UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

Tarea 04 - Algoritmo Genético

ASIGNATURA

Cómputo Evolutivo

PROFESOR

Oscar Hernández Constantino

AYUDANTE

Malinali Gónzalez Lara

ALUMNOS

Carlos Emilio Castañon Maldonado & Dana Berenice Hérnandez Norberto

Ejercicio 1. Optimización Continua

1 a) **Describe e implementa un algoritmo genético para las funciones de optimización continua utilizadas en la tarea 3.**

Se deben implementar y utilizar los siguientes componentes:

I) Representación binaria para las soluciones
II) Selección de padres por el método de la ruleta
III) Operador de cruza de n puntos
IV) Mutación flip
V) Reemplazo generacional con elitismo
[Se debe garantizar que la mejor solución siempre permanece en la población]

Primero generamos nuestra población aleatoria en la que realizaremos todos los calculos, la inicializamos aleatoriamente en una forma de representación binaria.

```
# Generamos una poblacion con la representacion binaria de soluciones
def poblacion(solution,nBit):
    #solution es el tamaño de la solucion que queremos generar
    #nBit es la cantidad de bit que queremos utilizar para representar
    # la poblacion generada
    return np.random.randint(2,size = (solution,nBit))
```

Ahora, por el método de la ruleta elegimos los índices de los padres con los que realizaremos la cruza de vectores binarios, para posteriormente extraer los padres de la población generada.

```
def ruleta(values):
    # Nos referimos a las aptitudes que posee cada individuo de la poblacion
    total = np.sum(values)
    probabilities = abs(values/total)
    parents = np.random.choice(np.arange(len(values)),size = 2, p = probabilities)
    return parents
```

Realizamos la función de cruza, determinando un punto de cruce desde el vector de bits de un solo padre, tomando la primera parte del vector de un padre, y la otra parte restante del otro. Así como la función de mutación, inicializando para cada bit una probabilidad aleatoria de mutar y comparándola con una probabilidad fija.

```
# Operacion de cruza de n puntos
def cruza(parent_1,parent_2):
    cruce = np.random.randint(len(parent_1)-1)
    hijo = np.concatenate((parent_1[:cruce],parent_2[cruce:]))
    return hijo

def mutacion(individuo,probability):
    for i in range(len(individuo)):
        # A cada bit del individuo le asigamos una probabilidad de ser invertido
        # solo muta si esta es mayor a un numero aleatorio generado
        if np.random.rand() < probability:
            individuo[i] = 1 - individuo[i]
    return individuo
```

Posteriormente, agregamos la función de reemplazo. Debido a que se solicita un reemplazo generacional con elitismo, extraemos los valores más pequeños debido a que queremos escapar de óptimos locales y los reemplazamos, al aplicar esta función modificamos la mitad de la generación y no la generación

completa, debido a que queremos explorar ambas soluciones, además de que reemplazar la población entera puede llegar a caer en un algoritmo similar a la búsqueda aleatoria. Agregamos una función extra para evaluar cada una de nuestras funciones fácilmente.

```
def reemplazo(population,values,new_population,new_values):
    # Con argsort podemos extraer los indices de los valores mas pequeños que tenemos
    index = np.argsort(values)[:len(new_population)]
    population[index] = new_population
    values[index]= new_values
    return population, values

#Agregamos una funcion auxiliar para evaluar las soluciones
def evalua(function,x):
    evaluation = None
    if function == 'sphere':
        evaluation = sphere(x)
    elif function == 'ackley':
        evaluation = ackley(x)
    elif function == 'griewank':
        evaluation = griewank(x)
    elif function == 'rastrigin':
        evaluation = rastrigin(x)
    elif function == 'rosenbrock':
        evaluation = rosenbrock(x)
    return evaluation
```

Ahora, aplicamos con las funciones anteriormente programadas nuestro algoritmo evolutivo, inicializando nuestra población, evaluando con nuestra función sus aptitudes. Generamos hijos para la mitad de la población, permitiéndonos de esa forma, reemplazar la mitad de la misma.

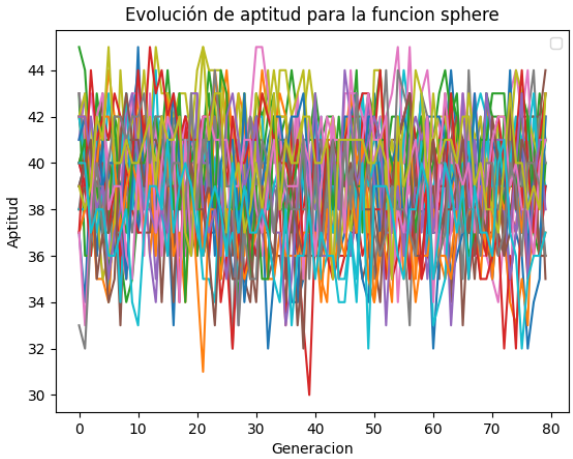
```
def algoritmo_genetico(generation,solution, nBit, probability,function):
    # generation: se refiere al número de iteraciones de realizaremos
    # solution: tamaño de la poblacion que queremos generar
    # nBit: cantidad de Bits con los que queremos representar la poblacion
    # function: funcion que queremos optimizar
    population = poblacion(solution,nBit)

    mejor_solucion = []
    mejor_evaluacion = []

    # Aplicamos el criterio de paro, para un numero determinado de generaciones
    for i in range(generation):
        evaluation = evalua(function,population)
        new_population = []
        new_evaluation = []
        #Solo reemplazamos una parte de la poblacion, no toda
        for j in range(solution // 2):
            parents = ruleta(evaluation)
            padre_1,padre_2 =  population[parents]
            hijo = cruza(padre_1, padre_2)
            muta = mutacion(hijo, probability)
            value = evalua(function,muta)
            new_evaluation.append(value)
            new_population.append(muta)

        new_population = np.array(new_population)
```

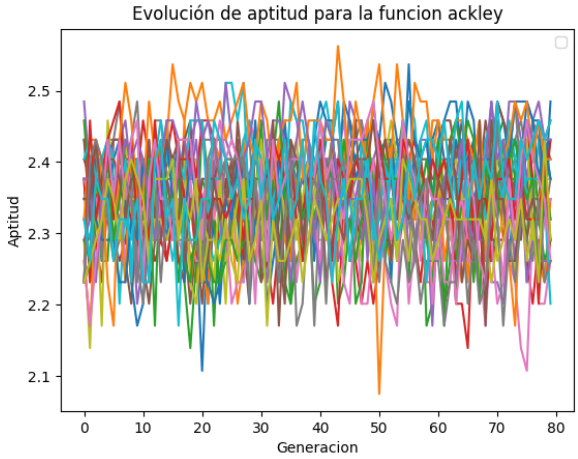
```
        population, evaluation = reemplazo(population,evaluation,new_population,new_evaluation)
        index = np.argsort(evaluation)[0]
        mejor_evaluacion.append(min(evaluation))
        mejor_solucion.append(population[index])

    return mejor_evaluacion,mejor_solucion
```

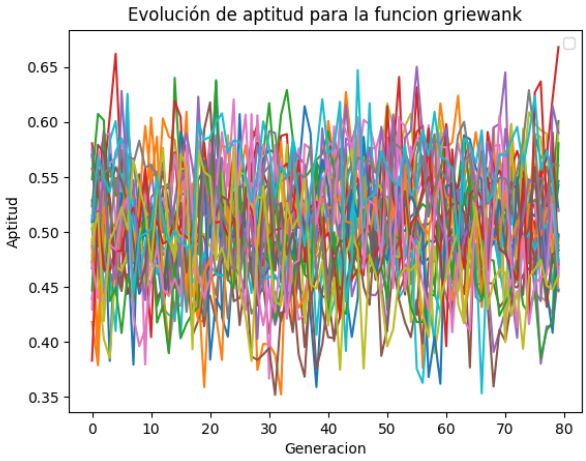
1 b) **Genera al menos una gráfica de evolución de aptitud ***, con una ejecución para cada una de las funciones de prueba.**

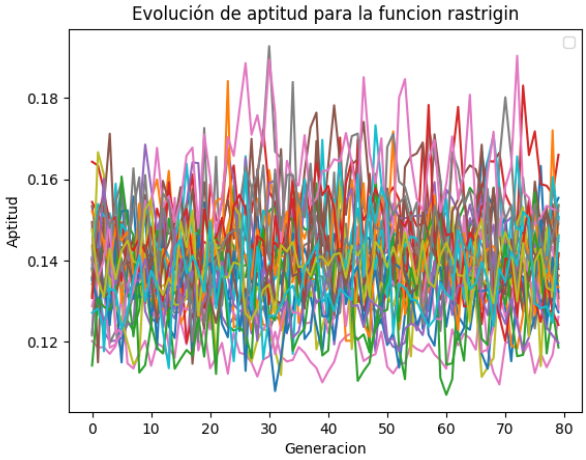
Realizando 30 ejecuciones obtenemos las siguientes gráficas. Con estos gráficos podemos visualizar como tenemos la oportunidad de encontrar y salir de mínimos locales y obtener nuevos. En su mayoría encuentran nuevas soluciones mínimas, con excepción de la función rastringin, la cual oscila en su mayoría en mínimos, obteniendo más bien, soluciones mayores.

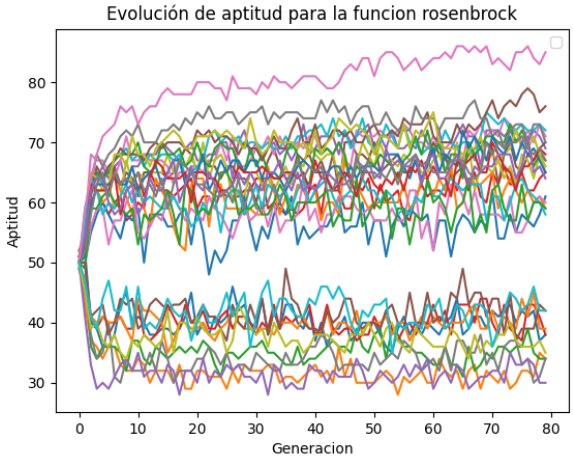
Otro caso interesante es la función rosenbreck, en esta función vemos dos tipos de comportamiento, ascendente y descendente, posiblemente debido a la naturaleza ruidosa mucho ruido de la función.






Evolución de aptitud para la funcion griewank
Aptitud
Generacion
0.65
0.60
0.55
0.50
0.45
0.40
0.35
0 10 20 30 40 50 60 70 80


Evolución de aptitud para la funcion rastrigin
Aptitud
Generacion
0.18
0.16
0.14
0.12
0 10 20 30 40 50 60 70 80


Evolución de aptitud para la funcion rosenbrock
Aptitud
Generacion
80
70
60
50
40
30
0 10 20 30 40 50 60 70 80

1 c) **Ejecutar el algoritmo genético al menos 30 veces para cada función de prueba, e incluir una tabla con los resultados estadísticos (mejor, peor y valor promedio en cada función). * En cada ejecución se debe utilizar una semilla diferente para el generador de números aleatorios.**

Con el fin de visualizar algunas diferencias en estas soluciones, inicializamos la población representada con 10 y 100 bits. En general, los resultados mejoraron con el algoritmo genético, permitiéndonos llegar al mínimo local en la mayoría de las funciones evaluadas, tras utilizar 100 generaciones y 30 iteraciones del algoritmo. Cabe mencionar la importancia del uso de cierta cantidad de bits, permitiéndonos darnos cuenta de que no siempre poseer más bit para representar una solución, es lo más óptimo.

Función	Mejor valor	Valor promedio	Peor valor
Sphere 10	0	3.58	7
Sphere 100	30	46	39.45
Ackley 10	0	2.14	3.27
Ackley 100	1.93	2.35	2.53
Griewank 10	0	0.37	0.77
Griewank 100	0.36	0.52	0.67
Rastrigin 10	0	0.33	0.82
Rastrigin 100	0.1	0.14	0.2
Rosenbrock 10	0	7.04	10
Rosenbrock 100	29	53.24	78

Ejercicio 2. Coloración en Gráficas

2 a) **Describe e Implementa un algoritmo genético para el problema de optimización de gráficas.**

Tomando como base lo que hemos venido trabajando en otras tareas, primero generaremos una población inicial enteramente aleatoria con la cual trabajaremos:

```
def generar_poblacion_inicial(n_vertices, tamano_poblacion):
    poblacion = []
    for _ in range(tamano_poblacion):
        solucion = SColoracion(n_vertices)
        for vertice in range(1, n_vertices + 1):
            color = random.randint(1, n_vertices)
            solucion.asignar_color(vertice, color)
        poblacion.append(solucion)
    return poblacion
```

Ahora, ya que tenemos nuestra población inicial, a continuación definiremos nuestra funcion de aptitud (fitness), la cual es hasta cierto punto sencilla ya que esta será hecha en base a que queremos a la gráfica con el menor numero de colores (que cumpla claro con las condiciones de la n-coloración), quedando esta de esta forma:

```
def evaluar_aptitud(solucion):
    n_colores = max(solucion.colores_asignados)
    for v1, v2 in aristas:
        if solucion.obtener_color(v1) == solucion.obtener_color(v2):
            return float('inf')
    return n_colores
```

Una vez definido lo anterior ahora podemos realizar nuestra funcion de torneo encargada de enfrentar unas soluciones con otras y quedarnos con la de menor coloración:

```
def seleccion_torneo(poblacion, tamano_torneo):
    seleccionados = []
    for _ in range(len(poblacion)):
        torneo = random.sample(poblacion, tamano_torneo)
        mejor_solucion = min(torneo, key=lambda x: evaluar_aptitud(x))
        seleccionados.append(mejor_solucion)
    return seleccionados
```

Definimos la mutación, la cual será aleatoria:

```
def mutacion(solucion, tasa_mutacion):
    for vertice in range(1, solucion.n_vertices + 1):
        if random.random() < tasa_mutacion:
            nuevo_color = random.randint(1, solucion.n_vertices)
            solucion.asignar_color(vertice, nuevo_color)
    return solucion
```

Ahora sobre un punto definimos el cruzamiento:

```
def cruzamiento(poblacion, tasa_cruzamiento):
    descendencia = []
    for i in range(0, len(poblacion), 2):
        padre1 = poblacion[i]
        padre2 = poblacion[i + 1]
        if random.random() < tasa_cruzamiento:
            punto_corte = random.randint(1, padre1.n_vertices - 1)
            descendiente1 = SColoracion(padre1.n_vertices)
            descendiente2 = SColoracion(padre2.n_vertices)
            descendiente1.colores_asignados = padre1.colores_asignados[:punto_corte] + padre2.col
            descendiente2.colores_asignados = padre2.colores_asignados[:punto_corte] + padre1.col
            descendencia.extend([descendiente1, descendiente2])
        else:
            descendencia.extend([padre1, padre2])
    return descendencia
```

De manera auxiliar definimos esta funcion para poder saber cual es la mejor solucion en todas las iteraciones del genético:

```
def mejor_solucion(poblacion):
    return min(poblacion, key=lambda x: evaluar_aptitud(x))
```

Con todo lo anterior definido ahora podemos definir nuestro algoritmo genetico de la siguiente manera:

```
def algoritmo_genetico(archivo, tamano_poblacion, tamano_torneo, tasa_cruzamiento, tasa_mutacion
    n_vertices, _, _, _ = leer_ArchivoCol(archivo)
    poblacion = generar_poblacion_inicial(n_vertices, tamano_poblacion)
    mejor_solucion_actual = mejor_solucion(poblacion)
    mejor_aptitud = evaluar_aptitud(mejor_solucion_actual)
    aptitudes = [mejor_aptitud]
    for generacion in range(max_generaciones):
        poblacion = seleccion_torneo(poblacion, tamano_torneo)
        poblacion = cruzamiento(poblacion, tasa_cruzamiento)
        for i in range(len(poblacion)):
            poblacion[i] = mutacion(poblacion[i], tasa_mutacion)
        mejor = mejor_solucion(poblacion)
        mejor_aptitud_generacion = evaluar_aptitud(mejor)
        aptitudes.append(mejor_aptitud_generacion)
        if mejor_aptitud_generacion < mejor_aptitud:
            mejor_solucion_actual = mejor
            mejor_aptitud = mejor_aptitud_generacion
        if mejor_aptitud == 1:
            break
    plt.plot(range(generacion + 2), aptitudes)
    plt.xlabel('Generacion')
    plt.ylabel('Aptitud')
    plt.show()
    return mejor_solucion_actual
```

2 b) **Ejecuta el algoritmo y compara con los resultados de tareas anteriores**
Sea: `grafica_Grandota` lo siguiente:

```
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
```

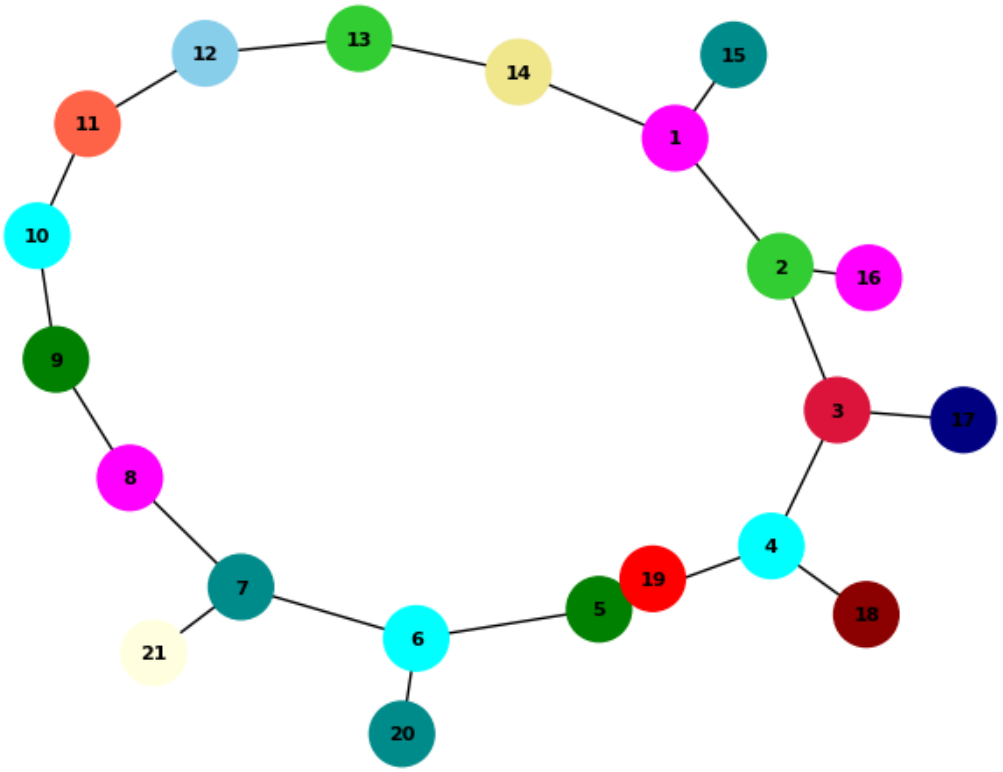
Tendremos los siguientes resultados con las siguientes configuraciones:

- **Solución Aleatoria**
- **Solución por Escalada**
- **Búsqueda Local Iterada**
- **Algoritmo Genético**

Solución Aleatoria:

```
archivo = 'grafica_Grandota.col'
n_vertices, n_aristas, vertices, aristas = leer_ArchivoCol(archivo)

print("Solucion Aleatoria:")
solucion = colorearGraficaConNColores(archivo)
dibujar_Grafica_coloreado(solucion, vertices, aristas)
# Imprimimos el numero de colores usados (sin repetidos)
# en la solucion aleatoria
print("Numero de colores usados:", len(set(solucion.colores_asignados)))

print("Solucion por escalada:")
solucionEscalada = busquedaEscalada(archivo)
dibujarsolucionEscalada(solucionEscalada, vertices, aristas)
# Imprimimos el numero de colores usados (sin repetidos) en
# la solucion por escalada
print("Numero de colores usados:",
len(set(solucionEscalada.colores_asignados)))
```

```
Solucion Aleatoria:
```

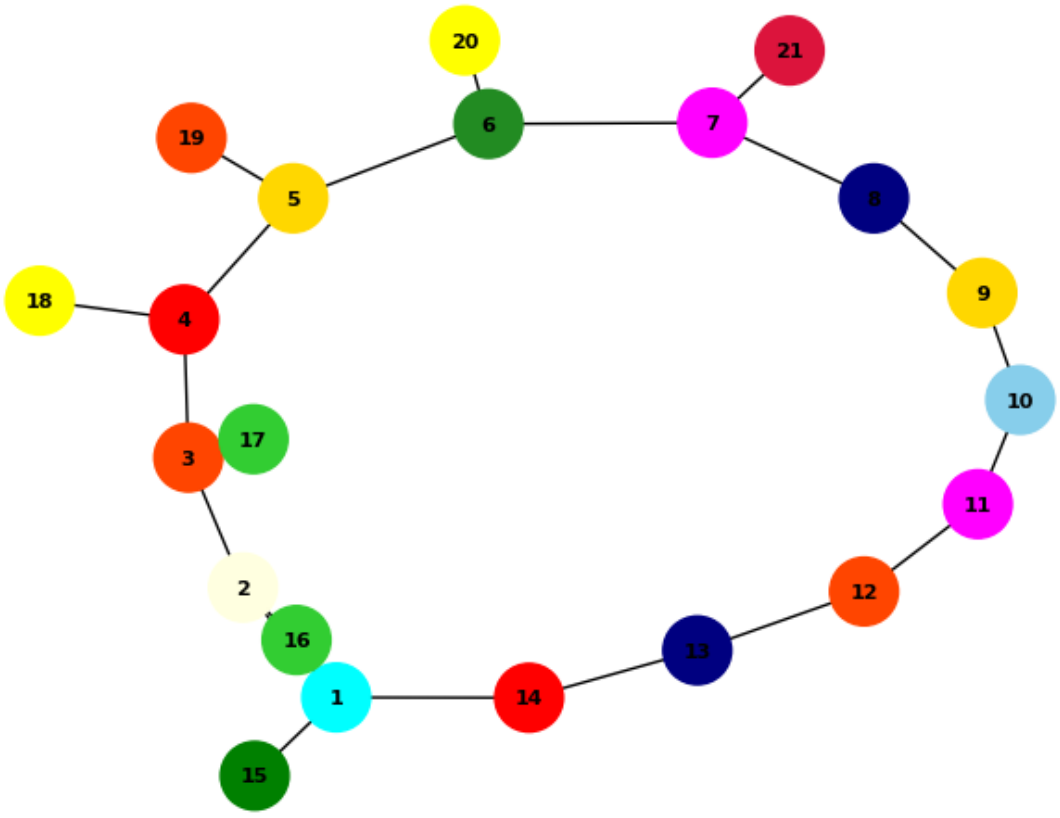



```
Numero de colores usados: 15
```

Solución por Escalada:

```
archivo = 'grafica_Grandota.col'
n_vertices, n_aristas, vertices, aristas = leer_ArchivoCol(archivo)

print("Solucion por escalada:")
solucionEscalada = busquedaEscalada(archivo)
dibujarsolucionEscalada(solucionEscalada, vertices, aristas)
# Imprimimos el numero de colores usados (sin repetidos)
en la solucion por escalada
print("Numero de colores usados:",
len(set(solucionEscalada.colores_asignados)))
```

```
Solucion por escalada:
```

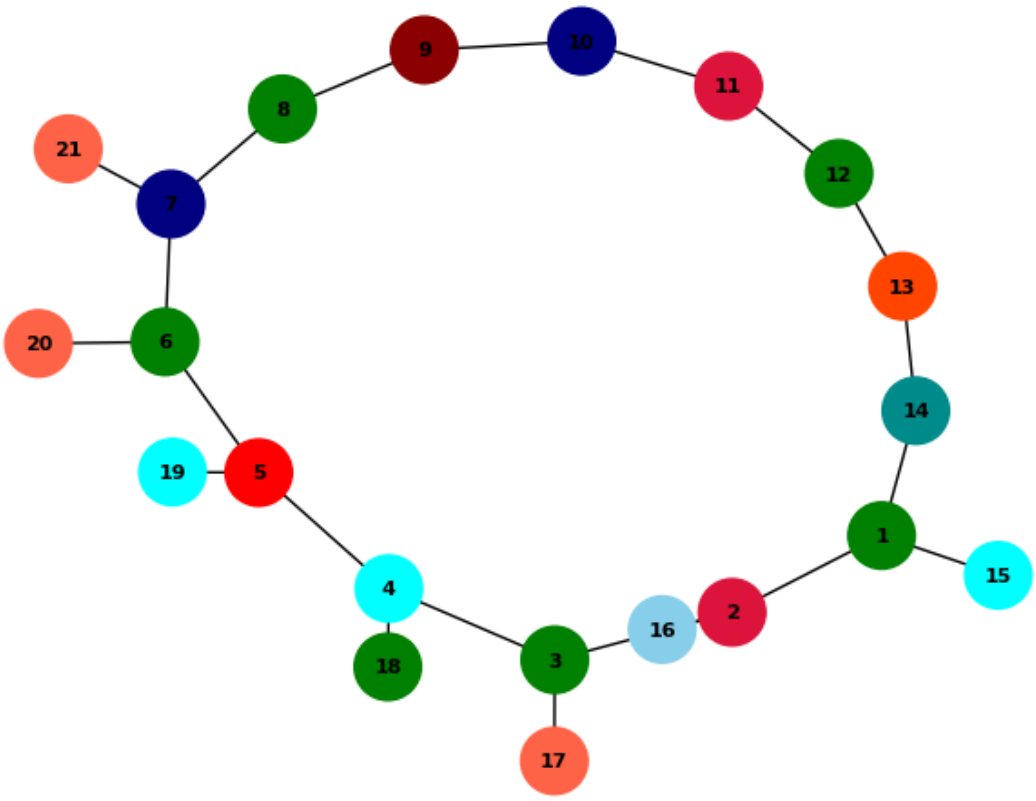



```
Numero de colores usados: 13
```

Búsqueda Local Iterada:

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

archivo = 'grafica_Grandota.col'
n_vertices, _, vertices, aristas = leer_ArchivoCol(archivo)

mejor_solucion = busqueda_local_iterada(archivo, max_iteraciones,
temperatura_inicial, factor_enfriamiento)
print("Mejor solucion encontrada:", mejor_solucion)
print("Numero de vertices:", n_vertices)
print("Numero de aristas:", len(aristas))
print("Numero de colores:", max(mejor_solucion))
colores = [mapear_color(color) for color in mejor_solucion]
dibujar_grafica_coloreada(vertices, aristas, colores)
```

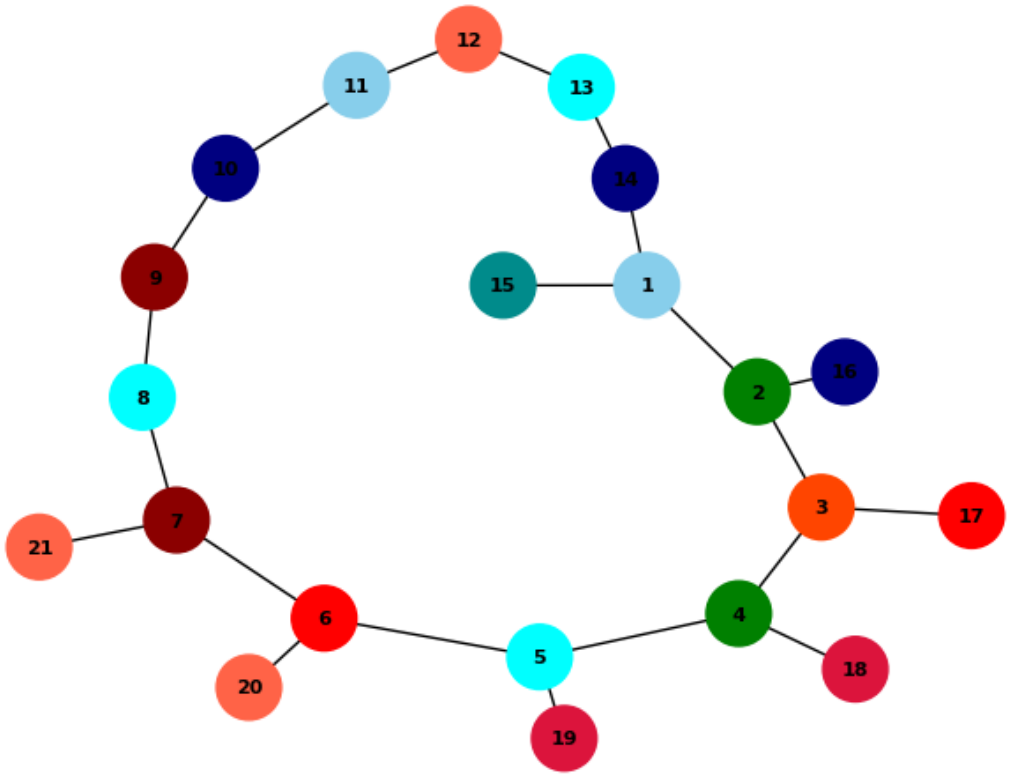
```
Numero de vertices: 21
Numero de aristas: 21
Numero de colores: 11
```



Algoritmo Genético:

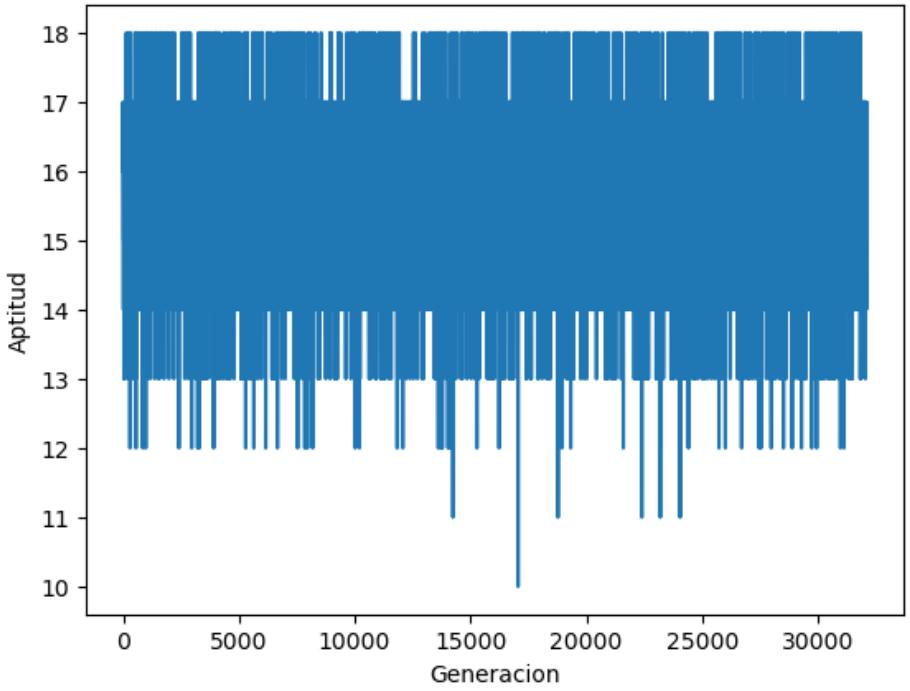
```
# Parametros del algoritmo genetico
archivo = 'grafica_Grandota.col'
tamano_poblacion = 1000
tamano_torneo = 10
tasa_cruzamiento = 0.999
tasa_mutacion = 0.82
max_generaciones = 32099

# Ejecucion del algoritmo genetico
solucion = algoritmo_genetico(archivo, tamano_poblacion,
tamano_torneo, tasa_cruzamiento, tasa_mutacion, max_generaciones)

# Visualizacion de la solucion
print("Numero de colores utilizados:", evaluar_aptitud(solucion))

# Convertir los colores de la solucion a un formato
# compatible con la funcion de dibujo
colores = [mapear_color(color) for color in solucion.colores_asignados]

# Dibujar la grafica con la coloracion resultante
dibujar_grafica_coloreada(vertices, aristas, colores)
```

```
Numero de colores utilizados: 10
```



2 c) **Incluye una tabla con la comparación de resultados, y gráficas de evolución de aptitud para al menos un ejemplar.**

Para la gráfica anterior, tenemos lo siguiente:



Notese como es que en todas esas generaciones hubo una que encontró una solucion de 10 colores la cual fue la que devolvió el algoritmo por ser la mas optima en todas las generaciones generadas.

Conclusiones:

Como podemos notar las soluciones menos eficientes fueron la Solución Aleatoria seguida de la Solución por Escalada, con la Búsqueda Local Iterada y el Algoritmo Genético pasa algo curioso, dependiendo de los parámetro que le pasemos, obtendremos buenos resultados (o tendremos resultados ineficientes), por ejemplo con la Búsqueda Local Iterada para tener buenos resultados tuvimos que poner valores muy altos de iteraciones, temperatura inicial y de enfriamiento, mientras que en el genético si bien también pusimos valores algo altos, no llegamos a una cantidad tan alta como en la búsqueda local iterada, además de que con el genético fue con el que mejores resultados tuvimos de todos los métodos antes mencionados.

Consideraciones Generales

➢ El reporte debe considerar el pseudocódigo de todos los componentes utilizados, una justificación de los parámetros seleccionados para la ejecución del algoritmo, comentarios de los resultados obtenidos (tabla y gráficas) así como conclusiones generales.

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 gráficas y cualquier otro archivo
                    generado
    - README.txt <--- Archivo con instrucciones para compilar y ejecutar
                      se debe incluir el comando para ejecutar un ejemplo
                      de cada inciso
    - makefile <-- [Opcional]
    - reporteT4.pdf <--- Reporte de la tarea
    - ejecuciones.csv <-- [Opcional] Hoja de cálculo con la información de las
                                     ejecuciones realizadas.
```

El reporte (reporteT4.pdf) deberá incluir al menos:

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

El formato para el reporte es libre. Pueden usar Word, LaTeX, o cualquier otro procesador; es obligatorio que se incluya el archivo pdf. No hay extensión mínima ni máxima, aunque se sugiere considerar un reporte entre 3-5 páginas, pero deben incluir las respuestas / comentarios que se piden en cada uno de los ejercicios.

Referencias

- https://www.sfu.ca/ssurjano/index.html
- https://patentimages.storage.googleapis.com/a3/d7/f2/0343f5f2c0cf50/US2632058.pdf