Nombre De La Práctica	**Templado simulado**			**No.**	**4**
Asignatura:	**Programación lógica y funcional**	**Carrera:**	**Ingeniería En Sistemas Computacionales**	**Duración De La Práctica (Hrs)**	**2 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
- Language Python
- Editor de Código Como Visual Studio Code
- Libreria de Flask

IV. Desarrollo de la práctica:

Simulated Annealing

Como se ha visto en la sección anterior, el mayor peligro con el que se enfrenta la búsqueda local es caer en un máximo local. Las tres técnicas que se van a presentar en lo que queda del curso, se van a enfocar a intentar escapar de los máximos locales. A esto se le conoce como **metaheurísticos** que son métodos heurísticos genéricos para problemas que no disponen de un algoritmo que los resuelva) parametrizables que permiten atacar un especto de problemas diferentes. Las tres técnicas que se van a analizar son **el templado simulado o Simulated Annealing**, **la búsqueda tabu** y **los algoritmos genéticos evolutivos**.

La técnica del templado simulado, tiene su origen en el procedimiento físico del templado del acero, el cristal o la cerámica. La técnica del templado consiste en elevar mucho la temperatura del material y luego enfriarlo lentamente para alterar sus propiedades físicas y hacerlos más resistentes.

El templado simulado se parece mucho a la técnica de Hill climbing pero añade algunas variaciones. La primera diferencia es cada iteración, no siempre elegirá un vecino que mejora la solución actual, si no que puedes seleccionar una solución peor. La aceptación o no de una solución depende de una función probabilística. De esta forma no siempre se tomará el camino de mayor pendiente como se hacía en Hill Climbing, si no que se añade un elemento de variación que permita escapar de los máximos locales.

Durante el proceso, en lugar de evaluar todos los vecinos, del estado actual, se evalúan solo algunos. Si el nuevo estado mejora el actual, se hará el cambio tal y como se hacía en Hill climbing, pero si no, se usara la función de probabilidad que decidirá si se hace o no el cambio.

¿Qué función de probabilidad se debe usar?

Básicamente la probabilidad de hacer el cambio dependerá de dos factores, el primer factor es la inferencia relativa que hay entre el estado actual y el nuevo. Si la nueva solución es considerable peor a la actual, tendrán menos probabilidad de realizar el cambio. El segundo factor se llamará temperatura y se denota como T. La función de probabilidad que se usara es la siguiente:

$$p = e^{\frac{eval(E_{actual}) - eval(E_{nuevo})}{T}}$$

Siendo E_{actual} el estado actual y el E_{nuevo} el nuevo vecino. La función eval() es la función de evaluación que se usa para medir la calidad de a solución. Por ejemplo, la distancia de la ruta en caso del TSP. Si lo que se busca es el estado mínimo, se cambia el minuendo y el sustraendo del numerador exponente.

El rol realiza la operación ($E_{actual} - E_{nuevo}$) es claro: la probabilidad de cambio depende de la medida en que una solución es peor que la otra (diferencia de costos), pero ¿Qué misión tiene T(Temperatura)? En la siguiente tabla se observa que ocurre al ir incrementando el valor de T.

T	$P=e^{\frac{-10}{T}}$
1	0.00004
5	0.13533
10	0.36787

Aquí vemos que entre más temperatura hay mejor probabilidad de obtener un resultado optimo. Aquí el objetivo saber el resultado que este más cercano a uno, sabiendo eso sabemos que es la temperatura adecuada. Aquí separamos las rutas chidas de las que no son.

Se observa que, a mayor de T, mayor es el valor de la función de probabilidad. Esto quiere decir que a un valor alto de la temperatura favorecerá que se produzca la selección de un estado peor que el actual.

La técnica se llama templado simulado porque consiste en comenzar con un valor alto de temperatura T e ir disminuyendo poco a poco en cada iteración de forma que los primeros momentos, sea fácil que se seleccione las soluciones peores, y con el paso del tiempo, según se enfrié T, ir estabilizándose en la mejor solución encontrada, ya que la probabilidad de seleccionar un vecino peor decrece.

El seudocódigo del algoritmo es el siguiente:

```
Funcion simulated_annealing(Estado_actual):
    Inicializar T
    Inicializar T_Min
    Inicializar v_enfriamiento
Mientras T<T_Min:
```

Para i = 0 hasta v_enfriamiento:

Estado_nuevo = elegir un estado vecino

Si eval(estado_actual)<eval(estado_nuevo):

Estado_actual = estado_nuevo

Si no:

Si random(0,1)<$e^{\frac{eval(Eactual)-eval(Enuevo)}{T}}$

Estado_actual = estado_nuevo

Enfriar(T)

Salir con el estado_Actual.

Se puede notar en las tres variables que se inicialicen al principio de la función. La variable T(temperatura) la cual es conocida, ¿Pero qué valor inicial hay que darle? Un valor muy alto haría que se tardara mucho en converger hacia la solución estable, mientras que un valor bajo no permitiría que la ejecución tuviera tiempo de alcanzar un estado estable aceptable.

La variable T_Min indica el valor inferior que se alcanza en el proceso de enfriamiento y por lo tanto, cuando la temperatura alcance el valor de T_Min se habrá terminado (es la condición de Stop). Es la practica es habitual que el algoritmo termine al llegar a un número máximo de iteraciones en lugar de alcanzar el valor T_min.

Finalmente, la variable v_enfriamiento (velocidad de enfriamiento) indica cuantos vecinos se evalúan en cada iteración estacionario para la temperatura de ese momento.

En general, la elección de los valores de estos tres parámetros dependerá de gran medida del problema en concreto y del tamaño de la vecindad de las soluciones. Ya se ha comentado que la función () enfriar va disminuyendo el valor de la temperatura T en cada iteración, pero con ¿qué velocidad hay que enfriar T?, existen diversos métodos y la elección de uno u otro dependerá, del problema sobre el que se esté trabajando la forma más intuitiva es enfrían T, disminuyéndola linealmente, es decir, ir restando una cantidad fija en cada iteración del bucle. Otras funciones que pueden ser utilizadas para el enfriamiento son:

Descenso Exponencial	$T_{i+1} = \delta x(\delta \in [0.8,0.9])$
Criterio de Boltzmann	$Ti = \frac{T_0}{1 + \log(i)}$
Criterio de Cauchy	$T_i = \frac{T_0}{1 + i}$

Código:

```
# TSP con templado simulado
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)

# Calcular 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

def simulated_annealing(ruta):
    T = 20
    T_MIN = 0
    V_enfriamiento = 100

    while T > T_MIN:
        dist_actual = evalua_ruta(ruta)
        for i in range(1, V_enfriamiento):
            # Intercambios de dos ciudades aleatoriamente
            i = random.randint(0, len(ruta)-1)
            j = random.randint(0, len(ruta)-1)
            ruta_tmp = ruta[:]
            ciudad_tmp = ruta_tmp[i]
            ruta_tmp[i] = ruta_tmp[j]
            ruta_tmp[j] = ciudad_tmp
            dist=evalua_ruta(ruta_tmp)
            delta = dist_actual - dist
            if(dist < dist_actual):
                ruta = ruta_tmp[:]
```

```
                break
            elif random.random() < math.exp(delta/T):
                ruta = ruta_tmp[:]
                break

        # Enfriar a T linealmente
        T = T - 0.005

    return ruta

if __name__ == "__main__" :
    coord = {
        'Jiloyork' :(19.916012, -99.580580),
        'Toluca':(19.289165, -99.655697),
        'Atlacomulco':(19.799520, -99.873844),
        'Guadalajara':(20.677754472859146, -103.34625354877137),
        'Monterrey':(25.69161110159454, -100.321838480256),
        'QuintanaRoo':(21.163111924844458, -86.80231502121464),
        'Michohacan':(19.701400113725654, -101.20829680213464),
        'Aguascalientes':(21.87641043660486, -102.26438663286967),
        'CDMX':(19.432713075976878, -99.13318344772986),
        'QRO':(20.59719437542255, -100.38667040246602)
    }

# Crear una ruta inicial aleatoria
ruta = []
for ciudad in coord:
    ruta.append(ciudad)
random.shuffle(ruta)

ruta = simulated_annealing(ruta)
print(ruta)
print("Distancia Total: " + str(evalua_ruta(ruta)))
```

Codificación:

1. Importamos las librerías necesarias, como math para operaciones matemáticas y random para la generación de números aleatorios. Esto resulta crucial para la creación de una ruta inicial aleatoria, fundamental en muchos problemas de optimización.

```
# TSP con Hill Climbing Iterativo

import math
import random
```

2. Creamos la función distancia, la cual solicita dos coordenadas o ciudades. Estas coordenadas se almacenan en las variables lat y lon, y la función procesa y retorna la distancia entre las ciudades.

```
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 recorrida mediante la inicialización de una variable total, que acumula la distancia total. Utilizamos un bucle for para iterar sobre cada ruta, sumando las distancias y retornando al final la distancia total recorrida.

```
# 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
```

4. Implementamos la función que incorpora el algoritmo de Templado Simulado. En el ciclo principal (while), se ejecuta el proceso mientras la temperatura sea mayor a la mínima. El bucle for controla el enfriamiento, limitando el número de intentos de cambio de ciudad en cada fase. El if determina si se acepta la nueva ruta; si su distancia es menor que la de la ruta actual, se acepta. La función retorna la mejor ruta encontrada después de las iteraciones de templado simulado.

```python
def simulated_annealing(ruta):
    T = 20
    T_MIN = 0
    V_enfriamiento = 100

    while T > T_MIN:
        dist_actual = evalua_ruta(ruta)
        for i in range(1, V_enfriamiento):
            # Intercambios de dos ciudades aleatoriamente
            i = random.randint(0, len(ruta)-1)
            j = random.randint(0, len(ruta)-1)
            ruta_tmp = ruta[:]
            ciudad_tmp = ruta_tmp[i]
            ruta_tmp[i] = ruta_tmp[j]
            ruta_tmp[j] = ciudad_tmp
            dist=evalua_ruta(ruta_tmp)
            delta = dist_actual - dist
            if(dist < dist_actual):
                ruta = ruta_tmp[:]
                break
            elif random.random() < math.exp(delta/T):
                ruta = ruta_tmp[:]
                break

        # Enfriar a T linealmente
        T = T - 0.005

    return ruta
```

5. Declaramos la sección de ejecución del programa en el contexto de coord. Generamos un diccionario que asigna el nombre de las ciudades a sus coordenadas geográficas, y lo enviamos como entrada a la primera función.

```python
coord = {
    'Jiloyork' :(19.916012, -99.580580),
    'Toluca':(19.289165, -99.655697),
    'Atlacomulco':(19.799520, -99.873844),
    'Guadalajara':(20.677754472859146, -103.34625354877137),
    'Monterrey':(25.69161110159454, -100.321838480256),
    'QuintanaRoo':(21.163111924844458, -86.80231502121464),
    'Michohacan':(19.701400113725654, -101.20829680213464),
    'Aguascalientes':(21.87641043660486, -102.26438663286967),
    'CDMX':(19.432713075976878, -99.13318344772986),
    'QRO':(20.59719437542255, -100.38667040246602)
}
```

6. Llamamos a la función de Templado Simulado (simulated_annealing) para encontrar una ruta optimizada utilizando este algoritmo. Posteriormente, imprimimos la ruta resultante y la distancia total de la misma, ofreciendo así una solución optimizada a nuestro problema.

7. Ahora ejecutamos el programa:

```
['Aguascalientes', 'Monterrey', 'QuintanaRoo', 'CDMX', 'Toluca', 'Jiloyork', 'Atlacomulco', 'QRO', 'Michohacan', 'Guadal
ajara']
Distancia Total: 38.608380356244794
```

Codificación de la aplicación en flask

1. Importamos los módulos esenciales, como flask y render_template, que son fundamentales para la creación de la aplicación web y la representación de las plantillas HTML. Además, mantenemos las otras librerías importadas anteriormente para las funcionalidades específicas del programa

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

app = Flask(__name__)
```

2. Creamos un diccionario llamado coord que contiene las coordenadas de las ciudades, proporcionando la información geográfica necesaria para el cálculo de distancias.

```
coord = {
    'Jiloyork': (19.916012, -99.580580),
    'Toluca': (19.289165, -99.655697),
    'Atlacomulco': (19.799520, -99.873844),
    'Guadalajara': (20.677754472859146, -103.34625354877137),
    'Monterrey': (25.69161110159454, -100.321838480256),
    'QuintanaRoo': (21.163111924844458, -86.80231502121464),
    'Michoacan': (19.701400113725654, -101.20829680213464),
    'Aguascalientes': (21.87641043660486, -102.26438663286967),
    'CDMX': (19.432713075976878, -99.13318344772986),
    'QRO': (20.59719437542255, -100.38667040246602)
}
```

3. Implementamos la función distancia, la cual calcula la distancia euclidiana entre dos puntos representados por coordenadas.Definimos la función evalua_ruta, encargada de evaluar la distancia total recorrida al seguir una ruta proporcionada.

```
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 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. Desarrollamos la función simulated_annealing, que implementa el algoritmo del Templado Simulado. Este algoritmo se basa en el concepto de enfriamiento, donde la aceptación de soluciones depende de la temperatura; a medida que disminuye la temperatura, la probabilidad de aceptar soluciones

subóptimas disminuye.

```
def simulated_annealing(ruta):
    T = 20
    T_MIN = 0
    V_enfriamiento = 100

    while T > T_MIN:
        dist_actual = evalua_ruta(ruta)
        for i in range(1, V_enfriamiento):
            # Intercambios de dos ciudades aleatoriamente
            i = random.randint(0, len(ruta) - 1)
            j = random.randint(0, len(ruta) - 1)
            ruta_tmp = ruta[:]
            ciudad_tmp = ruta_tmp[i]
            ruta_tmp[i] = ruta_tmp[j]
            ruta_tmp[j] = ciudad_tmp
            dist = evalua_ruta(ruta_tmp)
            delta = dist_actual - dist
            if dist < dist_actual or random.random() < math.exp(delta / T):
                ruta = ruta_tmp

        # Enfriar a T linealmente
        T = T - 0.005

    return ruta
```

5. Creamos una carpeta para almacenar nuestros archivos HTML, incluyendo los templates de "index" y "resultado". En cada uno, incorporamos líneas de código para enviar y recibir datos. Por ejemplo, en el "index", colocamos un botón que envía los datos contenidos en coord para su procesamiento.

6. Definimos rutas para la página principal y de resultados. La primera llama a la función principal(), que genera una nueva ruta y renderiza una plantilla HTML, y la segunda llama a la función resultado() para la visualización de los resultados. Define una ruta para la página de resultados. Llama a la función resultado () que genera una nueva ruta y renderiza otra plantilla HTML.

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

@app.route('/resultado')
```

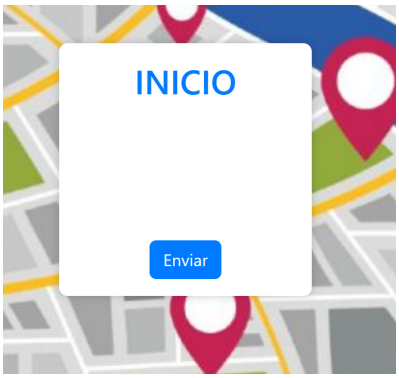
7. Inicializamos la sección donde el programa se inicia directamente. Esto también inicia el servidor, permitiendo la presentación de los resultados y su renderización.

```
    ruta = list(coord.keys())  # Crear una ruta inicial aleatoria
    random.shuffle(ruta)
    mejor_ruta = simulated_annealing(ruta)
    distancia_total = evalua_ruta(mejor_ruta)

    return render_template("resultados.html", mejor_ruta=mejor_ruta, distancia_total=distancia_total)

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

8. Ejecutamos de manera local con el siguiente comando:

```
 * 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: 924-931-096
```

9. Verificamos que la aplicación funcione correctamente, observando la renderización de la página principal. Al presionar el botón, se muestran los resultados en la página correspondiente. Este proceso valida el funcionamiento adecuado de la aplicación web.



Vemos que imprime resultados:

URL: http://tempsim.pythonanywhere.com

V. Conclusión

Simulated Annealing se posiciona como un elemento esencial en la resolución de problemas de optimización al abordar de manera efectiva los desafíos inherentes a la búsqueda local y superar los máximos locales. Este enfoque, considerado como una metaheurística parametrizable, destaca por su capacidad de adaptación a una variedad de problemas y se inspira en el proceso físico de templado de materiales.

A diferencia de Hill Climbing, Simulated Annealing incorpora una exploración probabilística que permite la selección de soluciones potencialmente peores. La función de probabilidad en este método se fundamenta en la diferencia relativa entre el estado actual y el nuevo, así como en la temperatura, que desempeña un papel crucial al controlar la aceptación de soluciones subóptimas.

El proceso de enfriamiento gradual emula la estabilización del sistema al disminuir la temperatura en cada iteración. La elección adecuada de parámetros, como la temperatura inicial, mínima y velocidad de enfriamiento, resulta crucial para la eficacia del algoritmo y varía según las características específicas del problema y la proximidad de las soluciones en el espacio de búsqueda.

La flexibilidad en la elección del método de enfriamiento, que puede ser lineal, exponencial, de Boltzmann o de Cauchy, proporciona la capacidad de adaptar el algoritmo a diversos contextos. En resumen, Simulated Annealing se destaca por ofrecer una exploración amplia, demostrar adaptabilidad ante distintas situaciones y presentar una estrategia efectiva para descubrir soluciones óptimas en problemas de optimización.