Nombre De La Práctica	**Hill Climbing Iterativo**			**No.**	**3**
Asignatura:	**Programación lógica y funcional**	**Carrera:**	**Ingeniería En Sistemas Computacionales**	**Duración De La Práctica (Hrs)**	**2 hr**
Nombre	**Alexander Martinez 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
- Language Python
- Editor de Código Como Visual Studio Code
- Libreria de Flask

IV. Desarrollo de la práctica:

Existe una variación del algoritmo de Hill Climbing que trata de combinar los problemas relativos a los óptimos locales, se trata de **Hill Climbing Iterativo**. La idea de ejecutar varias veces el algoritmo, pero partiendo de un estado inicial diferente cada vez. El siguiente código realiza 10 ejecuciones (Variable max_iteraciones) y al final de cada ejecución se compara el resultado con una variable que se llama mejor_ruta, que almacena la mejor ruta encontrada hasta el momento. Si la ruta encontrada en una de las iteraciones es mejor que la almacenada en mejor_ruta, se sustituye por la nueva mejor.

```
# TSP con Hill Climbing Iterativo


import math

import random


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)

# calcula la distancia cubierta por una ruta

def evalua_ruta(ruta):

    total=0

    for i in range(0,len(ruta)-1):

        ciudad1=ruta[i]

        ciudad2=ruta[i+1]
```

```
        total=total+distancia(coord[ciudad1], coord[ciudad2])
    ciudad1=ruta[i+1]
    ciudad2=ruta[0]
    total=total+distancia(coord[ciudad1], coord[ciudad2])
    return total


# la aleatoria es por medio de una semilla o centinela
def i_hill_climbing():
    #Crear ruta inicial aleatoria
    ruta = []
    for ciudad in coord:
        ruta.append(ciudad)
    mejor_ruta = ruta[:]
    max_iteraciones = 10


    while max_iteraciones > 0:
        mejora = True
            #Generar nueva ruta aleatoria
        random.shuffle(ruta)
        while mejora:
            mejora = False
            dist_actual = evalua_ruta(ruta)
```

```
        # Evaluar a los vecinos
    for i in range(0, len(ruta)):
        if mejora:
            break
        for j  in range(0, len(ruta)):
            if i!=j:
                ruta_tmp = ruta[:]
                ciudad_tmp = ruta_tmp[i]
                ruta_tmp[i] = ruta_tmp[j]
                ruta_tmp[j] = ciudad_tmp
                dist = evalua_ruta(ruta_tmp)
                if dist < dist_actual:
                    #Se encontro un vecino que mejora el resultado
                    mejora = True
                    ruta = ruta_tmp[:]
                    break
    max_iteraciones = max_iteraciones-1


if evalua_ruta(ruta)<evalua_ruta(mejor_ruta):
    mejor_ruta=ruta[:]
return mejor_ruta
```

```
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),
    }

    ruta = i_hill_climbing()
    print(ruta)
    print("Distancia total: " + str(evalua_ruta(ruta)))
```

Al ejecutar varias veces esta nueva versión se puede observar que el resultado es mucho más coherente cada vez que se compila (con suerte se obtendrá el mismo). Aun así, nada asegura que se haya alcanzado el máximo global.

Código:

```
# TSP con Hill Climbing Iterativo

import math
import random

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)
# calcula la distancia cubierta por una ruta
def evalua_ruta(ruta):
    total=0
    for i in range(0,len(ruta)-1):
        ciudad1=ruta[i]
        ciudad2=ruta[i+1]
        total=total+distancia(coord[ciudad1], coord[ciudad2])
    ciudad1=ruta[i+1]
    ciudad2=ruta[0]
    total=total+distancia(coord[ciudad1], coord[ciudad2])
    return total

# la aleatoria es por medio de una semilla o centinela
def i_hill_climbing():
    #Crear ruta inicial aleatoria
    ruta = []
    for ciudad in coord:
        ruta.append(ciudad)
    mejor_ruta = ruta[:]
    max_iteraciones = 10

    while max_iteraciones > 0:
        mejora = True
            #Generar nueva ruta aleatoria
        random.shuffle(ruta)
        while mejora:
            mejora = False
            dist_actual = evalua_ruta(ruta)
                # Evaluar a los vecinos
            for i in range(0, len(ruta)):
                if mejora:
```

```
                    break
                for j  in range(0, len(ruta)):
                    if i!=j:
                        ruta_tmp = ruta[:]
                        ciudad_tmp = ruta_tmp[i]
                        ruta_tmp[i] = ruta_tmp[j]
                        ruta_tmp[j] = ciudad_tmp
                        dist = evalua_ruta(ruta_tmp)
                        if dist < dist_actual:
                                #Se encontro un vecino que mejora el resultado
                            mejora = True
                            ruta = ruta_tmp[:]
                            break
            max_iteraciones = max_iteraciones-1

        if evalua_ruta(ruta)<evalua_ruta(mejor_ruta):
            mejor_ruta=ruta[:]
    return mejor_ruta

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),
    }

    ruta = i_hill_climbing()
    print(ruta)
    print("Distancia total: " + str(evalua_ruta(ruta)))
```

Codificación:

1. Importamos las librerías necesarias: math para operaciones matemáticas y random para generar rutas iniciales aleatorias.

```
# TSP con Hill Climbing Iterativo

import math
import random
```

2. Creamos la función distancia que calcula la distancia entre dos coordenadas o ciudades. Las coordenadas se almacenan en las variables lat y lon, y la función retorna la distancia.

```
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. Calculamos la distancia total recorrida. Inicializamos la variable total para acumular la distancia total. Utilizamos un bucle for para iterar sobre cada ruta, sumando las distancias. La función retorna la distancia total recorrida.

```
def evalua_ruta(ruta):
    total=0
    for i in range(0,len(ruta)-1):
        ciudad1=ruta[i]
        ciudad2=ruta[i+1]
        total=total+distancia(coord[ciudad1], coord[ciudad2])
    ciudad1=ruta[i+1]
    ciudad2=ruta[0]
    total=total+distancia(coord[ciudad1], coord[ciudad2])
    return total
```

4. Implementamos la función hill_climbing que utiliza el algoritmo de búsqueda local. Generamos una ruta inicial aleatoria y la almacenamos. Inicializamos una iteración y generamos nuevas rutas aleatorias hasta que no haya mejoras. Finalmente, retorna la mejor ruta encontrada.

```
# la aleatoria es por medio de una semilla o centinela
def i_hill_climbing():
    #Crear ruta inicial aleatoria
    ruta = []
    for ciudad in coord:
        ruta.append(ciudad)
    mejor_ruta = ruta[:]
    max_iteraciones = 10

    while max_iteraciones > 0:
        mejora = True
            #Generar nueva ruta aleatoria
        random.shuffle(ruta)
        while mejora:
            mejora = False
            dist_actual = evalua_ruta(ruta)
                # Evaluar a los vecinos
            for i in range(0, len(ruta)):
                if mejora:
                    break
                for j  in range(0, len(ruta)):
                    if i!=j:
                        ruta_tmp = ruta[:]
                        ciudad_tmp = ruta_tmp[i]
                        ruta_tmp[i] = ruta_tmp[j]
                        ruta_tmp[j] = ciudad_tmp
                        dist = evalua_ruta(ruta_tmp)
                        if dist < dist_actual:
                                #Se encontro un vecino que mejora el resultado
                            mejora = True
                            ruta = ruta_tmp[:]
                            break
            max_iteraciones = max_iteraciones-1

        if evalua_ruta(ruta)<evalua_ruta(mejor_ruta):
            mejor_ruta=ruta[:]
    return mejor_ruta
```

5. Declaramos la sección de ejecución del programa dentro de coord. Creamos un diccionario que mapea el nombre de las ciudades a sus coordenadas geográficas

```
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),
    }
```

6. Llamamos a la función hill_climbing iterativa para encontrar una ruta optimizada y luego imprimimos la ruta

y la distancia total.

```
ruta = i_hill_climbing()
print(ruta)
print("Distancia total: " + str(evalua_ruta(ruta)))
```

7. Ahora ejecutamos el programa:

```
['Jilotepec', 'Cancun', 'Monterrey', 'Aguascalientes', 'Guadalajara', 'Morelia', 'Toluca', 'CDMX', 'Atlacomulco', 'Querétaro']
Distancia total: 40.35508547087949
```

Codificación de la aplicación en flask

1. Importamos los módulos necesarios: flask y render_template, y mantenemos las otras librerías utilizadas anteriormente. Utilizamos una extensión de Flask para inicializar la aplicación.

```
from flask import Flask, render_template
import math
import random

app = Flask(__name__)
```

2. Creamos el diccionario coord con las coordenadas de las ciudades.

```
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),
}
```

3. Definimos la función distancia que calcula la distancia euclidiana entre dos puntos representados por coordenadas.

```
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)
```

4. Definimos nuevamente una función **evalua_ruta** que evalúa la distancia total recorrida al seguir la ruta proporcionada.

```
def evalua_ruta(ruta):
    total = 0
    for i in range(0, len(ruta) - 1):
        ciudad1 = ruta[i]
        ciudad2 = ruta[i + 1]
        total += distancia(coord[ciudad1], coord[ciudad2])
    total += distancia(coord[ruta[-1]], coord[ruta[0]])  # Agregar la distancia de regreso al punto de inicio
    return total
```

5. Implementamos la función hill_climbing que utiliza el algoritmo de Hill Climbing para minimizar la distancia total recorrida, realizando intercambios aleatorios en la ruta actual

```
def hill_climbing():
    # Crear la ruta inicial Aleatoria
    ruta = list(coord.keys())  # Inicializar con todas las ciudades
    random.shuffle(ruta)

    mejora = True
    while mejora:
        mejora = False
        dist_actual = evalua_ruta(ruta)
        # Evaluar vecinos
        for i in range(0, len(ruta)):
            if mejora:
                break
            for j in range(0, len(ruta)):
                if i != j:
                    ruta_tmp = ruta[:]
                    ciudad_tmp = ruta[i]
                    ruta_tmp[i] = ruta_tmp[j]
                    ruta_tmp[j] = ciudad_tmp
                    dist = evalua_ruta(ruta_tmp)
                    if dist < dist_actual:
                        # Se ha encontrado un vecino que mejora el resultado
                        mejora = True
                        ruta = ruta_tmp[:]
                        break
    return ruta
```

6. Creamos una carpeta para almacenar los templates HTML (index y resultado) con elementos para enviar y recibir datos y definimos una ruta para la página principal que renderiza la plantilla HTML del

index.

```
@app.route('/')
def principal():
    return render_template("index.html")

@app.route('/resultado')
def resultado():
    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),
    }
```

7. Definimos una ruta para la página de resultados que renderiza la plantilla HTML del resultado e Inicializamos la aplicación usando el método para correr el programa.

```
    ruta = hill_climbing()
    mejor_ruta = " -> ".join(ruta)  # Convierte la lista en una cadena
    distancia_total = evalua_ruta(ruta)
    return render_template("resultado.html", mejor_ruta=mejor_ruta, resultado=distancia_total)

if __name__ == '__main__':
    app.run()
```

8. Ejecutamos de manera local:

```
 * Serving Flask app 'flask_app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 122-051-617
```

9. Vemos que funciona nos renderiza la página principal, presionemos el botón para que nos muestre los resultados:

URL: http://alexmc18.pythonanywhere.com

V. Conclusión

En última instancia, el Hill Climbing Iterativo se destaca por su enfoque pragmático y flexible en la resolución de problemas. Su capacidad para ajustarse y mejorar iterativamente las soluciones lo convierte en una herramienta valiosa en situaciones donde las condiciones cambian o se desconocen. Al incorporar intercambios aleatorios, el algoritmo amplía su capacidad para explorar diferentes opciones, evitando quedar atrapado en soluciones subóptimas. Su aplicabilidad se encuentra en una variedad de áreas, desde la planificación de rutas de entrega hasta la optimización de estrategias comerciales.

En resumen, Hill Climbing Iterativo se presenta como un aliado confiable en la toma de decisiones, ofreciendo soluciones adaptables y eficientes en entornos dinámicos. Su enfoque iterativo refleja la importancia de aprender y ajustarse continuamente para alcanzar resultados óptimos en escenarios complejos y cambiantes.