Tarea 3 - Metaheurísticas de Trayectoria

1. **Recocido simulado**

(a) Describe e implementa un operador de vecindad para soluciones binarias.
- La función solo debe generar un vecino de manera aleatoria.
Descripción de operador vecindad:
Las soluciones binarias que manejamos son arreglos de números binarios y sus vecinos serán el mismo arreglo binario pero con alguno de sus números cambiados. Si nuestro arreglo es de n bits, entonces tendremos n vecinos, lo siguiente es saber qué vecino elegiremos para que sea aleatorio.
La forma en la que vamos a generar un vecino aleatorio es generando un índice aleatoria que esté en el rango de los índices del arreglo, luego vamos a cambiar el valor en dicho índice (cambiarlo a 1 si es 0 y a 0 si es 1) y ese será nuestro vecino aleatorio.

```java
public int[] vecinoRnd(int[] x) {
    Random random = new Random();
    int[] vecino = x.clone();
    int indiceAleatorio = random.nextInt(x.length);
    vecino[indiceAleatorio] = 1 - x[indiceAleatorio];
    return vecino;
}
```

Figure 1: Operador de vecino aleatorio binario

Como java es orientado a objetos, para generar el vecino usamos el método **clone()** para generar un objeto completamente distinto que será nuestro vecino, ya que si pusiéramos una igualdad, entonces vecino y solución actual apuntarían a un mismo objeto.

(b) Describe e implementa un esquema de enfriamiento.
- Justifica tu elección del esquema de enfriamiento.
No tenemos una justificación mas que a prueba y error.
Inicialmente habíamos implementado el enfriamiento lineal, pero aunque pusiéramos una temperatura de millones, aún así el algoritmo duraba poco en ejecución y no llegaba a tan buenos resultados.
Intentamos subir el número de operaciones o usar el módulo para que la temperatura bajara cada cierto número de iteraciones. Al final no logramos hacer que funcionara para nuestro problema.
Luego implementamos el enfriamiento geométrico (también llamado enfriamiento exponencial) con valor de $\alpha = 0.999$, esto nos dio los mejores resultados hasta el momento, el único problema fue que nos dimos cuenta que aunque pusiéramos la temperatura muy alta, daba los mismos resultados que una temperatura más

baja, es decir después de cierta iteración ya solo realizaba búsqueda local.
Este esquema de enfriamiento queda de la forma:

$$temp_{k+1} = temp_k * 0.999$$

(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]

```java
public double[] recocido(int numFun, int dimension) {
    Binario bin = new Binario();
    Evaluador eval = new Evaluador();
    double[] intervalo = eval.intervalo(numFun);

    // solución inicial y temperatura inicial
    int[] solucion = bin.generaSolucionAleatoria(NBITS * dimension);
    double temp = 10000;
    int numIt = 0;

    // condición de término
    while (numIt < MAXITER && temp > 0) {
        numIt++;

        // seleccionamos un vecino
        int[] vecino = bin.vecinoRnd(solucion);
        double[] valores = evaluaBin(solucion, vecino, numFun, intervalo);
        // si es mejor lo aceptamos automáticamente
        if (valores[1] < valores[0]) {
            solucion = vecino;
        } else {
            // si no es mejor lo aceptamos con cierta probabilidad
            double proba = Math.exp((valores[0] - valores[1]) / temp);
            double rnd = Math.random();
            if (rnd < proba) {
                solucion = vecino;
            }
        }

        // actualizamos temperatura
        temp = temp * 0.999;

    }
    return bin.decodifica(solucion, NBITS, intervalo[0], intervalo[1]);
```

Figure 2: Algoritmo de recocido simulado implementado

(d) Investiga en qué consiste la codificación de gray. ¿Qué ventajas o desventajas podría tener respecto a la representación implementada en la tarea 2?
De acuerdo a lo que estuve leyendo, la codificación de gray tiene la particularidad de que cuando una cifra binaria aumenta o decrementa en uno, la cifra binaria solo va a cambiar un solo número. [Adm23]
En dicha página se menciona que la mayor ventaja es evitar posibles pérdias, de

posicionamiento.
A mí lo que se me ocurre es que al momento de hacer la vecindad, cambiar un solo dígito nos podría ayudar a que la vecindad genere valores más cercanos con una evaluación distinta, eso podría ayudar a la búsqueda local cuando la temperatura ya sea baja.
Aunque por otro lado eso sería una desventaja cuando la temperatura fuera alta, ya que podría no alejarnos tanto al momento de hacer la exploración. Lo que se traduce a que quizá no pudiéramos escapar de los óptimos locales.

2. **Experimentación**
En este ejercicio vamos a comparar los resultados obtenidos por la búsqueda aleatoria y el recocido simulado, en las funciones de pruebas anteriores.

   (a) Propón un criterio de término que permita establecer una comparación justa (en cuanto al uso de recursos) entre los dos algoritmos.
   El criterio propuesto por el equipo fue el número de iteraciones.
   Inicialmente pensamos en poner el mismo número de iteraciones a ambos como criterio de término, intentamos con un millón de iteraciones, pero nos dimos cuenta que el recocido simulado después de cierto número de iteraciones ya no mejoraba, ya que se quedaba estancado en óptimos locales.
   Entonces decidimos probar con cien mil iteraciones para el recocido simulado y resultó que nos daba los mismos resultados que con un millón de iteraciones.
   Entonces para que la comparación sea justa de acuerdo al número de iteraciones, debemos correr 10 veces el recorrido simulado para igualar las iteraciones que hace la búsqueda aleatoria.
   Es una comparación más o menos justo porque el recocido simulado realiza muchas más operaciones en cada iteración.

   (b) 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.

Función	Mejor BA	Promedio BA	Peor BA	Mejor RC	Promedio RC	Peor RC
Sphere	4.54240	87.37550	209.95746	1.49×10^{-11}	1.49×10^{-11}	1.49×10^{-11}
Ackley	13.86400	20.95400	22.08995	2.86×10^{-5}	2.86×10^{-5}	2.86×10^{-5}
Griwank	19.07111	300.99936	715.12404	0.024	0.254	0.582
Rastrign	38.62993	185.32163	336.48723	6.216	16.370	26.494
Rosenbrock	75.06251	4446.55986	19781.91129	6.3221	28.533	79.272

Table 1: Comparación de valores para diferentes funciones utilizando los algoritmos BA (Búsqueda aleatoria) y RC (Recocido Simulado)

Análisis de la tabla:
De esta tabla tenemos muchas cosas que decir, lo haremos analizando función por función:

i. **Sphere**
Aunque pareciera que el mejor, el peor y el promedio con recocido simulado son exactamente iguales, eso ocurre porque no pude poner todos los decimales que tenía cada solución por el espacio que ocuparían en la tabla.
No son iguales pero su diferencia es tan pequeña que se empieza a ver hasta después de los primeros 18 decimales de izquierda a derecha.
Por otro lado, sobre la búsqueda aleatoria: aunque se podría pensar que no es tanta la diferencia entre el mejor de búsqueda aleatoria y el mejor con recocido simulado, si dividimos el mejor de BA con el mejor de RC:

$$\frac{4.54}{1.490116830361267 \times 10^{-11}} \approx 3.04286010475 \times 10^{11}$$

Es decir, la evaluación de la mejor solución encontrada por la búsqueda aleatoria es trescientos cuarenta y dos mil ochocientos sesenta millones de veces más grande que la evaluación de la mejor solución encontrada por el recocido simulado.

ii. **Ackley**
Contrario a lo que menciono en sphere, en este el mejor, el promedio y el peor me dieron exactamente lo mismo, y la solución solo cambiaba el signo de algunos, lo cual es entendible ya que tanto el coseno (función par) y la potencia cuadrada en la función de Ackley, hacen que ese signo no importe.
Entonces a la conclusión que llegué fue a que siempre llega al óptimo global, al menos el óptimo global que permite la partición, ya que el cero no es representable.
Otra cosa destacable es que nos podemos dar cuenta que en este caso la partición y el intevalo ayudan mucho, ya que el mejor, el peor y el promedio no son tan distintos en la búsqueda aleatoria.
Aún así comparando la solución con el recocido simulado, no se acerca ni un poco en lo preciso que es respecto a acercarse al óptimo global.

iii. **Griewank, Rastrign y Rosenbrock**
En estas tres encontramos cosas que las hacen parecidas, ya que las tres comparten que la búsqueda aleatoria tiene una gran diferencia entre el mejor valor, el valor promedio y el peor valor, cosa que también se ve reflejado cuando se resuelve por recocido simulado.
Esto a mí me sugiere que el espacio de búsqueda con las particiones y el intervalo fue mayor a las funciones anteriores.
También notamos que el mejor valor del recocido simulado en la función Rastrign y Rosenbrock no tiene un buen acercamiento a su óptimo global, por lo que estas funciones no se pudieron optimizar de la mejor forma, aunque en general sí mejora a una búsqueda aleatoria.

Quizá se podría optimizar de una mejor forma manipulando el término, la temperatura, el esquema de enfriamiento, etc. Para cada una de las funciones

¿Sería suficiente con ejecutar una vez cada uno de los algoritmos, en cada función, para poder establecer una comparación? ¿Por qué? Justifica tu respuesta
No sería suficiente, porque en el recocido simulado elegimos como término $100,000$ iteraciones como condición de término y en la búsqueda aleatoria hacemos $1,000,000$ de iteraciones.
Por lo que para igualar la cantidad de iteraciones tenemos que correr 10 veces el algoritmo de recocido simulado.
Ya con la misma cantidad de iteraciones podríamos decir que es un comparación justa.

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]

   (a) Describe los componentes implementados
   Componentes:

   i. **Solución inicial**
   Nuestra solución inicial la generamos aleatoriamente, es decir, asignamos un color aleatorio a cada vértice de la gráfica, considerando las restricciones, es decir, los vértices adyacentes no utilizan el mismo color.

   ii. **Estrategia de búsqueda local**
   Utilizamos búsqueda por escalada para realizar la búsqueda local, ésta se realiza generando soluciones vecinas a partir de una solución aleatoria, las soluciones vecinas se generan contando la frecuencia de cada color en la gráfica, una vez obtenido un arreglo que ordena los colores del más frecuente al menos frecuente, seleccionamos un vértice aleatorio y le asignamos el color más frecuente tratando de maximizar su uso, en caso de que algún vecino ya sea de ese color, tratamos de asignar el segundo color más frecuente, así sucesivamente.

   iii. **Perturbación**
   Para perturbar la solución obtenida en la búsqueda local y escapar del mínimo local, nuestro método de perturbación selecciona un subconjunto aleatorio de vértices de la gráfica para cambiarles el color, siguiendo con las reglas del coloreado. Ésto podemos hacerlo más débil o más fuerte dependiendo del porcentaje de vértices que seleccionemos, en este caso nosotros seleccionamos el treinta porciento de los vértices, pero si seleccionamos un porcentaje menor, la perturbación puede ser demasiado débil y no sacarnos del mínimo local, si seleccionamos un porcentaje mayor, la perturbación puede ser demasiado fuerte y llevarnos a una gráfica totalmente distinta que no hemos optimizado anteriormente.

   iv. **Evaluación**
   Para evaluar nuestras soluciones, utilizamos un método *evaluarSolucion* que

cuenta el número de colores utilizados para colorear la gráfica, en un principio la solución perturbada debe ser un poco peor a la ya optimizada con búsqueda iterada, por lo tanto comparamos las soluciones una vez la solución perturbada sea optimizada con el método de búsqueda iterada, es decir, comparamos las soluciones una vez ambas hayan sido optimizadas y la que utilice menos colores la asignamos a nuestra nueva mejor solución.

v. **Criterio de termino**
Para que nuestra búsqueda local iterada termine, deben cumplirse cierto número de iteraciones sin mejora, es decir, las iteraciones donde se encuentra una mejor solución no son tomadas en cuenta, puesto que nuestro objetivo es llegar al mínimo global mientras sea posible. Cuando se ha llegado al número de iteraciones permitido, el programa devuelve la configuración de colores y la evaluación de la misma.

vi. **Cambios en el código anterior**
Para realizar la búsqueda local iterada, fue necesario realizar algunos cambios al código de la tarea anterior, estos cambios se enuncian a continuación:

Método encontrarColorValido
Éste método nos simplifica el código puesto que ésta verificación se utiliza en métodos como *solucionAleatoria* y *perturbarSolucion*.

Algorithm 1 Encontrar color válido

```
function encontrarColorValido(vertice, solucion)
    rand ← Random
    coloresInvalidos ← arreglo de booleanos de tamaño numVertices + 1
    for i ← 0 to numVertices − 1 do
        if matrizAdyacencia[vertice][i] == 1 then
            coloresInvalidos[solucion[i]] ← verdadero
        end if
    end for
    repeat
        color ← 1+ rand.nextInt(numVertices)
    until not coloresInvalidos[color]
    return color
end function
```

Método generarSolucionVecina
Éste método que se utiliza para nuestra búsqueda local, que en este caso es búsqueda por escalada, fue modificado al momento de colorear el vértice aleatorio con el color más frecuente, anteriormente si no se podía pintar con el más frecuente entonces se le asignaba un color aleatorio válido, ahora se colorea con los colores de mayor a menor frecuencia.

Algorithm 2 Generar solución vecina

```
1: function GenerarSolucionVecina(solucionActual)
2:     rand ← New(Random)
3:     solucionVecina ← solucionActual.clone()
4:     vertice ← rand.nextInt(numVertices)
5:     coloresUsados ← new boolean[numVertices + 1]
6:     for i ← 0 to numVertices − 1 do
7:         if matrizAdyacencia[vertice][i] = 1 then
8:             coloresUsados[solucionActual[i]] ← true
9:         end if
10:    end for
11:    frecuenciasColores ← new int[numVertices + 1][2]
12:    for i ← 1 to numVertices do
13:        frecuenciasColores[i][0] ← i
14:        frecuenciasColores[i][1] ← 0
15:    end for
16:    for color in solucionActual do
17:        if color ≠ 0 then
18:            frecuenciasColores[color][1] + +
19:        end if
20:    end for
21:    Sort(frecuenciasColores, by frequency descending)
22:    for i ← 0 to numVertices − 1 do
23:        colorCandidato ← frecuenciasColores[i][0]
24:        if not coloresUsados[colorCandidato] then
25:            solucionVecina[vertice] ← colorCandidato
26:            break
27:        end if
28:    end for
29:    return solucionVecina
30: end function
```

Método busquedaPorEscaladaconInicial
Realiza lo mismo que el método busquedaPorEscalada de la tarea anterior, la única diferencia es que ahora inciamos con una solución pasada como parámetro, esto apoya a la memoria que debe conservar la búsqueda local iterada, donde guardamos la mejor solución al momento y después utilizamos la búsqueda por escalada, así no comenzamos desde solución aleatoria sino con una solución ya optimizada.

Método perturbarSolucion
Nuestro método de perturbación fue diseñado para corresponder con nuestra representación de soluciones, que es un arreglo donde cada celda representa un vértice de la gráfica y el valor que almacena el color, es por ello que la idea fue seleccionar un porcentaje de vértices al azar, es decir, celdas aleatorias que no se repiten y le cambiamos el color a otro que no sea utilizado por sus adyacencias, es decir, cambiamos el número almacenado en esa celda.

Algorithm 3 Perturbar Solución

```
1: function PerturbarSolucion(solucion)
2:     solucionPerturbada ← solucion.clone()
3:     rand ← New(Random)
4:     vertices ← new ArrayList()
5:     for i ← 0 to longitud(solucionPerturbada) - 1 do
6:         vertices.add(i)
7:     end for
8:     Collections.shuffle(vertices, rand)
9:     for i ← 0 to longitud(solucionPerturbada) / 30 - 1 do
10:        vertice ← vertices.get(i)
11:        solucionPerturbada[vertice] ← EncontrarColorValido(vertice, solucionPerturbada)
12:    end for
13:    return solucionPerturbada
14: end function
```

(b) **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.

Realizamos pruebas con el archivo *jean.col*, que es una instancia del problema de coloración con ochenta vértices y quinientas ocho aristas, del sitio web *https* : *//mat.tepper.cmu.edu/COLOR/instances.html* Realizamos la búsqueda por escalada con tolerancia de 1000 iteraciones para tratar de encontrar el mínimo global, que es 10. Lo cual resultó en lo siguiente

```
Primer solución aleatoria: [12, 26, 65, 45, 62, 73, 61, 64, 59, 51,
20, 13, 67, 14, 17, 24, 8, 78, 56, 6, 9, 40, 37, 6, 69, 2, 45, 72,
```

```
45, 62, 39, 79, 51, 69, 15, 33, 52, 3, 30, 71, 52, 42, 56, 61, 59,
6, 44, 65, 60, 24, 70, 5, 68, 18, 40, 58, 55, 4, 39, 33, 46, 13, 40,
46, 44, 28, 27, 21, 31, 50, 37, 47, 23, 79, 25, 51, 13, 53, 31, 35]

Evaluación de la primer solución aleatoria: 55

Solución vecina generada: [12, 26, 65, 45, 62, 73, 61, 64, 59, 51,
20, 13, 67, 14, 17, 24, 8, 78, 56, 6, 9, 40, 37, 6, 69, 2, 45, 72,
45, 62, 39, 79, 51, 69, 15, 33, 52, 3, 30, 71, 52, 42, 56, 61, 59,
6, 44, 65, 60, 24, 70, 5, 68, 18, 40, 58, 55, 4, 39, 33, 46, 13, 40,
46, 44, 28, 27, 21, 31, 50, 37, 6, 23, 79, 25, 51, 13, 53, 31, 35]

Evaluación de la solución vecina: 54

Solución más optimizada encontrada después de 29 iteraciones
y 1000 iteraciones sin mejora.

Vértice 1: Color 78
Vértice 2: Color 40
Vértice 3: Color 76
Vértice 4: Color 76
Vértice 5: Color 5
Vértice 6: Color 7
Vértice 7: Color 78
Vértice 8: Color 31
Vértice 9: Color 7
Vértice 10: Color 7
Vértice 11: Color 40
Vértice 12: Color 19
Vértice 13: Color 5
Vértice 14: Color 17
Vértice 15: Color 28
Vértice 16: Color 5
Vértice 17: Color 68
Vértice 18: Color 18
Vértice 19: Color 17
Vértice 20: Color 19
Vértice 21: Color 80
Vértice 22: Color 29
Vértice 23: Color 18
Vértice 24: Color 80
Vértice 25: Color 64
Vértice 26: Color 29
Vértice 27: Color 5
Vértice 28: Color 47
```

```
Vértice 29: Color 44
Vértice 30: Color 28
Vértice 31: Color 63
Vértice 32: Color 64
Vértice 33: Color 29
Vértice 34: Color 5
Vértice 35: Color 68
Vértice 36: Color 5
Vértice 37: Color 78
Vértice 38: Color 29
Vértice 39: Color 76
Vértice 40: Color 77
Vértice 41: Color 72
Vértice 42: Color 19
Vértice 43: Color 19
Vértice 44: Color 63
Vértice 45: Color 77
Vértice 46: Color 7
Vértice 47: Color 19
Vértice 48: Color 19
Vértice 49: Color 19
Vértice 50: Color 29
Vértice 51: Color 39
Vértice 52: Color 76
Vértice 53: Color 77
Vértice 54: Color 44
Vértice 55: Color 29
Vértice 56: Color 78
Vértice 57: Color 43
Vértice 58: Color 31
Vértice 59: Color 5
Vértice 60: Color 19
Vértice 61: Color 19
Vértice 62: Color 5
Vértice 63: Color 7
Vértice 64: Color 44
Vértice 65: Color 47
Vértice 66: Color 9
Vértice 67: Color 44
Vértice 68: Color 77
Vértice 69: Color 44
Vértice 70: Color 43
Vértice 71: Color 9
Vértice 72: Color 44
Vértice 73: Color 19
```

```
Vértice 74: Color 5
Vértice 75: Color 19
Vértice 76: Color 39
Vértice 77: Color 19
Vértice 78: Color 9
Vértice 79: Color 19
Vértice 80: Color 72
Total de colores utilizados: 22

Arreglo de valores discretos: [78, 40, 76, 76, 5, 7, 78, 31, 7, 7,
40, 19, 5, 17, 28, 5, 68, 18, 17, 19, 80, 29, 18, 80, 64, 29, 5, 47,
44, 28, 63, 64, 29, 5, 68, 5, 78, 29, 76, 77, 72, 19, 19, 63, 77, 7,
19, 19, 19, 29, 39, 76, 77, 44, 29, 78, 43, 31, 5, 19, 19, 5, 7, 44,
47, 9, 44, 77, 44, 43, 9, 44, 19, 5, 19, 39, 19, 9, 19, 72]
```

Ahora realizamos la búsqueda local iterada con criterio de termino de 1000 iteraciones, de igual forma 1000 iteraciones para la búsqueda escalada.

```
Primer solución aleatoria: [79, 59, 30, 45, 60, 52, 23, 23, 20, 50,
24, 63, 4, 19, 57, 42, 65, 19, 40, 6, 45, 8, 24, 62, 20, 31, 75, 40,
3, 41, 50, 73, 54, 4, 68, 73, 69, 21, 56, 55, 24, 53, 5, 32, 45, 14,
35, 75, 73, 15, 79, 42, 35, 66, 15, 5, 56, 65, 11, 27, 56, 9, 12, 53,
42, 44, 75, 58, 26, 58, 15, 55, 71, 37, 71, 29, 78, 55, 12, 25]

Evaluación de la primer solución aleatoria: 50

Solución más optimizada encontrada después de 1000 iteraciones
y 1000 iteraciones sin mejora.
Vértice 1: Color 74
Vértice 2: Color 74
Vértice 3: Color 22
Vértice 4: Color 11
Vértice 5: Color 74
Vértice 6: Color 13
Vértice 7: Color 51
Vértice 8: Color 51
Vértice 9: Color 41
Vértice 10: Color 36
Vértice 11: Color 74
Vértice 12: Color 57
Vértice 13: Color 74
Vértice 14: Color 75
Vértice 15: Color 36
Vértice 16: Color 74
Vértice 17: Color 75
```

```
Vértice 18: Color 74
Vértice 19: Color 22
Vértice 20: Color 22
Vértice 21: Color 75
Vértice 22: Color 13
Vértice 23: Color 74
Vértice 24: Color 74
Vértice 25: Color 75
Vértice 26: Color 74
Vértice 27: Color 36
Vértice 28: Color 46
Vértice 29: Color 13
Vértice 30: Color 74
Vértice 31: Color 74
Vértice 32: Color 74
Vértice 33: Color 11
Vértice 34: Color 75
Vértice 35: Color 51
Vértice 36: Color 74
Vértice 37: Color 11
Vértice 38: Color 11
Vértice 39: Color 51
Vértice 40: Color 74
Vértice 41: Color 74
Vértice 42: Color 75
Vértice 43: Color 75
Vértice 44: Color 13
Vértice 45: Color 36
Vértice 46: Color 75
Vértice 47: Color 74
Vértice 48: Color 22
Vértice 49: Color 51
Vértice 50: Color 36
Vértice 51: Color 74
Vértice 52: Color 74
Vértice 53: Color 22
Vértice 54: Color 46
Vértice 55: Color 36
Vértice 56: Color 74
Vértice 57: Color 41
Vértice 58: Color 75
Vértice 59: Color 75
Vértice 60: Color 41
Vértice 61: Color 74
Vértice 62: Color 41
```

```
Vértice 63: Color 51
Vértice 64: Color 74
Vértice 65: Color 74
Vértice 66: Color 13
Vértice 67: Color 57
Vértice 68: Color 46
Vértice 69: Color 57
Vértice 70: Color 51
Vértice 71: Color 36
Vértice 72: Color 57
Vértice 73: Color 13
Vértice 74: Color 74
Vértice 75: Color 13
Vértice 76: Color 11
Vértice 77: Color 51
Vértice 78: Color 74
Vértice 79: Color 74
Vértice 80: Color 74
Total de colores utilizados: 10

Arreglo de valores discretos: [74, 74, 22, 11, 74, 13, 51, 51, 41, 36,
74, 57, 74, 75, 36, 74, 75, 74, 22, 22, 75, 13, 74, 74, 75, 74, 36, 46,
13, 74, 74, 74, 11, 75, 51, 74, 11, 11, 51, 74, 74, 75, 75, 13, 36, 75,
74, 22, 51, 36, 74, 74, 22, 46, 36, 74, 41, 75, 75, 41, 74, 41, 51, 74,
74, 13, 57, 46, 57, 51, 36, 57, 13, 74, 13, 11, 51, 74, 74, 74]
```

Ambas búsquedas comienzan con una solución aleatoria, por lo que los primeros resultados coloreaban la gráfica con aproximadamente cincuenta colores. Como la búsqueda local iterada hace uso de la búsqueda por escalada, el resultado será mucho mejor utilizando ésta búsqueda, y no solamente utilizado búsqueda por escalada. Notamos que con la anterior búsqueda el resultado se queda a 10 colores del color mínimo, es un resultado aceptable para la dimensión de la gráfica, mientras que con la búsqueda local iterada logramos llegar al mínimo global. La gráfica utilizada se encuentra dentro de la carpeta $graficas$ de nuestro archivo de entrega.

Conclusiones

La búsqueda local iterada implementada con la búsqueda por escalada como nuestra búsqueda local, presenta una gran mejora respecto a la búsqueda por escalada normal, puesto que son múltiples búsquedas por escalada por cada iteración, es decir, cada solución perturbada se sigue optimizado para poder escapar del mínimo local hasta llegar a nuestro criterio de termino, que son el número de iteraciones ingresado.

References

[Adm23] Administrador. Código GRAY - Electrónica UnicRom, 11 2023.