UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

Tarea 03 - Metaheurísticas de Trayectoria

ASIGNATURA

Cómputo Evolutivo

PROFESOR

Oscar Hernández Constantino

AYUDANTE

Malinali Gónzalez Lara

ALUMNOS

Carlos Emilio Castañon Maldonado & Dana Berenice Hérnandez Norberto

Considera los siguientes problemas de prueba de optimización continua.

a) **Sphere**

$$f(x) = \sum_{i=1}^{n} x_i^2$$
$$x_i \in [-5.12, +5.12]$$

b) **Ackley**

$$f(x) = 20 + e - 20\ exp\Big(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\Big) - exp\Big(\frac{1}{n}\sum_{i=1}^{n}(\cos\ 2\ \pi\ x_i)\Big)$$
$$x_i \in [-30, +30]$$

c) **Griewank**

$$f(x) = 1 + \sum_{i=1}^{n} \frac{x_i^2}{4000} - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right)$$
$$x_i \in [-600, +600]$$

d) **Rastrigin**

$$f(x) = 10n + \sum_{i=1}^{n} [x_i^2 - 10\cos(2\ \pi\ x_i)]$$
$$x_i \in [-5.12, +5.12]$$

e) **Rosenbrock**

$$f(x) = \sum i = 1^{n-1} \Big[100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2\Big]$$
$$x_i \in [-2.048, +2.048]$$

Ejercicio 1. Recocido Simulado

a) **Describe e implementa un operador de vecindad para soluciones binarias. La función solo debe generar un vecino de manera aleatoria.**
Primero implementamos todas las funciones de prueba

```
#Implementamos todas las funciones a utilizar
def sphere(x):
    x = np.array(x)
    return sum(x**2)

def ackley(x):
    x = np.array(x)
    t1 = -0.2*np.sqrt(np.mean(x**2))
    t2 = np.mean(np.cos(2*np.pi*x))
    y = 20+np.exp(1)-20*np.exp(t1)-np.exp(t2)
    return y

def griewank(x):
    x = np.array(x)
    i = np.array(range(1,len(x)+1))
    t1 = sum(x**2)/4000
    t2 = np.prod(np.cos(x/np.sqrt(i)))
    return 1 + t1 - t2

def rastrigin(x):
    x = np.array(x)
```

```
    t1 = 10*len(x)
    t2 = sum(x**2-10*np.cos(2*np.pi*x))
    return t1 + t2

#La función aplica para parametros mayores o iguales a dos dimensiones
def rosenbrock(x):
    x = np.array(x)
    x1 = x[1:]
    x0 = np.delete(x,-1)
    t1 = 100*(x1-x0**2)**2
    t2 = (1-x0)**2
    return sum(t1 + t2)
```

La representación de soluciones que estamos utilizando es binaria, por lo tanto, los vecinos de una solución debe estar representado de igual manera de forma binaria. Por lo tanto, si cambiamos un bit de forma aleatoria, este será un vecino generado.

```
def vecino(solucion):
    neighbor = np.copy(solucion)
    #Ya que tenemos soluciones binarias y queremos generar un vecino de manera aleatoria
    #basta con generar el cambio de un bit de forma aleatoria
    #Elegimos aleatoriamente el lugar de un bit
    indice = np.random.randint(len(neighbor))
    # Modificamos el valor del bit que cambiaremos
    neighbor[indice] = 1 - neighbor[indice]
    return neighbor
```

De esta manera aplica no solo para una cadena de binarios, si no también para vectores de binarios, esto será muy útil al aplicarlo en funciones que programaremos más adelante.

b) **Describe e implementa un esquema de enfriamiento. Justifica tu elección del esquema de enfriamiento.**

Elegimos el esquema de enfriamiento con descenso lento, debido a que nos pareció interesante explorarlo, además de que nos permite elegir temperaturas no muy altas e ir descendiendo de una forma más lenta, obteniendo una exploración más amplia de soluciones.

```
def esquema_enfriamiento(temp,beta=0.01):
    return temp/(1+beta*temp)
```

c) **Implementa el algoritmo de recocido simulado para optimizar funciones de optimización continua, utilizando un esquema de representación de soluciones con un arreglo de bits [de acuerdo a lo implementado en la Tarea 2]**

Implementamos el algoritmo de recocido simulado, estableciendo el criterio de paro al alcanzar una temperatura muy baja, al aplicar nuestra función establecemos a la temperatura inicial como una temperatura alta de 100, decreciendo hasta llegar una temperatura de 0.01. Primero inicializamos una solución aleatoria en un intervalo dado, evaluamos la función en la solución inicial y codificamos la solución creada para poder generar los vecinos con representación de soluciones binarias, establecemos como constante el número de bits para la representación como 100.
Con base en el criterio de paro, vamos generando vecinos de la solución aleatoria y evaluando si en la función poseen un mejor resultado, si es así, actualizamos directamente nuestra solución. Si no es así, no rechazamos la solución, creamos un criterio aleatorio entre 0 y 1 y calculamos la probabilidad de que la solución sea mejor que la solución inicial, si dicha probabilidad es mayor que el criterio establecido, actualizamos la solución y

el resultado.
Cabe mencionar que el criterio lo establecimos como aleatorio debido a que no todas las funciones son optimizadas con la misma probabilidad, al hacerlo aleatorio le damos la oportunidad a todas de ser optimizadas de alguna forma.

```
def recocido_simulado(temperatura_inicial,temperatura_final
,funcion_objetivo, dimension, intervalo,beta=0.01):
    mejor_solucion = np.random.uniform(intervalo[0], intervalo[1], dimension)
    mejor_evaluacion = funcion_objetivo(mejor_solucion)
    #Codificamos la solucion para poder generar un vecino y realizar la pertubacion
    solucion = codifica_v(mejor_solucion,100,intervalo[0],intervalo[1])
    temperatura = temperatura_inicial

    while temperatura > temperatura_final:
        #Generamos el vecino a evaluar
        nueva_solucion_cod = vecino(solucion)
        #Decodificamos para evaluar la solucion
        nueva_solucion = decodifica_v(nueva_solucion_cod,100,intervalo[0],intervalo[1])
        nueva_evaluacion = funcion_objetivo(nueva_solucion)
        delta = nueva_evaluacion - mejor_evaluacion
        probabilidad = np.exp(-delta/temperatura)
        #Debido a que no todas las funciones necesitan perturbaciones similares
        #establecemos un criterio de aceptacion de probabilidad arbitrario aleatorio
        criterio = random.random()

        if nueva_evaluacion < mejor_evaluacion:
            solucion = codifica_v(nueva_solucion,100,intervalo[0],intervalo[1])
            mejor_evaluacion = nueva_evaluacion
            #Si la probabilidad de que la solucion sea mejor es mayor a nuestro criterio
            #aleatorio, entonces aceptamos la nueva solucion generada
        elif probabilidad > criterio:
            solucion = codifica_v(nueva_solucion,100,intervalo[0],intervalo[1])
            mejor_evaluacion = nueva_evaluacion

        #Aplicamos el esquema de enfriamiento para reducir la temperatura constantemente
        temperatura = esquema_enfriamiento(temperatura,beta)

    mejor_solucion = decodifica_v(solucion,100,intervalo[0],intervalo[1])

    return mejor_solucion, mejor_evaluacion
```

d) **Investiga en qué consiste la codificación de gray. ¿Qué ventajas o desventajas podría tener respecto a la representación implementada en la tarea 2?**

La codificación de Gray es un método de codificación binaria caracterizado principalmente en que dos números consecutivos difieren en un solo bit. La codificación de Gray se aplica a números binarios, aplicando XOR con el mismo número desplazado un bit a la derecha, el primer bit queda igual y aplicamos la compuerta a los demás, sin contar el último bit que es eliminado.
Una ventaja que podría tener con respecto a la solución implementada en la tarea 2 es que podemos tener un mejor orden consecutivo de números en un intervalo dado, así como una representación más exacta. Por otro lado, una desventaja de este método es que utilizamos una mayor cantidad de recursos computacionales para implementar el método.

Ejercicio 2. Experimentación

1 **Propón un criterio de término que permita establecer una comparación justa (en cuanto al uso de recursos) entre los dos algoritmos.**

El algoritmo de recocido simulado utiliza más recursos que el de búsqueda aleatoria, por lo tanto, una buena comparación que podríamos realizar es bajando la temperatura inicial para realizar menos iteraciones hasta nuestro criterio de paro. En este caso como mencionamos anteriormente, establecemos una temperatura de 100 hasta llegar a una temperatura final de 0.01.

2 **Ejecuta cada algoritmo (recocido simulado y búsqueda aleatoria) al menos 10 veces para cada una de las funciones de prueba en dimensión 10. Incluye en el reporte una tabla de resultados, con las estadísticas de los resultados obtenidos.**

Ejemplo de tabla que deben incluir en el reporte:

Funcion	Mejor valor f(x) BA	Valor promedio f(x) BA	Peor valor f(x) BA	Mejor valor f(x) RC	Valor promedio f(x) RC	Peor valor f(x) RC
Sphere	13.12	15.6	19.6			
Ackley	...	...	...		...	...
...	...	...	...		...	...

¿Sería suficiente con ejecutar una vez cada uno de los algoritmos, en cada función, para poder establecer una comparación? ¿Por qué? Justifica tu respuesta

Los resultados que obtuvimos en la comparación de algoritmos fueron variables, dependiendo principalmente de la función que estábamos evaluando. Al principio tomamos como parámetros iniciales un criterio fijo de 0.8, es decir, que solo se tomaban soluciones que tuvieran una probabilidad de que fueran mejores mayor a 0.8, sin embargo, en funciones como la de Griewank no era lo mejor, debido a que tenemos mucho ruido en dicha función, por lo cual son necesarias las pertubaciones para obtener mejores resultados.Por lo tanto, optamos por elegir un criterio de paso aleatorio.
Veamos como en general en la función sphere los resultados no mejoraron debido a que la función no tiene mucho ruido, por lo cual solo retrasa su llegada al óptimo global las perturbaciones. Caso contrario a la función de Rosenbrock que es la función con más ruido, posee mejores resultados con el algoritmo de recocido simulado que con el de búsqueda aleatoria.

Función	Mejor valor	Valor promedio	Peor valor
Sphere 10	4.95	7.88	11.19
Sphere 100	11.87	17.03	27.05
Ackley 10	4.45	4.88	5.20
Ackley 100	4.98	5.97	6.65
Griewank 10	0.42	0.67	0.78
Griewank 100	0.48	0.85	1.07
Rastrigin 10	55.27	76.68	94.01
Rastrigin 100	61.06	122.06	156.09
Rosenbrock BA	589.23	1853.13	2934.28
Rosenbrock RC	426.81	1642.23	2688.68

En general no tenemos resultados tan diferentes, creemos que esto se debe a la representación de soluciones que estamos tomando, no siempre obtenemos una representación tan exacta, por lo cual obtenemos soluciones finitas e inexactas.
No pensamos que sea suficiente con ejecutar una sola vez cada uno de los algoritmos, debido a que están basados en procesos aleatorios, así como estos tienen una tendencia, los algoritmos que creamos también los tienen y nos permiten obtener resultados más acertados.

Ejercicio 3. Búsqueda Local Iterada

Implementa un algoritmo de búsqueda local iterada, para el problema de coloración en gráficas [de acuerdo a lo implementado en la Tarea 2]

- Describe los componentes implementados
- Ejecuta el algoritmo y compara con los resultados de la tarea anterior
- Realiza pruebas con al menos un ejemplar suficientemente grande, en donde no se haya logrado encontrar el óptimo global en la tarea anterior.

Las nuevas funciones utilizadas fueron:

```
def colorearGraficaConNColores2(archivo):
    n_vertices, n_aristas, vertices, aristas = leer_ArchivoCol(archivo)
    # dibujar_Grafica(vertices, aristas)
    n_colores = n_vertices
    solucion = SColoracion(n_vertices)
    vertices = list(range(1, n_vertices+1))

    vertices.sort(key=lambda v: len([v1 for v1,
             v2 in aristas if v1 == v or v2 == v]), reverse=True)

    #vecinos = [v for v1, v2 in aristas
               if v1 == vertice for v in [v2]] + [v for v1, v2 in aristas
                    if v2 == vertice for v in [v1]]

    for vertice in vertices:
        color = (random.randint(1, n_colores))
        while color in coloresVecinos(solucion, vertice):
            color = (random.randint(1, n_colores))
        solucion.asignar_color(vertice, color)

        #for vecino in vecinos:
        #    color = (random.randint(1, n_colores))
        #    solucion.asignar_color(vecino, color)
    # Regresamos una lista de con los colores de la solucion
    return solucion.colores_asignados
```

Esta función genera una solución inicial aleatoria para el problema de colorear una gráfica con un número fijo de colores.

```
# Función para generar una solución vecina
def generar_vecino_iterada(solucion_actual, n_colores):
    vecino = copy.deepcopy(solucion_actual)
    vertice_a_modificar = random.randint(0, len(solucion_actual) - 1)
    nuevo_color = random.randint(1, n_colores)
    vecino[vertice_a_modificar] = nuevo_color
    return vecino
```

Esta función genera una solución vecina a partir de la solución actual (modifica aleatoriamente el color de un vértice en la solución actual).

```
# Función para evaluar la calidad de una solución
def evaluar_solucion(solucion):
    n_colores = max(solucion)
    if n_colores > len(solucion):
        return float('inf')
    for v1, v2 in aristas:
```

```
        if solucion[v1 - 1] == solucion[v2 - 1]:
            return float('inf')
    return n_colores
```

Esta función genera una solución vecina a partir de la solución actual, notese como es que este modifica aleatoriamente el color de un vértice en la solución actual.

```
# Función para dibujar la gráfica con la coloración dada
def dibujar_grafica_coloreada(vertices, aristas, colores):
    G = nx.Graph()
    G.add_nodes_from(vertices)
    G.add_edges_from(aristas)
    pos = nx.spring_layout(G)
    nx.draw(G, pos, with_labels=True, node_size=700,
        node_color=colores, font_size=8, font_weight='bold')
    plt.show()
```

Esta funcion es la misma que la de la tarea pasada, su funcion es dibujar la grafica final.

```
# Función de búsqueda local iterada (ILS)
def busqueda_local_iterada(archivo, max_iteraciones,
                           temperatura_inicial, factor_enfriamiento):
    n_vertices, _, _, _ = leer_ArchivoCol(archivo)
    n_colores = n_vertices
    mejor_solucion = None
    mejor_costo = float('inf')

    # Generar una solución inicial aleatoria
    solucion_actual = colorearGraficaConNColores2(archivo)
    costo_actual = evaluar_solucion(solucion_actual)

    # Búsqueda local iterada
    for _ in range(max_iteraciones):
        vecino = generar_vecino_iterada(solucion_actual, n_colores)
        costo_vecino = evaluar_solucion(vecino)

        if costo_vecino < costo_actual:
            solucion_actual = vecino
            costo_actual = costo_vecino
        else:
            probabilidad_aceptacion = temperatura_inicial / (temperatura_inicial + 1)
            if random.random() < probabilidad_aceptacion:
                solucion_actual = vecino
                costo_actual = costo_vecino

        # Actualizar la mejor solución encontrada
        if costo_actual < mejor_costo:
            mejor_solucion = solucion_actual
            mejor_costo = costo_actual

        # Enfriar la temperatura
        temperatura_inicial *= factor_enfriamiento

    return mejor_solucion
```

Esta función implementa el algoritmo de búsqueda local iterada (ILS) para encontrar una solución óptima al problema de colorear una gráfica, está empieza generando una solución inicial aleatoria, y luego explora vecinos de esta solución en busca de una solución mejor, es importante resaltar como es que la temperatura inicial y el factor de enfriamiento controlan la probabilidad de aceptar soluciones peores durante la búsqueda.

Al probar con la gráfica:

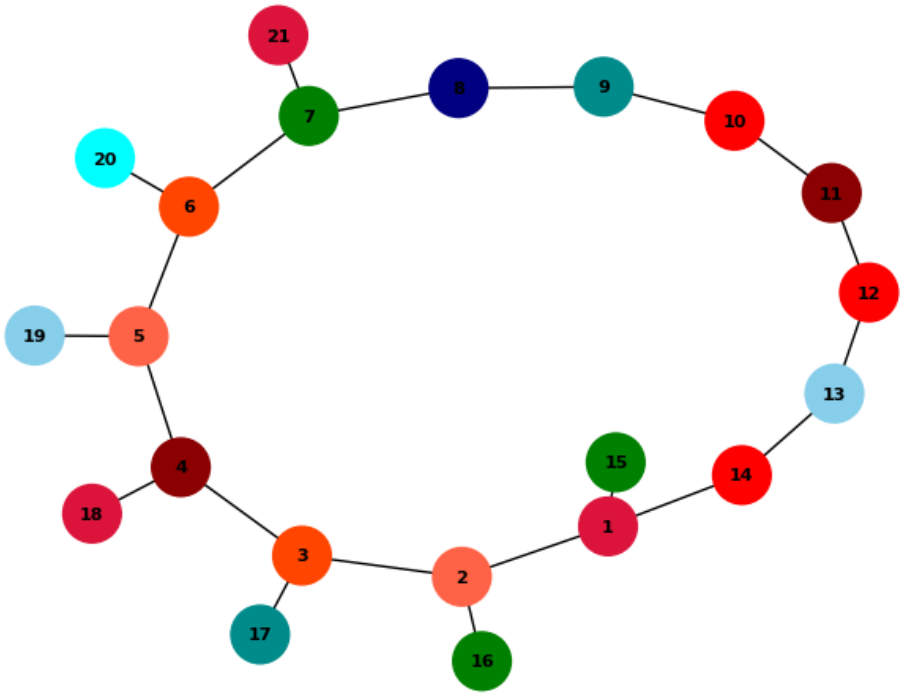
```
c Archivo: grafica_Grandota.col
p edge 21 21
e 1 2
e 2 3
e 3 4
e 4 5
e 5 6
e 6 7
e 7 8
e 8 9
e 9 10
e 10 11
e 11 12
e 12 13
e 13 14
e 14 1
e 1 15
e 2 16
e 3 17
e 4 18
e 5 19
e 6 20
e 7 21
```

Y los parámetros:

```
# Parámetros del algoritmo
max_iteraciones = 10000000
temperatura_inicial = 500000
factor_enfriamiento = 0.99999999999999
```

Tenemos como resultado:

```
Mejor solución encontrada: [9, 6, 8, 7, 6, 8, 10, 2, 4, 5, 7, 5, 3, 5, 11, 10, 4, 9, 3, 1, 9]
Número de vértices: 21
Número de aristas: 21
Número de colores: 11
```



En esta gráfica (tanto como en los otros ejemplos que testeamos), hemos podido concluir que entre mayor sea la relación de vértices y aristas en la gráfica, mayor sera la dificultad de encontrar una solucion óptima por parte del algoritmo, además de que en esto ultimo influye enormemente los valores que hemos puesto para el numero máximo de iteraciones, la temperatura inicial y el enfriamiento ya que entre mas grandes sean estos valores, podemos encontrar soluciones mas óptimas (a costa de que el tiempo para poder llegar a esas soluciones se nos dispare con valores mas grandes que los propuestos en este ejemplo).

Consideraciones Generales

➢ En el reporte deberán incluir algún comentario / discusión sobre la representación utilizada en cada uno de los problemas.

➢ Incluir archivos con un ejemplo de las soluciones aleatorias generadas para cada uno de los ejemplares (instancias) de prueba de los diferentes problemas.

Por ejemplo, para el ejercicio 1, deberán tendrán los siguientes archivos:

- ➢ `solAleatoria_Sphere_D2.txt`
- ➢ `solAleatoria_Ackley_D2.txt`
- ➢ .
- ➢ `solAleatoria_Sphere_D10.txt`
- ➢ `solAleatoria_Ackley_D10.txt`

El entregable para esta tarea deberá ser un archivo zip con la siguiente estructura:

```
+ # cuenta / <--- Nombre de la carpeta
    - src / <--- Carpeta con el código fuente de su implementación
    - output / <--- Carpeta con ejemplos de las soluciones de prueba
                    generadas
    - README.txt <--- Archivo con instrucciones para compilar y ejecutar
                      se debe incluir el comando para ejecutar un ejemplo
                      de cada inciso
    - makefile <-- [Opcional]
    - reporteT3.pdf <--- Reporte de la tarea
    - ejecuciones.csv <-- [Opcional] Hoja de cálculo con la información de las
                                     ejecuciones realizadas.
```

El reporte (reporteT3.pdf) deberá incluir al menos:

- ✪ Nombre completo
- ✪ Título y número de Tarea
- ✪ Respuestas a los ejercicios planteados
- ✪ Comentarios / Conclusiones

No hay extensión mínima ni máxima, pero deben incluir las respuestas / comentarios que se piden en cada uno de los ejercicios.

Referencias

- https://patentimages.storage.googleapis.com/a3/d7/f2/0343f5f2c0cf50/US2632058.pdf