacen = distancia(coord[c2], almacen)
                    s[c1, c2] = d_c1_almacen + d_c2_almacen - d_c1_c2
    # ordenar ahorros
    s = sorted(s.items(), key=itemgetter(1), reverse=True)
    # construir rutas
    rutas = []
    for k, v in s:
        rc1 = en_ruta(rutas, k[0])
        rc2 = en_ruta(rutas, k[1])
        if rc1 == None and rc2 == None:
            # no están en ninguna ruta. La creamos.
```

```
            if peso_ruta([k[0], k[1]]) <= max_carga:
                rutas.append([k[0], k[1]])
        elif rc1 != None and rc2 == None:
            # ciudad 1 ya en una ruta. Agregamos la ciudad 2
            if rc1[0] == k[0]:
                if peso_ruta(rc1) + peso_ruta([k[1]]) <= max_carga:
                    rutas[rutas.index(rc1)].insert(0, k[1])
            elif rc1[len(rc1) - 1] == k[0]:
                if peso_ruta(rc1) + peso_ruta([k[1]]) <= max_carga:
                    rutas[rutas.index(rc1)].append(k[1])
        elif rc1 == None and rc2 != None:
            # ciudad 2 ya en una ruta. Agregamos la ciudad 1
            if rc2[0] == k[1]:
                if peso_ruta(rc2) + peso_ruta([k[0]]) <= max_carga:
                    rutas[rutas.index(rc2)].insert(0, k[0])
            elif rc2[len(rc2) - 1] == k[1]:
                if peso_ruta(rc2) + peso_ruta([k[0]]) <= max_carga:
                    rutas[rutas.index(rc2)].append(k[0])
        elif rc1 != None and rc2 != None and rc1 != rc2:
            # ciudad 1 y 2 ya en una ruta. Tratamos de unirlas
            if rc1[0] == k[0] and rc2[len(rc2) - 1] == k[1]:
                if peso_ruta(rc1) + peso_ruta(rc2) <= max_carga:
                    rutas[rutas.index(rc2)].extend(rc1)
                    rutas.remove(rc1)
            elif rc1[len(rc1) - 1] == k[0] and rc2[0] == k[1]:
                if peso_ruta(rc1) + peso_ruta(rc2) <= max_carga:
                    rutas[rutas.index(rc1)].extend(rc2)
                    rutas.remove(rc2)
    return rutas

if __name__ == "__main__":
    coord = {
        'Jilotepec': (19.984146, -99.519127),
        'Toluca': (19.283389, -99.651294),
        'Atlacomulco': (19.797032, -99.875878),
        'Guadalajara': (20.666006, -103.343649),
        'Monterrey': (25.687299, -100.315655),
        'Cancun': (21.080865, -86.773482),
        'Morelia': (19.706167, -101.191413),
        'Aguascalientes': (21.861534, -102.321629),
        'Querétaro': (20.614858, -100.392965),
        'CDMX': (19.432361, -99.133111),
    }
    pedidos = {
```

```
        'Jilotepec':10,
        'Toluca':15,
        'Atlacomulco':0,
        'Guadalajara':0,
        'Monterrey':40,
        'Cancun':50,
        'Morelia':25,
        'Aguascalientes':45,
        'CDMX':60,
        'Querétaro':100,
    }

    almacen = (40.23, -3.40)
    max_carga = 40
    rutas = vrp_voraz()
    for ruta in rutas:
        print(ruta)
```

Codificación:

1. Importamos las librerías necesarias, math e itemgetter. La librería math proporciona funciones matemáticas avanzadas, mientras que itemgetter es una función del módulo operator que extrae elementos específicos de una secuencia.

```
# VRP
import math
from operator import itemgetter
```

2. Definimos una función para calcular la distancia euclidiana entre dos coordenadas geográficas (latitud y longitud), devolviendo el ahorro.

```
def distancia(coord1, coord2):
    lat1 = coord1[0]
    lon1 = coord1[1]
    lat2 = coord2[0]
    lon2 = coord2[1]
    return math.sqrt((lat1 - lat2) ** 2 + (lon1 - lon2) ** 2)
```

3. Creamos una función para determinar la ruta en la que se encuentra un cliente. Si el cliente no está en ninguna ruta, se retorna un valor nulo. Utilizamos un bucle para verificar la presencia del cliente en cada ruta.

```
def en_ruta(rutas, c):
    ruta = None
    for r in rutas:
        if c in r:
            ruta = r
    return ruta
```

4. Calculamos el peso total (demanda) de una ruta sumando los pedidos de los clientes en la ruta. Utilizamos un bucle para acceder a la demanda de cada cliente y acumular el total.

```
def peso_ruta(ruta):
    total = 0
    for c in ruta:
        total = total + pedidos[c]
    return total
```

5. Definimos una función principal que consta de cuatro fases:

1. Calcular Ahorros:

- Creamos un diccionario s para almacenar los ahorros entre pares de clientes.
- Utilizamos bucles anidados para calcular las distancias y ahorros entre clientes.

- Almacenamos los ahorros en el diccionario s.

2. Ordenar Ahorros:
   - Convertimos el diccionario s en una lista de tuplas y la ordenamos en orden descendente según los ahorros.
3. Construir Rutas:
   - Inicializamos la lista de rutas.
   - Iteramos sobre las tuplas ordenadas y decidimos unir o crear rutas según la lógica condicional.
   - Almacenamos las rutas resultantes en la lista.
4. Devolver Rutas:
   - La función devuelve la lista final de rutas construidas.

```python
def vrp_voraz():
    # calcular los ahorros
    s = {}
    for c1 in coord:
        for c2 in coord:
            if c1 != c2:
                if not (c2, c1) in s:
                    d_c1_c2 = distancia(coord[c1], coord[c2])
                    d_c1_almacen = distancia(coord[c1], almacen)
                    d_c2_almacen = distancia(coord[c2], almacen)
                    s[c1, c2] = d_c1_almacen + d_c2_almacen - d_c1_c2
    # ordenar ahorros
    s = sorted(s.items(), key=itemgetter(1), reverse=True)
    # construir rutas
    rutas = []
    for k, v in s:
        rc1 = en_ruta(rutas, k[0])
        rc2 = en_ruta(rutas, k[1])
        if rc1 == None and rc2 == None:
            # no están en ninguna ruta. La creamos.
            if peso_ruta([k[0], k[1]]) <= max_carga:
                rutas.append([k[0], k[1]])
        elif rc1 != None and rc2 == None:
            # ciudad 1 ya en una ruta. Agregamos la ciudad 2
            if rc1[0] == k[0]:
                if peso_ruta(rc1) + peso_ruta([k[1]]) <= max_carga:
                    rutas[rutas.index(rc1)].insert(0, k[1])
            elif rc1[len(rc1) - 1] == k[0]:
                if peso_ruta(rc1) + peso_ruta([k[1]]) <= max_carga:
                    rutas[rutas.index(rc1)].append(k[1])
        elif rc1 == None and rc2 != None:
            # ciudad 2 ya en una ruta. Agregamos la ciudad 1
            if rc2[0] == k[1]:
                if peso_ruta(rc2) + peso_ruta([k[0]]) <= max_carga:
                    rutas[rutas.index(rc2)].insert(0, k[0])
            elif rc2[len(rc2) - 1] == k[1]:
                if peso_ruta(rc2) + peso_ruta([k[0]]) <= max_carga:
                    rutas[rutas.index(rc2)].append(k[0])
        elif rc1 != None and rc2 != None and rc1 != rc2:
            # ciudad 1 y 2 ya en una ruta. Tratamos de unirlas
            if rc1[0] == k[0] and rc2[len(rc2) - 1] == k[1]:
                if peso_ruta(rc1) + peso_ruta(rc2) <= max_carga:
                    rutas[rutas.index(rc2)].extend(rc1)
                    rutas.remove(rc1)
            elif rc1[len(rc1) - 1] == k[0] and rc2[0] == k[1]:
                if peso_ruta(rc1) + peso_ruta(rc2) <= max_carga:
                    rutas[rutas.index(rc1)].extend(rc2)
                    rutas.remove(rc2)
    return rutas
```

6. En la sección principal del código, definimos los diccionarios de datos coord y pedidos, que mapean ubicaciones a coordenadas y demandas respectivamente. También se define la ubicación del almacén y su carga máxima.

7. Llamamos a la función voraz (vrp_voraz) para calcular las rutas basadas en el algoritmo voraz. Los resultados se almacenan en la variable rutas. Iteramos sobre cada ruta e imprimimos su contenido.

```
if __name__ == "__main__":
    coord = {
        'Jilotepec': (19.984146, -99.519127),
        'Toluca': (19.283389, -99.651294),
        'Atlacomulco': (19.797032, -99.875878),
        'Guadalajara': (20.666006, -103.343649),
        'Monterrey': (25.687299, -100.315655),
        'Cancun': (21.080865, -86.773482),
        'Morelia': (19.706167, -101.191413),
        'Aguascalientes': (21.861534, -102.321629),
        'Querétaro': (20.614858, -100.392965),
        'CDMX': (19.432361, -99.133111),
    }
    pedidos = {
        'Jilotepec':10,
        'Toluca':15,
        'Atlacomulco':0,
        'Guadalajara':0,
        'Monterrey':40,
        'Cancun':50,
        'Morelia':25,
        'Aguascalientes':45,
        'CDMX':60,
        'Querétaro':100,
    }

    almacen = (40.23, -3.40)
    max_carga = 40
    rutas = vrp_voraz()
    for ruta in rutas:
        print(ruta)
```

8. Finalmente, ejecutamos el código desde el terminal utilizando el comando python nombre_del_script.py, lo cual debería proporcionar la ruta más óptima.

```
C:\Users\juan\Downloads\Programaci-n-Logica-Funcional-PLF-main>python acw.py
['Guadalajara', 'Morelia', 'Atlacomulco', 'Toluca']
```

Aplicación flask

1. Definición de Datos del Problema:
   - Modifica el diccionario coord para incluir las coordenadas geográficas de las ciudades.
   - Ajusta el diccionario pedido para establecer la cantidad de pedidos para cada ciudad.
   - Define las coordenadas del almacén y la capacidad máxima de carga en las variables almacen y max_carga, respectivamente.

```python
if __name__ == "__main__":
    coord = {
        'JiloYork': (19.984146, -99.519127),
        'Toluca': (19.286167856525594, -99.65473296644892),
        'Atlacomulco': (19.796802401380955, -99.87643301629244),
        'Guadalajara': (20.655773344775373, -103.35773871581326),
        'Monterrey': (25.675859554333684, -100.31405053526082),
        'Cancún': (21.158135651777727, -86.85092947858692),
        'Morelia': (19.720961251258654, -101.15929186858635),
        'Aguascalientes': (21.88473831747085, -102.29198705069501),
        'Queretaro': (20.57005870003398, -100.45222862892079),
        'CDMX': (19.429550164848152, -99.13000959477478)
    }
    pedidos = {
        'JiloYork':10,
        'Toluca':15,
        'Atlacomulco': 0,
        'Guadalajara': 0,
        'Monterrey': 40,
        'Cancún': 50,
        'Morelia': 25,
        'Aguascalientes': 45,
        'CDMX':60,
        'Queretaro': 100
    }
    almacen = (19.979264, -99.609680)
    max_carga = 40

    rutas = vrp_voraz()  # Llama a tu función VRP
```

2. Ejecución del Algoritmo VRP:
   - Llama a la función vrp_voraz() para obtener la lista de rutas optimizadas.

```
def vrp_voraz():
    #Calcular los ahorros
    s={}
    for c1 in coord:
        for c2 in coord:
            if c1 != c2:
                if not (c2,c1) in s:
                    d_c1_c2 = distancia(coord[c1], coord[c2])
                    d_c1_almacen = distancia (coord[c1], almacen)
                    d_c2_almacen = distancia (coord[c2], almacen)
                    s[c1,c2] = d_c1_almacen + d_c2_almacen - d_c1_c2
    #Ordenar Ahorros
    s = sorted(s.items(), key = itemgetter(1), reverse = True)
```

3. Definición de Rutas como Endpoints Web:
   - Entiende cómo se han definido las rutas principales / y /mostrar_rutas.
   - La función render_template se utiliza para mostrar la información en una plantilla HTML.

```
@app.route('/')
def index():
    return render_template('mostrar_rutas.html', rutas=rutas)

@app.route('/mostrar_rutas')
def mostrar_rutas():
    return render_template('mostrar_rutas.html', rutas=rutas)

app.run(debug=True)
```

4. Definición de funciones para las rutas

```
app = Flask(__name__)

def distancia(coord1, coord2):
    lat1 = coord1[0]
    lon1 = coord1[1]
    lat2 = coord2[0]
    lon2 = coord2[1]
    return math.sqrt((lat1 - lat2)**2 + (lon1 - lon2)**2)

def en_ruta(rutas, c):
    ruta = None
    for r in rutas:
        if c in r:
            ruta = r
    return ruta

def peso_ruta(ruta):
    total = 0
    for c in ruta:
        total = total + pedidos[c]
    return total
```

URL: http://alexmc.pythonanywhere.com

V. Conclusión

En conclusión, el Algoritmo de los Ahorros destaca como una solución eficiente para el Problema de Rutas de Vehículos (VRP). Este algoritmo, también conocido como el Algoritmo de Clark y Wright, utiliza heurísticas para construir rutas de vehículos de manera sistemática.

Se adapta a diversas restricciones logísticas, como límites de carga y condiciones específicas de clientes, lo que lo hace versátil. Su enfoque se centra en el concepto de ahorro al combinar clientes en una misma ruta, optimizando la distancia recorrida y aprovechando eficientemente los recursos de los vehículos.

La importancia del Algoritmo de los Ahorros radica en su capacidad para encontrar soluciones eficientes y su aplicabilidad a problemas logísticos más amplios. Su flexibilidad y eficacia lo convierten en una valiosa herramienta en la optimización de procesos de distribución y planificación logística, extendiendo su utilidad más allá del VRP.