Proyecto Final

Algoritmo Genético Aplicado a la Programación de Horario de Cursos usando Enfoque de Coloreación de Gráficas

Computación Concurrente

I.I.M.A.S.

U.N.A.M

Barajas Cervantes Alfonso
Cabello Figueroa Israel
Cerritos Lira Carlos
Franco López Benito Vicente

Dr. Óscar Alejandro Esquivel Flores

9 de febrero del 2021

Resumen

En este documento presentamos un diseño, análisis e implementación de un algoritmo genético para el problema de coloreo de gráficas. El algoritmo genético desarrolado aquí utiliza más de una selección de padre y métodos de mutación dependiendo en el estado de fitness de la mejor solución. Esto resulta en un cambio de la solución al óptimo global más rápidamente que usando una selección de un padre o el método de mutación. Conseguimos colorear de un noventa a un cien por ciento nuestras gráficas de muestra con el algoritmo con la cantidad de colores requerida para cada problema. Así mismo se mejora este programa usando una implementación concurrente de este algoritmo. Presentamos la implementación y comparación del algoritmo paralelo con una versión secuencial. Obtuvimos un rendimiento de tres a seis veces superior para nuestros casos de aplicación con la versión paralela.

1. Algoritmos evolutivos , computación evolutiva y más

Marco Histórico

En 1859, Darwin publica su libro El origen de las especies que levantó agrias polémica en el mundo científico por las revolucionarias teorías que sostenían que las especies evolucionan acorde al medio, para adaptarse a éste. De esta manera el universo pasaba de ser una creación de Dios estática y perfecta y se planteaba como un conjunto de individuos en constante competición y evolución para poder perpetuar su especie en el tiempo. Las especies se crean, evolucionan y desaparecen si no se adaptan de forma que solo los mejores, los más aptos, los que mejor se adapten al medio sobreviven para perpetuar sus aptitudes. De acuerdo con esta visión de la evolución, la computación ve en dicho marco un claro proceso de optimización: se toman los individuos mejores adaptados –mejores soluciones temporales –, se cruzan –mezclan–, generando nuevos individuos –nuevas soluciones– que contendrán parte del código genético –información– de sus antecesores, y el promedio de adaptación de toda la población se mejora.

La computación evolutiva se ha convertido en un concepto general adaptable para resolución de problemas, en especial problemas difíciles de optimización, ella encuentra fuerza combinada con otras técnicas como lo es la computación concurrente.[**Coley (1999)**]

Explicación sobre los algortimos evolutivos

La rama de la computación evolutiva es en si una rama que envuelve una gran comunidad de gente, ideas y de aplicaciones. Aunque sus raices genealogicas podemos tenerlos desde 1930, surgió como la emergencia de una relativa tecnologia computacional digital no costosa en la década de los 60, que sirvió como un importante catalizador para la rama. La viabilidad de usar esta tecnología fue de gran importancia para realizar simulaciones como herramiento para analizar sistemas mucho más complejos que aquellos que se podrían analizar matemáticamente.[**Coley (1999)**]

Los algoritmos genéticos son herramientas que podemos usar aplicando aprendizaje máquina que algunas veces nos permite encontrar soluciones para problemas que podrían tener un número muy grande de potenciales soluciones. Cuando resolvemos un problema con un algoritmo genético en lugar de emplear una solución específica, necesitamos dar carácterísitcas y reglas a la solución para que estás sean aceptadas. Cuando estamos viendo por ejemplo si pensamos en la tarea de llenar un camión de carga, debemos establecer principios y reglas que restringan los posibles ordenes en que acomodemos las cosas, por ejemplo una primera regla podría ser meter las cosas más pesadas primero, las cosas frágiles al final, para colocarlas en sitios donde no se golpeen. Estás reglas restrigen el espacio de posibles soluciones.

Ahora pensemos que estamos resolviendo un problema en particular de adivinar un numero, entre 1 y 1000, y tenemos solo diez opciones para adivinar. Si la única retroalimentación que tienes del número, es si escogiste correctamente, o incorrectamente, entonces tu mejor oportunidad es de 1 en 100 de adivinar el número. Con los algoritmos genéticos recibimos información adicional, dan una retroalimentación acerca de que tan cerca o lejos están de la solución. Si en lugar de correcto o incorrecto recibimos este indicador, la probabilidad de encontrarlo es total, porque con búsqueda binaria podemos encontrar cualquier número entre 1 y 1000 en diez pasos.

1.1. Clasificación de los algoritmos evolutivos

1. Estrategias Evolutivas: Fue diseñada inicialmente con la meta de resolver problemas de optimización discretos y continuos, principalmente experimentales y considerados difíciles. Trabaja con vectores de números reales Dcon desviaciones estándarD que codifican las posibles soluciones de problemas numéricos. Utiliza recombinación o cruce (crossover aritmético), mutación y la operación de selección, ya sea determinística o probabilística, elimina las peores soluciones de la población y no genera copia de aquellos individuos con una aptitud por debajo de la aptitud promedio.

2. Programación Evolutiva: Inicialmente fue diseñada como un intento de crear inteligencia artificial. La representación del problema se realiza mediante números reales (cualquier estructura de datos), y emplea los mecanismos de mutación y selección. El procedimiento es muy similar a las estrategias evolutivas con la diferencia de que no emplea la recombinación, de tal forma que son denominadas en conjunto algoritmos evolutivos como una manera de diferenciarlas de los algoritmos genéticos.

3. Algoritmos Genéticos: Modelan el proceso de evolución como una sucesión de frecuentes cambios en los genes, con soluciones análogas a cromosomas. Trabajan con una población de cadenas binarias para la representación del problema, y el espacio de soluciones posibles es explorado aplicando transformaciones a éstas soluciones candidatas tal y como se observa en los organismos vivientes: cruce, inversión y mutación. Como método de selección emplean en mecanismo de la ruleta (a veces con elitismo). Constituyen el paradigma más completo de la computación evolutiva ya que resumen de modo natural todas las ideas fundamentales de dicho enfoque. Son muy flexibles ya que pueden adoptar con facilidad nuevas ideas, generales o específicas, que surjan dentro del campo de la computación evolutiva. Además, se pueden hibridar fácilmente con otros paradigmas y enfoques, aunque no tengan ninguna relación con la computación evolutiva. Se trata del paradigma con mayor base teórica.[**Coley (1999)**]

1.2. Procesos Evolutivos Básicos

Un lugar bueno para empezar es preguntarse los componentes básicos de un sistema evolutivo. La primer nota que debemos tomar en cuenta es que existen al menos dos posibles interpretaciones de un *Sistema Evolutivo*. Es frecuentemente usado para describir un sistema que cambia con incrementos a lo largo del tiempo. La segunda interpretación es aquella que tiene un sentido biológico, es decir, que es un sistema evolutivo Darwiniano.[**Goldberg(1989)**]

Para ello, precisamos saber lo que entendemos por un *Sistema*. Una manera es identificando lo que precisamente constituye al sistema, es decir todas sus partes. A continuación se presenta el conseso de lo que se entiende por un sisteme evolucionario Darwiniano.

- Una o más poblaciones compiten por limitados recursos
- Noción de que las poblaciones cambien dinámicamente debido al nacimiento y la muerte de individuos.
- El concepto de la aptitud en donde refleja la habilidad de un individuo de sobrevivir y de reproducirse
- El concepto de herencia variacional: la descendencia se parece mucho a sus padres, pero son no es identicos.

1.3. Un primer ejemplo: Un simple Sistema Evolutivo

La primera cuestión a afrontar es cómo representar a los individuos (organismos) que componen una población en evolución. Una técnica bastante general es describir a un individuo como un vector de longitud de las características L que se eligen presumiblemente debido a su relevancia (potencial) para estimar la aptitud de un individuo. Entonces, por ejemplo, los individuos podrían caracterizarse por:

$$< \text{color cabello, color ojos, color piel, altura, peso} >$$

Podríamos pensar libremente en este vector como que especifica la composición genética de un individuo, es decir, su genotipo especificado como un cromosoma con cinco genes cuyos valores dan como resultado

un individuo con un conjunto particular de rasgos. Alternativamente, podríamos considerar tales vectores como descripciones de los rasgos físicos observables de los individuos, es decir, su fenotipo. En cualquier caso, por especificando adicionalmente el rango de valores (alelos) que tales características pueden asumir, se define un espacio de cinco dimensiones de todos los posibles genotipos (o fenotipos) que los individuos pueden tener en este mundo artificial.[1]

```
EV:
    Generar una poblacion inicial de M individuos
    DO FOREVER:
        Seleccionar un miembro del la actual poblacion para que sea padre

        Usa el seleccionado padre para que produzca una descendencia que
        sea similar pero generalmente no una copia precisa del padre

        Selecciona a un miembro de la poblacion para que muera
        END DO
```

1.4. Algoritmos Evolutivos,investigación

1.4.1. Búsqueda Aleatoria

El método de búsqueda aleatoria es el primer método que basó su estrategia de optimización en un proceso totalmente estiocástico. Bajo este método, sólo una solución canidato x^k es mantenida durante el proceso de evolución. En cada iteración, la solución cadidato x^k es modificada añadiendo un vector aleatorio Δx. De esta manera, la nueva solución candidato es modelada bajo la siguiente expresión:

$$x^{k+1} = x^k + \Delta x$$

Considerando que la solución candidato x^k tiene d dimensiones $\left(x_1^k, x_2^k, \cdots, x_d^k\right)$, cada coordenada es modificada $(\Delta x = \{\Delta x_1, \Delta x_2, \cdots, \Delta x_d\})$ mediante una perturbación aleatoria Δx, $(i \in [1, 2, \ldots, d])$ mpde;ada por una distribución de probabilidad gaussiana $p(\Delta x_i) = N(\mu_i, \sigma_i)$, se considera que $\mu_i = 0$ dado que el valor de Δx_i añade una modificación alrededor de x_i^k.
Una vez calculado $x^k + 1$, se prueba si la nueva posición mejora la calidad de la solución candidato anterior x^k. De esta manera, si la calidad de $x^k + 1$ es mejor que x^k, el valor de $x^k + 1$ es aceptado como la nueva solución candidato, en caso contrario permanece x^k sin cambio. Esta prueba puede definirse para un caso de minimización como:

$$x^{k+1} = \begin{cases} x^{k+1} & \text{si } f(x^{k+1}) < f(x^k) \\ x^k & \text{si } f(x^{k+1}) \geq f(x^k) \end{cases}$$

A este criterio de sustitución, de aceptar solamente los cambios que mejoren la calidad de la solución es conocido como "*greedy*". En búsqueda aleatoria, la perturbación Δx impuesta a x^k podría hacer que el nuevo valor x^{k+1} quede fuera del espacio de búsqueda denido X. Fuera del espacio de búsqueda X no existe denición de la función objetivo $f(x)$. Para evitar este problema, el algoritmo debe proteger la evolución de la solución candidato x^k, de tal forma que si x^{k+1} queda fuera del espacio de búsqueda X, se le debe de asignar una muy mala calidad (representado por u valor muy grande). Esto es $f(x^{k+1}) = \infty$ para el caso de la minimización o $f(x^{k+1}) = -\infty$ para el caso de la maximización. [**Cuevas (2016)**]

1.4.2. Temple Simulado

El temple simulado o en inglés 'Simulated annealing' es una técnica de optimización que emula el proceso de templado en materiales metálicos. La idea del templado es enfriar el material controladamente de tal manera que las estructuras cristalinas puedan orientarse y evitar los defectos en las estructuras metálicas.
El uso del templado como inspiración para la formulación de algoritmos de optimización fue por primera vez propuesto por Kirkpatrick, Gelatt y Vecchi en 1983. Desde entonces se han sugerido varios estudios y aplicaciones para analizar los alcances del método.A diferencia de algoritmos basados en gradiente, el método del temple simulado presenta una gran habilidad para evitar quedar atrapados en los mínimos locales.

[1] El ejemplo fue obtenido de [**Cuevas (2016)**]

Algoritmo 1.2 Método de búsqueda aleatoria para resolver 1.14	
1.	k ← 0
2.	$x_1^k \leftarrow$ Aleatorio $[-3,3]$, $x_2^k \leftarrow$ Aleatorio $[-3,3]$
3.	**while** (k < *Niter*) {
4.	$\Delta x_2 = N(0,1)$
5.	$x_1^{k+1} = x_1^k + \Delta x_1$, $x_1^{k+1} = x_1^k + \Delta x_1$
6.	$\mathbf{x}^k = (x_1^k, x_2^k)$, $\mathbf{x}^{k+1} = (x_1^{k+1}, x_2^{k+1})$
7.	**If**($\mathbf{x}^{k+1}$ **x**) { $f(\mathbf{x}^{k+1}) = -\infty$ }
8.	**If** $\left(f(\mathbf{x}^{k+1}) < \left(f(\mathbf{x}^k)\right)\right)$ { $\mathbf{x}^{k+1} = \mathbf{x}^k$
9.	k ← k + 1 }

Figura 1: Algoritmo de Búsqueda Aleatoria

En el templado simulado, el valor de la función objetivo que se intenta optimizar es análogo a la energía de un sistema termodinámico. En altas temperaturas el algoritmo permite la exploración de puntos muy distantes entre sí en el espacio de búsqueda, además de que la probabilidad de aceptación de soluciones que no mejoran su estado anterior es muy grande. Po el contrario, a bajas temperaturas, el algoritmo permite únicamente la generación de puntos muy cercanos entre sí, además de que la probabildiad de aceptación se reduce, por lo que ahora sólo las nuevas soluciones que mejoren su estado anterior, serán consideradas.
El *algoritmo de temple simulado* matiene urante su funcionamiento una sola solución candidato (x^k). Dicha solución es modificada en cada iteración utilizando un procedimiento similar al de método de búsqueda aleatoria, donde cada punto es modificado mediante la generación de un vector aleatorio Δx. Sin embargo, este algoritmo no solo acepta los cambios que mejoren su función objetivo, sino que incorpora un mecanismo probabilístico, que le permite aceptar incluso soluciones que le empeoren. Esto con el fin de salir de los mínimos locales. [**Cuevas(2016)**] Bajo estas circunstancias, una nueva solución será aceptada x^{k+1} considerando dos diferentes alternativas

- Si su calidad es superior a la de su antecesor x^k. Esto es si $f(x^{k+1}) > f(x^k)$
- Bajo una probabilidad de aceptación $p_a = e^{\frac{\Delta f}{T}}$, donde T representa la temperatura que controla el proceso de templado, mientras que f es la diferencia de energía entre el puntos $\Delta f = f(x^{k+1}) - f(x^k)$

A continuación mostramos el algoritmo de temple simulado.

1.4.3. Genéticos

Un algoritmo genético es un método de búsqueda que imita la teoría de la evolución biológica de Darwin para la resolución de problemas. Para ello, se parte de una población inicial de la cual se seleccionan los individuos más capacitados para luego reproducirlos y mutarlos para finalmente obtener la siguiente generación de individuos que estarán más adaptados que la anterior generación. [**D.Gutierrez**, **A. Tapia**, **2020**]

Algoritmo 1.3 Método del temple simulado	
1.	Configura T_{ini}, T_{fin}, $\beta\sigma$ y $Niter$.
2.	$k \leftarrow 0, T = T_{ini}$
3.	$x^k \leftarrow$ Aleatorio $[x]$
4.	**while** $((T > T_{fin})$ **and** $(k < Niter))$ {
5.	**while** $\left(x^{k+1} \notin X\right)$ {
6.	$\Delta x \leftarrow N\left(0, \sigma \cdot \mathrm{T}\right)$
7.	$x^{k+1} \leftarrow x^k + \Delta x\}$
8.	If $\left(x^{k+1}$ es peor que $x^k\right)$
9.	$\{\Delta f \leftarrow f\left(x^{k+1}\right) - f\left(x^k\right)$
10.	Pa $\leftarrow$ e- $\frac{\Delta f}{T}$
11.	**If** $(Pa <$ rand $)\{x^{k+1} = x^k\}\}$
12.	$k \leftarrow k + 1$
13.	$T \leftarrow \beta \cdot T\}$

Figura 2: Algoritmo de temple simulado

1.4.4. Implementación

Una premisa es conseguir que el tamaño de la población sea lo suficientemente grande para garantizar la diversidad de soluciones. Los pasos básicos de un algoritmo genético son:

- Evaluar la puntuación de cada uno de los cromosomas generados.
- Permitir la reproducción de los cromosomas siendo los más aptos los que tengan más probabilidad de reproducirse.
- Con cierta probabilidad de mutación, mutar un gen del nuevo individuo generado.
- Organizar la nueva población

Se puede fijar un número máximo de iteraciones antes de finalizar el algoritmo genético o detenerlo cuando no se produzcan más cambios en la población.

Parametros de los algoritmos geneticos.

- Tamaño de la población.
- Probabilidad de cruce.
- Probabilidad de mutación.

Al igual que en otros algoritmos metaheurísticos, los AG presentan algunas ventajas respecto a los métodos clásicos de optimización; por ejemplo, pueden trabajar con poca o ninguna información acerca de la función objetivo, y por ser poblacionales tienen mayores probabilidades de escapar de óp-timos locales, en comparación con los métodos determinísticos. Sin embargo, entre sus desventajas está el hecho de que no garantizan una convergencia al óptimo real, y en muchos casos son más tardados que las técnicas deterministas

Un ejemplo Veamos como aplicar un algoritmo genetico para minimizar la siguiente función [2]. la cual es dificil desde el punto de vista computacional, pues contiene bastantes optimos locales

$$f(x_1, x_2) = a + exp(1) - a * exp(-b\sqrt{\frac{1}{d}(x_1^2 + x_2^2)}) - exp(\frac{1}{d}(\cos c_1 x_1 + \cos c_2 x_2))$$
$$x_1, x_2 \in [-15, 30]$$

```
function [temp, ind] = FUNCION(x, d, N) y=[];%
Ackley
for j=1:N
   a = 20; b = 0.2; c = 2*pi;
   s1 = 0; s2 = 0;
   for i=1:d;
      s1 = s1+x(j,i)^2;
      s2 = s2+cos(c*x(j,i));
end
y(j,1) = -a*exp(-b*sqrt(1/d*s1))-exp(1/d*s2)+a+exp(1);
end
[temp,ind]=sort(y);
```

Básicamente un AG es un programa que genera una población inicial aleatoria de padres, y durante cada generación selecciona pares de padres, considerando su valor de $f(x_i,)$, para realizar intercambios de material genético, o cruza, y generar pares de hijos; tales hijos serán mutados con una cierta probabilidad, y nalmente competirán por sobrevivir a la siguiente generación con los padres, proceso que se conoce como elitismo. Aunque dicho comportamiento produce la convergencia del algoritmo, también es cierto que provoca que la población sea 'arrastrada' por el mejor individuo de la población, lo que conduce en ocasiones un estancamiento de la población

La desventaja de esto es que podria llegarse encontrar un optimo local en ves de uno global

1.4.5. Estrategias Evolutivas

Las EE son algoritmos inspirados en la evolución y al igual que otros basados en tal metáfora, sus individuos realizarán una "evolución", o mejora, con respecto a alguna función objetivo y mediante operadores que imitan dicho proceso.

Podriamos por ejemplo considerar minimizar f tal que:

$$f(x_1, x_2) = -sen(x1)sen^{2m}(\frac{x_1^2}{\pi}) + sen(x_2)sen^{2m}(\frac{2x_2^2}{\pi} x_1, x_2 \in [0, \pi] m = 10$$

Para ello podemos implementar el siguiente codigo

```
function [f, i1] = FUNCION(x, d, mu)
y=[];
for i1=1:mu
m = 10;   s = 0;
for i2 = 1:d
s = s + sin(x(i1,i2))*(sin(i2*x(i1,i2)^2/pi))^(2*m);
end
```

[2] El algoritmo y ejemplo fueron obtenidos de [**Cuevas(2016)**]

```
y(i1,1) = -s;
end
[f, i1]=sort(y);
```

Esta función en particular tambien presenta minimos locales por lo que es dificil estudiarla al menos con las estrategias que conocemos hasta el momento Esencialmente, los operadores que se utilizan en las EE son muy parecidos a los de un AG, cuando menos nominalmente; sin embargo, la diferencia más notoria es que en las EE el operador de mutación no es un solo valor, sino una matriz de valores de mutación. Esta es una de sus características más interesantes, e incluso se considera como la más importante. La principal diferencia en la inicialización en las EE con respecto a otros algoritmos evolutivos radica en que se crean las matrices de covarianza y rotaciones (σ, α)

Se muestra a continuación las principales estrategias a seguir para generar un algoritmo evolutivo.

Algoritmo 3.1 Estrategias Evolutivas, versión $(\mu / \rho + \lambda)$ -EE	
1.	Configurar parámetros del algoritmo
2.	Inicializar y evaluar población inicial
3.	Mientras (no se cumpla criterio)
4.	**for** 1 : λ
5.	Seleccionar ρ padres, parámetros de estrategia
6.	Seleccionar ρ padres, variables de decisión
7.	Recombinar parámetros de estrategia
8.	Recombinar variables de decisión
9.	Mutar parámetros de estrategia
10.	Mutar variables de decisión
11.	Seleccionar de la población de hijos y de la población de padres anterior a los nuevos padres
12.	Mostrar resultado

Figura 3: Muestra de un algoritmo evolutivo

[3]

1.4.6. Evolución Diferencial

El algoritmo de Optimización Evolución Diferencial (Differential Evolution, por sus siglas en inglés), es un algoritmo poblacional de búsqueda directa y simple, el cual es capaz de optimizar hasta alcanzar el óptimo global en funciones multimodales, no diferenciales y no lineales. Fue propuesto por *Kenneth Price y Rainer Storn* en 1995. Desde entonces, el algoritmo DE ha probado sus capacidades en concursos Internacionales de la IEEE en Optimización Evolutiva, asi como con grandes aplicaciones en el mundo real.

El DE se basa en perturbar a los miembros de la población generada con diferencias escaladas de distintos miembros de una misma población. Lo hacen característico de los demás algoritmos evolutivos dado que esta inspirado en un fénomeno biológico, social, físico o químico. El algoritmo DE fue diseñado para que se cumpliera con (i) capacidad de lidiar con funcio-nes objetivo no diferenciables, no lineales y multimodales, (ii) fácil implementación, (iii) pocos parámetros de ajuste, (iv) propiedades de convergencia consistentes en pruebas consecutivas in-dependientes y (v) capacidad de paralelizarse para lidiar con funciones de alto costo computa-cional. A continuación se presenta el algoritmo o un diagrama de flujo génerico de las operaciones llevadas acabo por el algoritmo evolución diferencial. De acuerdo con estudios reportados recientemente, el algoritmo DE ha mostrado un mejor desempeño que muchos algoritmos evolutivos en términos de convergencia, velocidad y robustez sobre distintas funciones de prueba.

[3] [**Cuevas(2016)**] pp 67

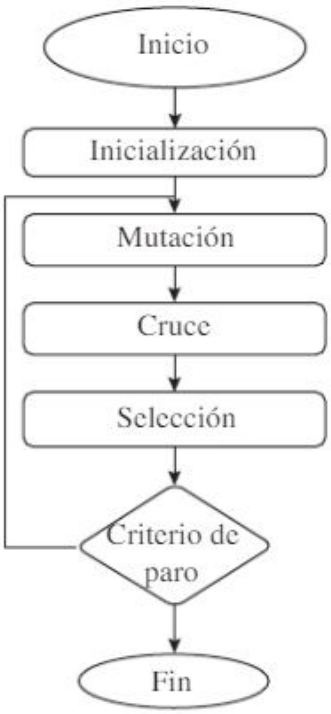


Figura 4: Diagrama de flujo de la búsqueda armónica

1.4.7. Búsqueda Armónica

En este algoritmo se representa a una armonía como en vector n-dimensional de números reales.[**Coley (1999)**]

- Un conjunto inicial de vectores de armonía son generados aleatoriamente y almacenados dentro de la memoría de armonías (HM).
- Una nueva armonía candidada es calculada a partir de los elementos contenidos en la HM, para esto existe un parámetro conocido como consideración de memoria.
- Se actualiza la memoria de armonías, por medio de la comparación de la nueva armonía candidata y el peor de los vectores de armonía.

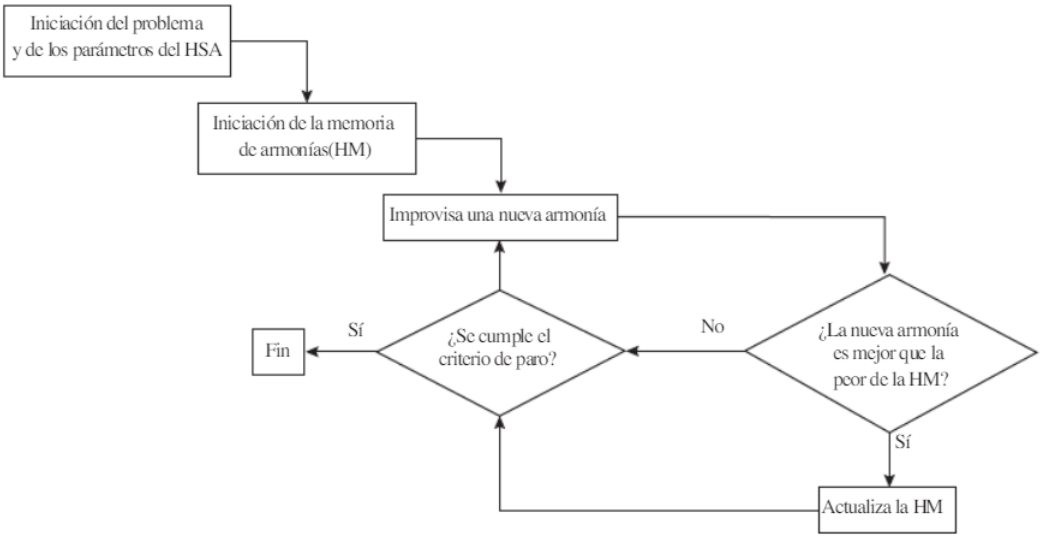


Figura 6.2. Diagrama de flujo del algoritmo HSA.

Un popularidad del algoritmo HSA se debe a un conjunto de características que lo distinguen del resto de los algoritmos metaheurísticos. Algunas de éstas son:
1. Para generar nuevas soluciones se consideran todas las soluciones existentes, no sólo dos de ellas como los padres en los algoritmos genéticos.
2. Se considera de forma independiente cada variable de decisión en un vector de soluciones.
3. Los valores de las variables de decisión son continuos, por lo que no se pierde precisión.

1.4.8. Sistemas Inmunológicos Artificiales

Para definir lo que es un sistema inmune artificial recurriremos a una de las definiciones más completa propuesta por Leandro Nunes de Castro y Jonathan Timmis.
Los sistemas inmunes artificiales son sistemas adaptativos, inspirados por la teoría inmunológica, funciones, principios y modelos inmunológicos observados, los cuales son aplicados a la solución de problemas.
Aunque la cantidad de modelos y aplicaciones del sistema inmune va en ascenso, no existe un esquema general de cuáles son los elementos esenciales que un sistema inmune artificial debe poseer.
Nunes De Castro y Timmis sugieren en su libro utilizar un esquema general de sistemas computacionales con inspiración biológica, tal como los algoritmos evolutivos o las redes neuronales. Este esquema consta de tres partes principales que deben denirse, y que a continuación se muestran:

- Representación de los componentes del sistema.
- Un conjunto de mecanismos para evaluar las interacciones de los individuos con el ambiente y entre ellos.
- Un proceso de adaptación que gobierne las dinámicas del sistema, es decir, el algoritmo en sí.

Representación de los componentes del sistema.
Un conjunto de mecanismos para evaluar las interacciones de los individuos con el ambiente y entre ellos.
Un proceso de adaptación que gobierne las dinámicas del sistema, es decir, el algoritmo en sí.

A diferencia de otras técnicas bio-inspiradas (por ejemplo los algoritmos genéticos), el sistema inmune artificial no tiene un algoritmo general único.[**Cuevas(2016)**]

1.4.9. Optimización basada en electromagnetismo

El algoritmo EMO es un método basado en población propuesto por Birbil y Fang para resolver modelos de optimización continuos usando variables acotadas. Es decir, problemas de optimización. El algoritmo EMO replica el problema de las cargas, donde cada carga representa una solución a un campo dado, lo cual representa que cada cantidad de carga es una parte de la solución y refleja su cálidad. Este algoritmo, permite encontrar solución a problemas de minimización del tipo $f(X)X\epsilon[l,u]$ a través de los siguientes cuatro pasos.

- Inicialización: un conjunto de m partículas son tomadas aleatoriamente considerando un espacio R definido por el límite superior u y el límite inferior l
- Búsqueda local: se realiza la búsqueda de un valor mínimo en la vecindad de un punto X_p, donde $p\epsilon[m1, m2]$ y m es el número total de individuos de la población.
- Cálculo del vector de fuerza total: con base en el valor de la función objetivo se calculan las cargas y fuerzas para cada elemento de población de partículas. : cada partícula de población es desplazada de acuerdo a la fuerza total calculada con base en el valor de la función objetivo.[4]

```
%Inicializacion, Algoritmo EMO Standar
%Prueba con función de Rosenbrock
function [x.fx] = inicializa (m,n,u,l)
%Se genera las partículas de forma aleatoria en el espacio de búsqueda
for i = 1:n
%Se obtiene el número aleatorio
lambda= rand(l,m);
x(i,:) = l(i) + lambda * (u(i) -l(i));
end
%Se evalúa cada partícula en la función objetivo (Rosenbrock)
for j =1:m
fx(j) = rosen(x(:,j));
end
```

El código anterior es una implementación del algoritmo EMO para la función de Rosenbrock.

1.4.10. Colonia Artificial en Abejas, Hormigas

El problema de optimización de colonia, es una tecnica probabilística para resolver problemas computacionales que pueden ser reducidos a encontrar un camino bueno en una gráfica.

La forma más fácil de entender como el algoritmo de optimización por colonia de hormigas funciona, es a través de un ejemplo. Consideremos el problema del agente viajero, consiste en, dado un conjunto de ciudades y la distancia entre estas, encontrar la ruta más corta que visite a todas y regrese al lugar de partida.

Las hormigas construyen la solución de la siguiente manera. Cada hormiga selecciona una ciudad aleatoria. Después, en cada paso se mueve a un nuevo vértice queno haya sido visitado de forma probabilística, esta decisión esta basada en información como la feromona. Cada vez que una hormiga finaliza su recorrido, la feromona de cada arista es actualizado de acuerdo a la calidad de la solución resultante.

[4]El ejemplo fue obtenido de [**Cuevas(2016)**]

2. Estudio y evaluación de un algoritmo evolutivo, de manera que se pueda implementar de manera concurrente.

2.1. Estudio y evaluación

El algoritmo genético como ya mencionamos en la sección anterior, es un algoritmo que restringe una búsqueda con los siguientes pasos.

- Generar población inicial.
- Seleccionar de esta población individuos más capacitados.
- Reproducir y mutar estos individuos.
- Obtiene una nueva generación de individuos mejor adaptados

El proceso se repite hasta llegar a una solución aceptable. El algoritmo genético podemos notar que se basa de un sistema de selección natural, en el cual tras cada iteración solo los individuos más aptos sobreviven, y que por tanto serán base para las generaciones posteriores. Esta capacidad les permite resolver muchos problemas que no son de un campo en específico. Solo debemos definir un problema y una función de evaluación de los individuos para poder resolver cualquier tipo de problema. Al tratarse de una técnica heurística el tiempo de ejecución puede volverse un obstáculo si el espacio de búsqueda de soluciones es muy amplio, además de que si el factor de mutación no es elegido con cuidado, podríamos hacer un algoritmo que no convergiese a el óptimo buscado. Por eso es necesario garantizar la diversidad de los individuos al realizar la técnica, y también es esta complicación por la cual es natural aplicar paralelización, precisamente en la subsección siguiente discutiremos esto con más profundidad, para dar la justificación a la elección de un algoritmo genético.[**Coley (1999)**]

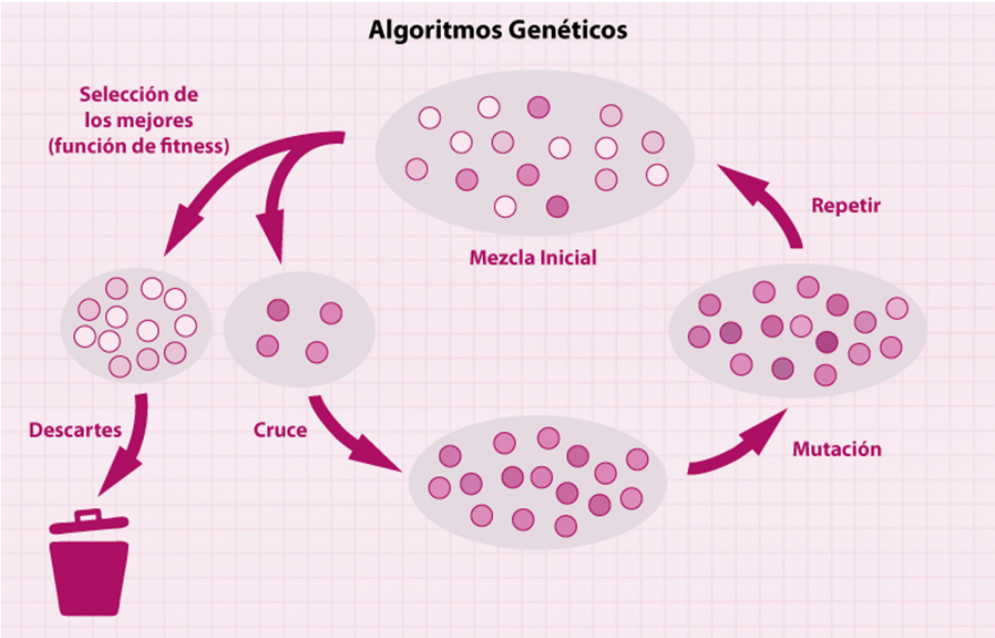


Figura 5: Esquema gráfico de ciclo del algoritmo genético.

2.2. Justificación

Decidimos elegir el algoritmo genético dado que la labor de selección de los individuos, es una tarea que se puede paralelizar y que además no afecta la solución final del algoritmo. La selección debe ser realizada

de manera global si queremos que el algoritmo produzca el mismo resultado que la secuencial original. Para paralelizar nuestro algoritmo, se puede optar por generar grupos pequeños de poblaciones que se comuniquen entre ellas en lugar de generar una población muy grande, de acuerdo a la bibliografía más apremia, realizar una paralelización en este punto puede ser incluso más favorable que el algoritmo secuencial.de hecho este proceso de paralelización se conoce como generar nichos. [5]

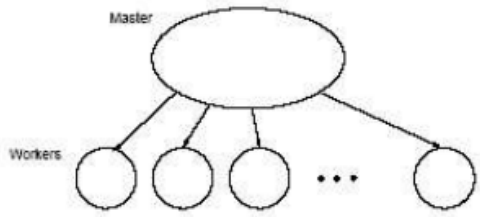


Figura 6: Diagrama proceso de paralelizacion tipo maestro-trabajador

Esta es una representación esquemática de un algoritmo genético paralelizado, con la aplicación del proceso paralelo en la evaluación de los individuos, ya que suele ser también la que toma más tiempo.

El intercambio de información entre los nodos realiza los siguientes pasos

- Nodo maestro envia subconjunto de individuos a cada nodo.
- Cada nodo retorna el valor de evaluación al nodo maestro.

De forma tradicional, el nodo maestro espera a recibir los valores de adecuación de todos los individuos para hacer la siguiente generación, pero también podemos hacer que los nodos más lentos no se comuniquen con el nodo maestro, aunque aquí perderíamos información del comportamiento original.

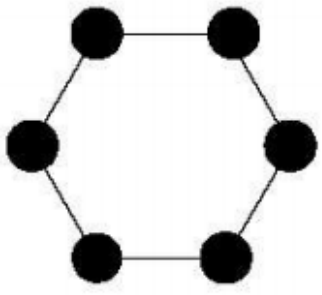


Figura 7: Diagrama de paralelización de poblaciones distribuidas.

Una segunda forma de generar paralelización, es generar una arquitectura de muchas poblaciones, este es el caso en donde cada sub población es repartida hacía un diferente procesador y es evaluada por separado. Se encontró que en está arquitectura, las poblaciones por separado convergen más rápido hacía valores óptimos, aunque en contraste, estos valores no son tan buenos como los del proceso secuencial.[**parallelgenetic2011**]

3. Problema de aplicación elegido en el punto anterior

En nuestro trabajo deseamos encontrar un coloración de una gráfica aleatoria dada [6] de la gráfica El problema de colorear una gráfica es un problema bastante conocido de tipo *NP-Completo*.El coloreado de

[5] Esto lo obtuvimos del artículo [**S. Poddar**, **2020**]

[6] Se dice que hay una coloración de la gráfica si para cualesquiera dos vértices adyacentes estos dos estan coloreados de diferente color

una gráfica incluye tanto a los vértices como a las aristas. Sin embargo, el término de coloreo de una gráfica se refiere a colorear los vértices en vez de al coloreado de aristas.

Dados un número n de vértices, que forman una gráfica conexa, el objetivo es colorear cada vértice de manera que dos vértices vecinos (i.e. adyacentes) sean coloreados de manera diferente. Claramente esta tarea se puede realizar si el número de colores que usamos es mayor al número de vértices, por esta razón definimos el número cromático χ de una gráfica como el número de colores más pequeños que se necesita para tener una coloración correcta.[**Prugel-Bennett**, **A. 2003. Genetic algorithm for graph coloring**]

Una herramienta que es útil en este contexto es el teorema de los cuatro colores, el cual nos dice que dada una gráfica plana [7] esta se puede colorear con únicamente 4 colores[**Prugel-Bennett**, **A. 2003. Genetic algorithm for g** Nuestro problema es un poco más general, pues para cualquier gráfica no necesariamente plana, necesitamos encontrar una coloración, para la cual podemos requerir de 1 hasta n colores.

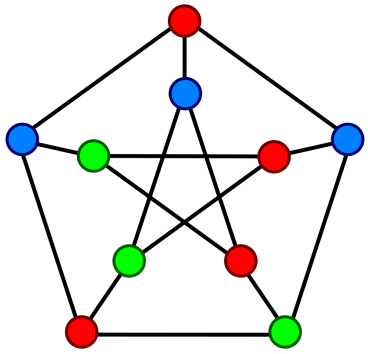


Figura 8: Gráfica coloreada con 3 colores.

3.1. Contribución del algoritmo evolutivo para resolver el problema

Observe que si k es el número de colores necesarios para hacer una coloración de nuestra gráfica, podemos encontrar dicha coloración a partir de un algoritmo genético. Un algoritmo genético es ideal para resolver este problema, pues podemos pensar a la persona más apta como aquella que tiene más aristas coloreados con vértices de diferente color, lo ideal seria que todos los aristas de nuestra gráfica cumplieran dicha condición, así podríamos comenzar con una población inicial, que seria un conjunto de coloraciones aleatorias, e ir seleccionando aquellas que sean más aptas, es decir, aquellas que tienen más aristas coloreados con vértices de distinto color y cruzarlas entre ellas para obtener una nueva población de manera que obtengamos cada vez mejores coloraciones.

[**M. Hindi and V.Yampolskiy**, **2011**]

Diremos que hemos encontrado una coloración valida cuando el número de aristas que tengan 2 vértices adyacentes coloreados de distinto color, sea igual al número de aristas de nuestra gráfica.

[7] Se dice que una gráfica es plana, si es posible dibujarla en el plano de manera que no haya dos aristas que se crucen entre ellos,el teorema de Euler, que nos dice que dada una gráfica, está es plana, si dadas sus caras, vértices y aristas, el número de vértices menos el número de aristas mas el numero de caras es exactamente igual a dos

4. Análisis diseño e implementación de la paralelización del algoritmo.

4.1. Diseño

4.1.1. Definiciones

En este caso se tienen las siguientes definiciones:

- **Person**: Una coloración arbitraria de la gráfica, esta se puede ver como una lista que nos indica el color de cada vértice $[1, 0, 2, 1, ..., 0]$.
- **Fitness**: Número de aristas que conectan a dos vértices de distinto color.
- **Crossover**: Dadas 2 coloraciones (padres) elegimos un número aleatorio entre 0 y n, y creamos una nueva coloración que tenga los primeros k colores iguales a los de la primera y el resto igual a los de la segunda.
- **Mutation**: Dada una coloración, tomar un vértice arbitrario y cambiar su color.

4.1.2. Algoritmo

En resumen se podría diseñar nuestro algoritmo de la siguiente manera:

1. Generar población aleatoria (primer generación). Nuestra población seria un conjunto de coloraciones.
2. Distribuir a la población en islas, cada isla evolucionará de manera independiente. Para cada isla:
3. Obtener fitness para cada individuo. Número de aristas que conectan 2 vértices de distinto color.
4. Seleccionar padres, favoreciendo aquellos que hayan obtenido un mayor fitness.
5. Hacer crossover y mutaciones para obtener nueva generación.
6. Repetir paso 2, según el número de generaciones que deseemos.
7. Obtener el individuo con mayor fitness para cada isla, el mejor de estos será la coloración que usaremos (the best of the best).

4.1.3. Parámetros

Del algoritmo se tienen los siguientes parámetros:

- **G**: Gráfica que se desea colorear.
- **population_size**: Número de personas en la población.
- **n_islands**: Número de islas en las que se distribuirá la población.
- **n_generations**: Número de generaciones que se usarán para evolucionar a la población.
- **percentage_to_keep**: Porcentaje de la población con mejor fitness que se copia de a la siguiente generación.
- **n_colors**: Número de colores con los que se planea colorear la gráfica.

4.1.4. Como medir performance

Para evaluar nuestras soluciones, contamos con gráficas previamente estudiadas para las cuales su número cromático χ es conocido (Figura 9). Obtenidas del *Center for Discrete Mathematics and Theorical Computer Science* (DIMACS) challenge [**Graph colors instance**], el cual es un conjunto comúnmente usado para medir el performance de este tipo de algoritmos . De esta forma, esperamos que fitness_percentage, definido como:

$$fitness\_percentage = \frac{\#AristasColoreadas}{\#Aristas}$$

sea 1 o muy cercano a 1, pues sabemos que existe una solución, y podemos entonces evaluar el tiempo de ejecución y velocidad de convergencia.

File	|V|	|E|	Expected $\chi(G)$	k_{min}	HPGAGCP Result	Time (s)
myciel3.col	11	20	4	4	4	0.003
myciel4.col	23	71	5	5	5	0.006
queen5_5.col	23	160	5	5	5	0.031
queen6_6.col	**25**	**290**	**7**	**7**	**8**	**6.100**
myciel5.col	36	236	6	6	6	0.014
queen7_7.col	**49**	**476**	**7**	**7**	**8**	**6.270**
queen8_8.col	**64**	**728**	**9**	**9**	**10**	**47.482**
huck.col	74	301	11	11	11	0.015
jean.col	80	254	10	10	10	0.015
david.col	87	406	11	11	11	0.019
games120.col	120	638	9	9	9	0.027
miles250.col	128	387	8	8	8	0.076
miles1000.col	128	3216	42	42	42	48.559
anna.col	138	493	11	11	11	0.058
fpsol2.i.1.col	496	11654	65	65	65	22.656
homer.col	561	1629	13	13	13	0.760

Figura 9: Datos de gráficos conocidos estudiados con HPGAGCP

Podemos notar en la Figura 9 [8] que estas gráficas previamente estudiada, cuentan con diferentes características, van de los 11 vértices a los 561, y van desde 20 a 3216 aristas , esto significa que tenemos una muestra muy amplia de casos de estudio que nos permitirán evaluar y validar nuestro algoritmo. Contamos también con la cantidad de colores necesarios para cada gráfica y el tiempo de ejecución con el algoritmo HPGAGCP. [9]

Para la simulación se utilizara python y la libreria networkx junto con matplotlib asi como multiprocessing lo que nos permitirá implementar nuestro algoritmo de manera paralela.

[8] Table 1: Results of running the proposed algorithm on 16 .col files from the DIMACS collection en [**M. Hindi and V.Yampolskiy**, **2011**]

[9] El algoritmo HPGAGCP viene descrito en la sección Proposed Approach en [**M. Hindi and V.Yampolskiy**, **2011**]

4.2. Implementación del algoritmo evolutivo de manera concurrente

El algoritmo principal quedaría de la siguiente manera:

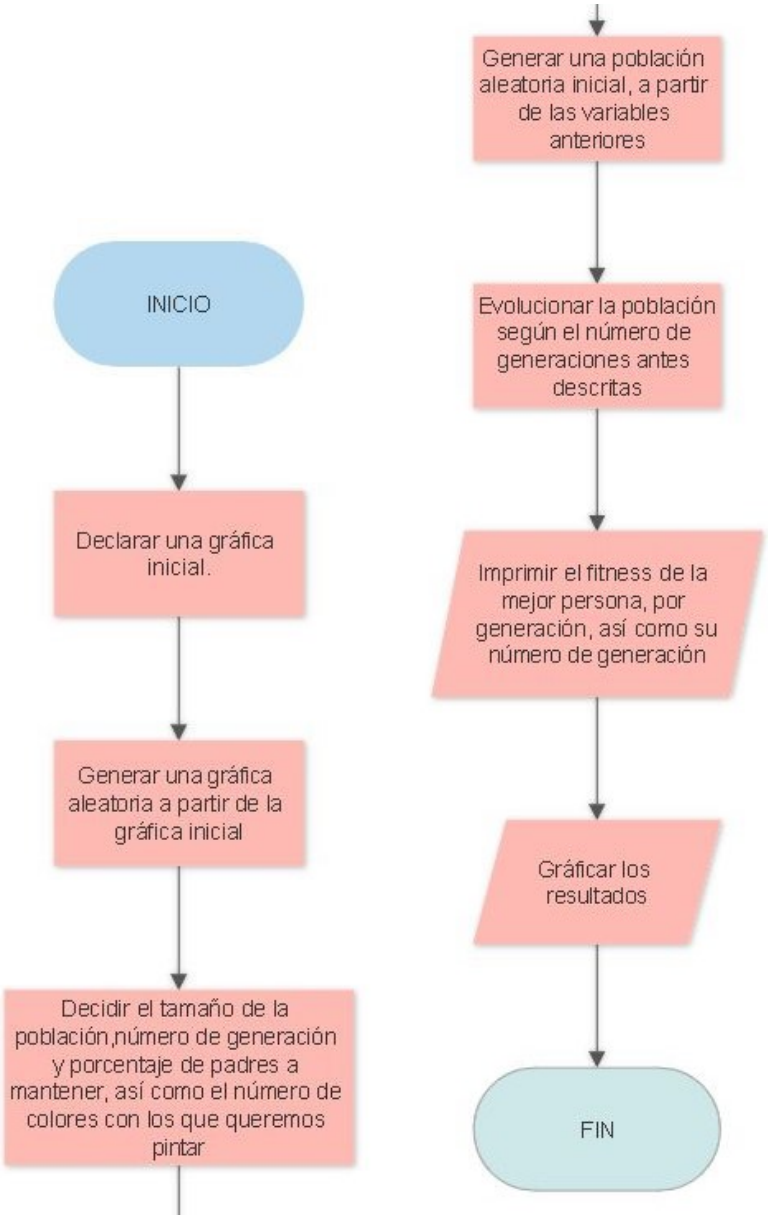


Figura 10: Algoritmo principal sin concurrencia

Observe que el tiempo de ejecución de evolucionar nuestra población depende directamente de el tamaño de nuestra población pues debemos generar tantos hijos como población tengamos, tener una población grande nos trae beneficios como mejorar nuestros individuos más rápido, sin embargo esto trae consigo la desventaja de hacer el programa más lento pues necesitamos calcular el fitness de cada gráfica en nuestra población así como generar más hijos, observe que podemos solucionar este problema implementando un algoritmo concurrente que evolucione cierta parte de nuestra población es decir crear nichos, un problema que tuvimos al implementar este programa es que en principio, podríamos pensar que es mejor crear un proceso tal que para cada individuo en nuestra población calcule su fitness por separado y de hay evolucionar nuestra población, sin embargo esto en vez de mejorar el programa lo hacia más lento, pues el número de procesos que debíamos de crear eran tantos como el número de individuos de nuestra población por el número de generaciones que queríamos generar. para la solucionar nuestro problema optamos por generar islas, de manera que cada isla tuviera cierto porcentaje de nuestra población y cada proceso se encargara de evolucionarlas por separado para finalmente compararlas entre ellas y encontrar la mejor coloración posibles, de esta manera la implementación de la imagen anterior se mejoro de la siguiente manera:

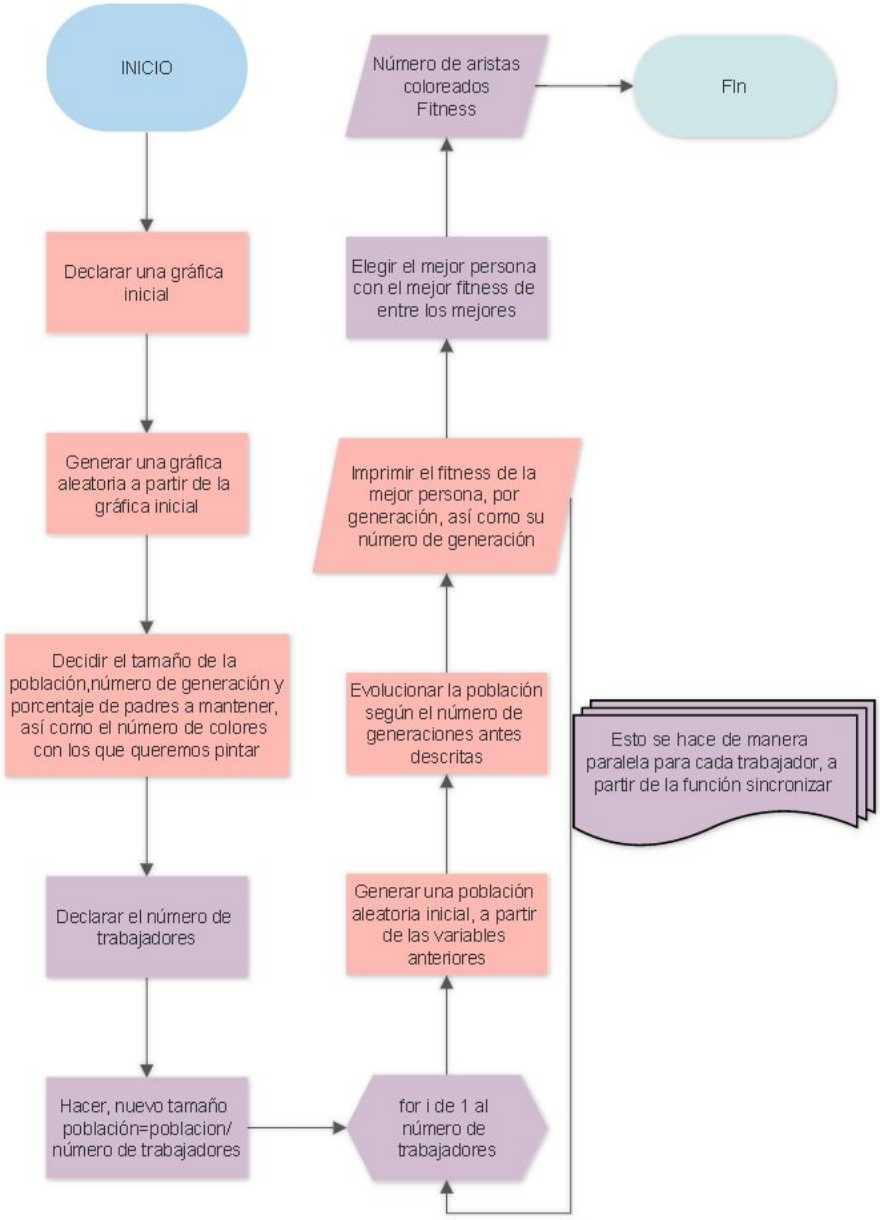


Figura 11: Algoritmo principal con una implementación concurrente

Como se puede observar para la implementación de manera concurrente, agregamos una nueva función que es la función sincronizar, esta se encargaría de hacer lo siguiente:

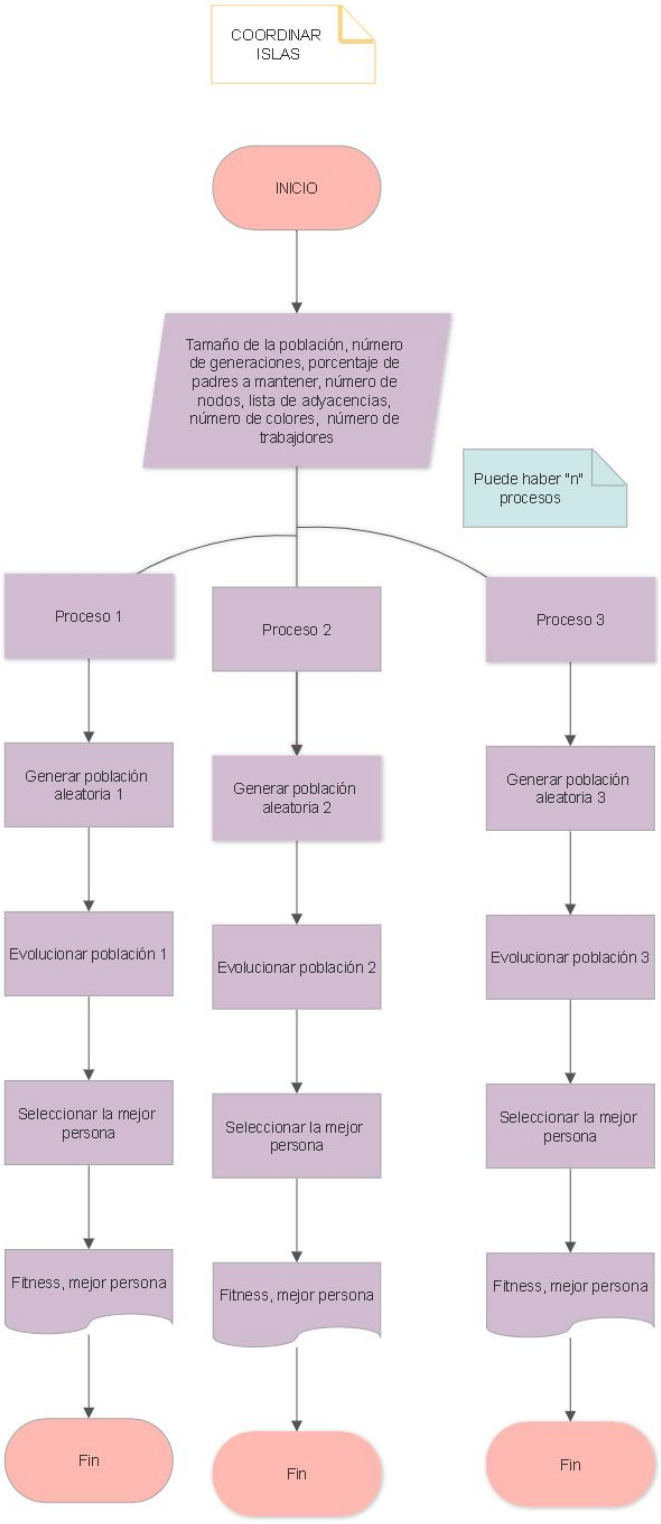


Figura 12: Algoritmo principal con una implementación concurrente

Es decir ahora lo que hacemos es evolucionar a cierta parte de la población de manera separada para luego elegir el fitness del mejor individuo. Podemos también obtener información más detallada de lo que hacen las funciones, generar población aleatoria y evolucionar población en los siguientes diagramas.

Figura 13: Algoritmo para generar una población aleatoria

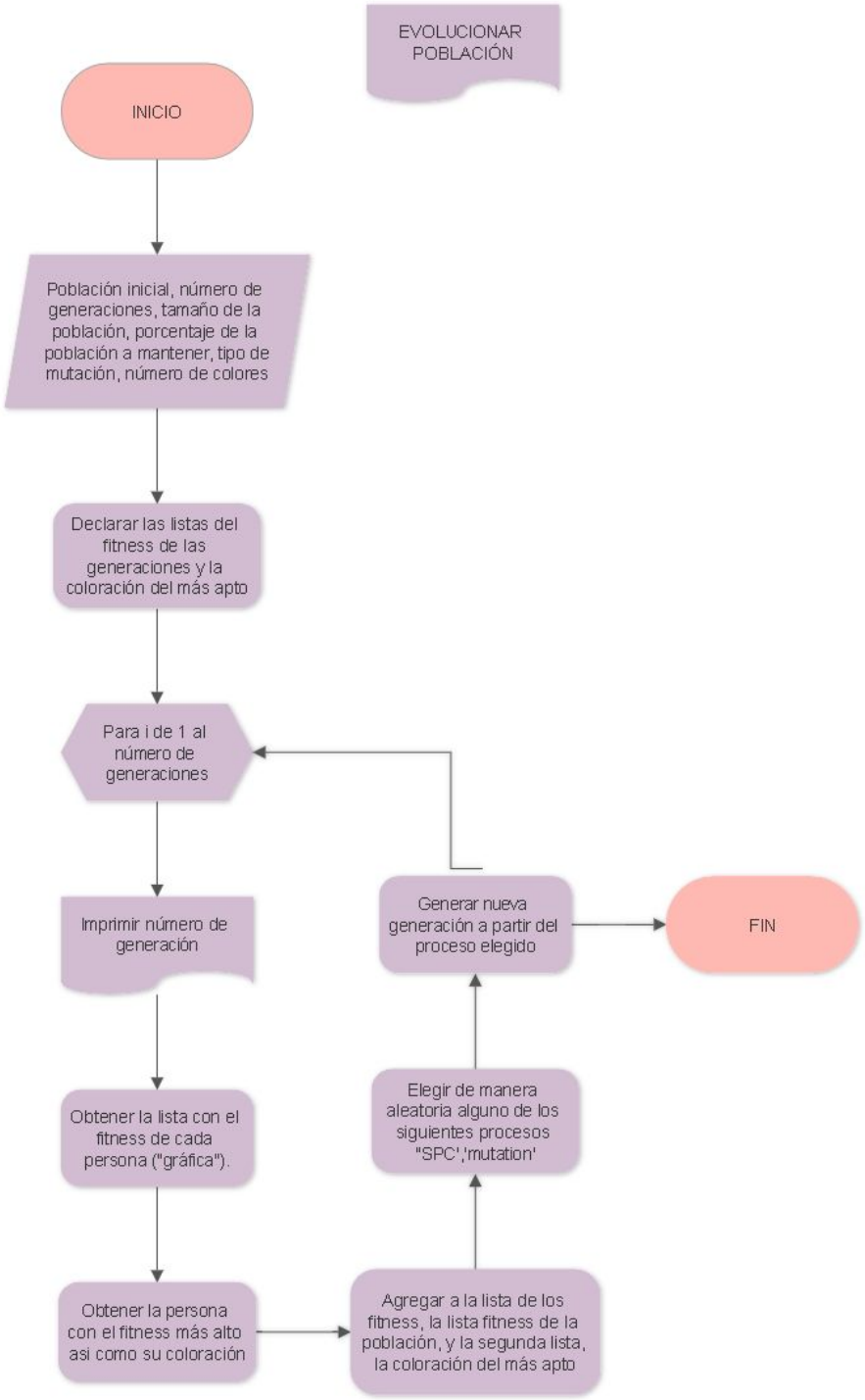


Figura 14: Algoritmo para evolucionar población

5. Documentación del Programa

En esta sección documentamos las funciones principales de nuestro programa, para no pegar todo el código solo pondremos que parámetros reciben así como lo que nos regresa cada función, nota los diagramas que describen el funcionamiento del programa se encuentran en la sección anterior. La implementación se realizo usando como base al código libre hecho por Jan Krepl [**Graph Colouring Repository**].

```
class Coloration:
    def __init__(self, colors, adj):
        """
        Creacion de un objeto Coloration
        :param colors:
        :type colors: string - contiene la coloración de la gráfica, si tenemos 3 colores puede ser 'rgggbbrg...
        :param adj: lista de adyacencia de la grafica
        :type adj: lista de listas
        :return: objecto Coloration
        :rtype: Coloration
        """
    def get_fitness(self, colors, adj):
        """
        Obtiene cuantas aristas que conecten a dos vértices de distinto color genera Coloration.
        :param colors:
        :type colors: string - contiene la coloración de la gráfica, si tenemos 3 colores puede ser 'rgggbbrg...
        :param adj: lista de adyacencia de la grafica
        :type adj: lista de listas
        :return: fitness de Coloration
        :rtype: int
        """

def parent_selection(input_population, number_of_pairs, method='FPS'):
    """
    Forma pares de padres, favoreciendo aquellos que tengan un mejor fitnesss
    :param input_population: poblacion de la que se desea obtener pares de padres
    :type input_population: lista de coloraciones
    :param number_of_pairs: numero de pares de padres deseados
    :type number_of_pairs: int
    :param method: metodo de seleccion
    :type method: str
    :return: lista de tuplas de pares de padres
    :rtype: lista de tuplas de Coloration
    """

def genetic_operator(pair_of_parents, method='SPC', n_colors=3):
    """
    Dados dos padres, regresa dos hijos resultantes de la cruza de estos
    :param pair_of_parents: par de padres
    :type pair_of_parents: lista que contenga dos padres
    :param method: metodo de cruza
    :type method: str
    :param n_colors: colores con los que se desea colorear la gráfica
    :type n_colors: int
    :return: par de hijos
    :rtype: par de Coloration
    """

def population_update(input_population, percentage_to_keep=0.1, genetic_op='mutation', n_colors=3):
    """
    Evoluciona la población input_population en una generación
    :param input_population: poblacion de la que se desea obtener pares de padres
    :type input_population: lista de Coloration
```

```
    :param percentage_to_keep: porcentaje de individuos que se preservan para la siguiente generación
    :type percentage_to_keep: int
    :param genetic_op: metodo de cruza
    :type genetic_op: str
    :param n_colors: colores con los que se desea colorear la gráfica
    :type n_colors: int
    :return: población evolucionada en una generación
    :rtype: lista de Coloration
    """

def generate_random_initial_population(population_size, n_nodes, adj, n_colors):
    """
    Genera una nueva población de manera aleatoria
    :param population_size: tamaño de la población
    :type population_size: int
    :param n_nodes: número de nodos
    :type n_nodes: int
    :param adj: lista de adyacencia de la gráfica que se desea colorear
    :type adj: lista de listas
    :param n_colors: colores con los que se desea colorear la gráfica
    :type n_colors: int
    :return: población aleatoria
    :rtype: lista de objetos Coloration
    """

def find_fittest(input_population):
    """
    Encuentra Coloration con mayor fitness de input_population
    :param input_population: poblacion de la que se desea obtener pares de padres
    :type input_population: lista de coloraciones
    :return: Coloration con mayor fitness de input_population
    :rtype: Coloration
    """

def evolution(input_population, n_generations, percentage_to_keep, n_colors, n_edges):
    """
    Regresa lista con la persona con mayor fitness para cada generación
    :param input_population: poblacion de la que se desea obtener pares de padres
    :type input_population: lista de Coloration
    :param n_generations: número de generaciones a simular
    :type n_generations: int
    :param percentage_to_keep: porcentaje de individuos que se preservan para la siguiente generación
    :type percentage_to_keep: int
    :param n_colors: colores con los que se desea colorear la gráfica
    :type n_colors: int
    :param n_edges: número de aristas en la gráfica que se desea colorear
    :type n_edges: int
    :returns: Lista con la persona con mayor fitness para cada generación
    :rtype: Coloration
    """

def visualize_results(islands_fittest, n_edges):
    """
    Genera visualización de generación vs fitness para cada isla
    :param islands_fittest: contiene listas del fitness para cada isla para cada generación
    :type islands_fittest: lista de listas
    :param n_edges: número de aristas en la gráfica que se desea colorear
    :type n_edges: int
    """
```

```
def coordinate(population_size, n_generations, percentage_to_keep, n_nodes, n_edges, n_colors, adj, n_islands):
    """
    Crea los procesos que se ejecutan de forma paralela y encuentra el individuo con mayor fitness
    entre todas las islas
    :param population_size: tamaño de la población que se desea simular
    :type population_size: int
    :param n_generations: número de generaciones a simular
    :type n_generations: int
    :param percentage_to_keep: porcentaje de individuos que se preservan para la siguiente generación
    :type percentage_to_keep: int
    :param n_nodes:
    :type n_nodes:
    :param n_edges: número de aristas en la gráfica que se desea colorear
    :type n_edges: int
    :param n_colors: colores con los que se desea colorear la gráfica
    :type n_colors: int
    :param adj: lista de adyacencia de la gráfica que se desea colorear
    :type adj: lista de listas
    :param n_islands: número de islas en las que distribuira la población (número de procesos que se crearan)
    :type n_islands: int
    :return: persona con mayor fitness de entre todas las islas
    :rtype: Coloration
    """
    islands_params = [(population_size//n_islands, n_nodes, adj, n_colors)]*n_islands
    with Pool(n_islands) as p:
        islands = p.starmap(generate_random_initial_population, islands_params)

    new_islands_params = [(island, n_generations, percentage_of_parents_to_keep, n_colors, n_edges) for island ir
    with Pool(n_islands) as p:
        islands_fittest = p.starmap(evolution, new_islands_params)

    visualize_results(islands_fittest, n_edges)

    best_fit = 0
    for island_fittest in islands_fittest:
        best_fit = max(best_fit, island_fittest[-1])

    return best_fit

def get_graph(name):
    """
    Regresa número de nodos, número de airstas, número cromatico y matriz de adyacencia de la gráfica name
    :param name: nombre de la gráfica
    :type name: str
    :return n_nodes, n_edges, n_colors, adj:
    """

if __name__ == '__main__':
    """
    Función principal, aqui se definen todos los parámetros del modelo como population_size,
    n_generations y n_colors.
    """
    names = ['anna', 'david', 'games120', 'homer', 'huck', 'jean', 'miles1000', 'miles250',
    'myciel3', 'myciel4', 'myciel5', 'queen5_5', 'queen6_6', 'queen7_7', 'queen8_8']
    names.sort()
    for graph_name in names:
        t1 = time.time()
        n_nodes, n_edges, n_colors, adj = get_graph(graph_name)
```

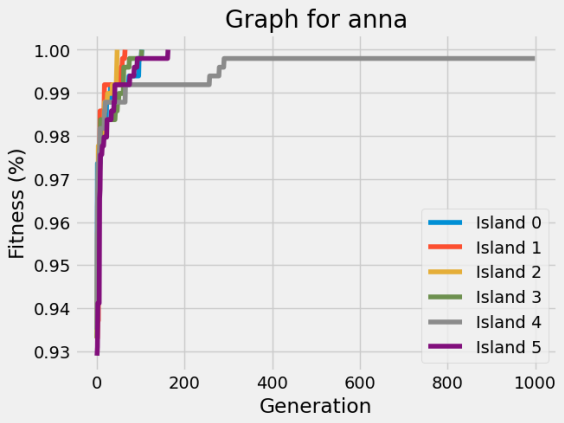
```
population_size = 300
n_generations = 1000
percentage_of_parents_to_keep = 0.3
n_islands = 6
print('Graph name: ', graph_name)
print('Number of nodes: ' + str(n_nodes))
print('Number of edges: ' + str(n_edges))
print('Number of colors: ' + str(n_colors))
fittest = coordinate(population_size, n_generations, percentage_of_parents_to_keep,
                     n_nodes, n_edges, n_colors, adj, n_islands)
print(f'Fittest: {fittest: .2f}')
t2 = time.time()
total_time = t2-t1
print(f"{total_time: .2f}")
```

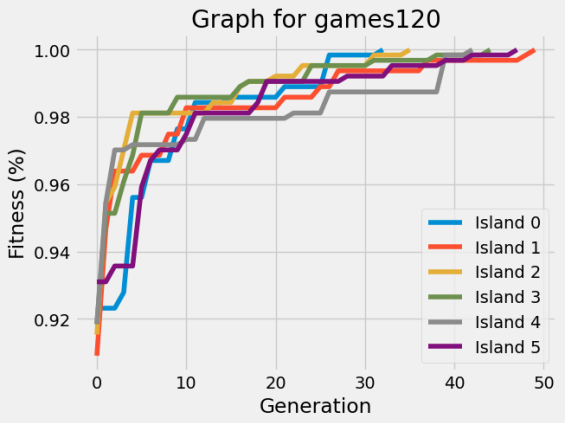
6. Resultados

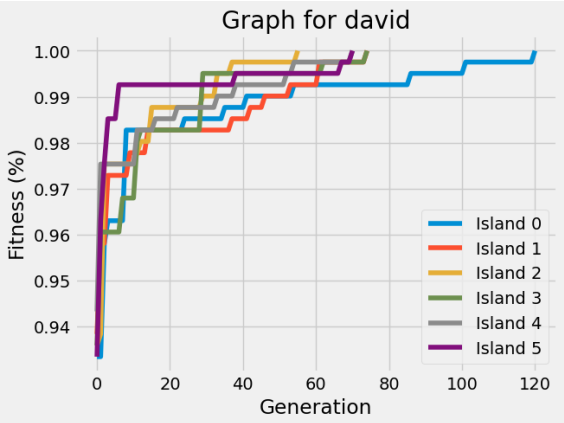
6.1. Descripción de resultados y convergencia de las soluciones

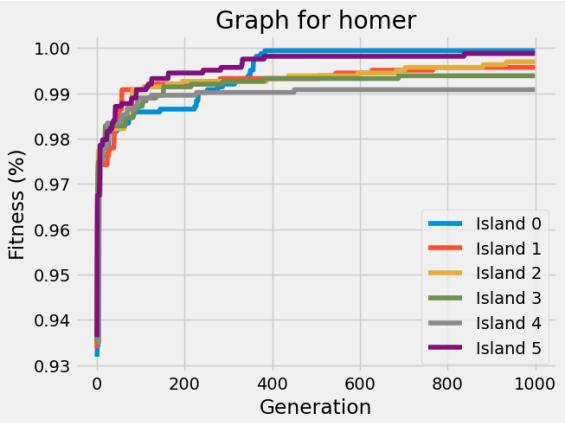
A continuación el análisis de convergencia para las gráficas de prueba (Table 9). Se muestra el número de generaciones vs el fitness (en porcentaje) para cada gráfica, se utilizaron los siguientes parámetros para el modelo:

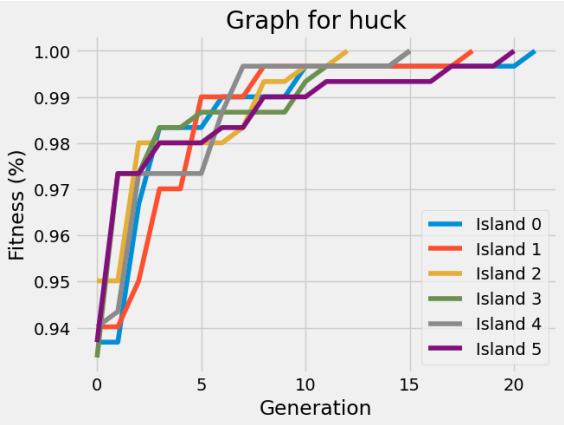
- population_size = 700
- n_generaciones = 1000 (máximo)
- n_islands = 6
- percentage_to_keep = 30 %

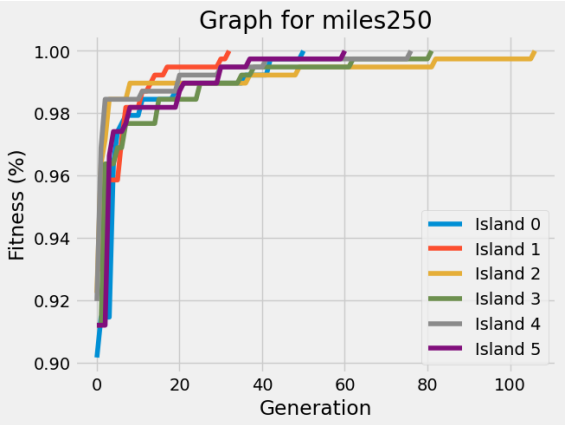


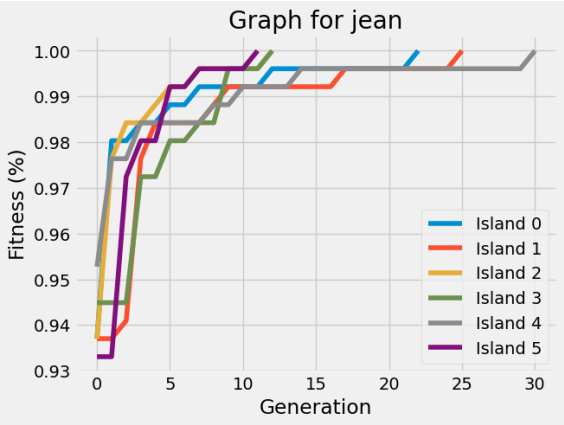


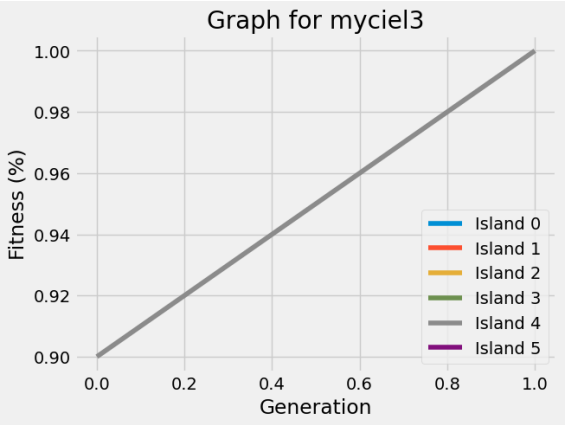


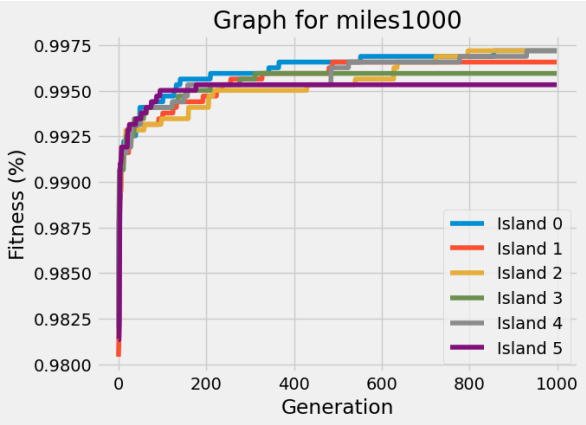


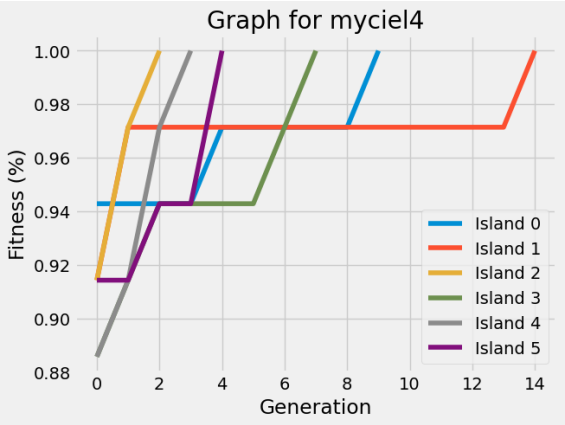

Graph for huck
Fitness (%)
Generation
Island 0
Island 1
Island 2
Island 3
Island 4
Island 5

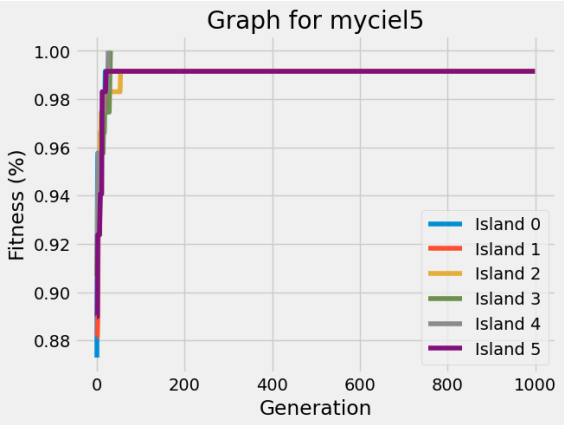

Graph for miles250
Fitness (%)
Generation
Island 0
Island 1
Island 2
Island 3
Island 4
Island 5

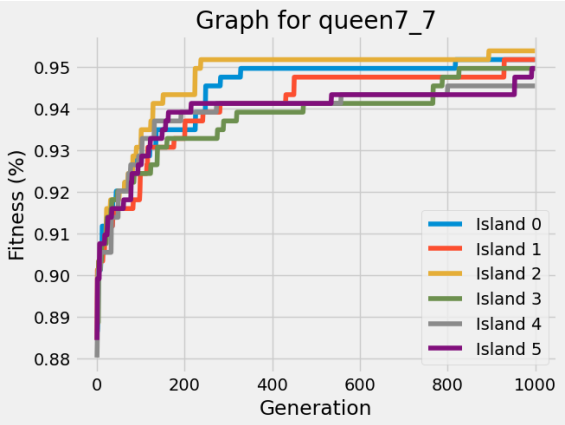

Graph for jean
Fitness (%)
Generation
Island 0
Island 1
Island 2
Island 3
Island 4
Island 5

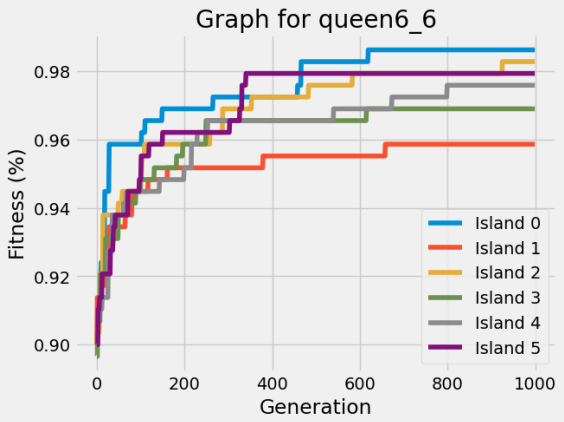

Graph for myciel3
Fitness (%)
Generation
Island 0
Island 1
Island 2
Island 3
Island 4
Island 5

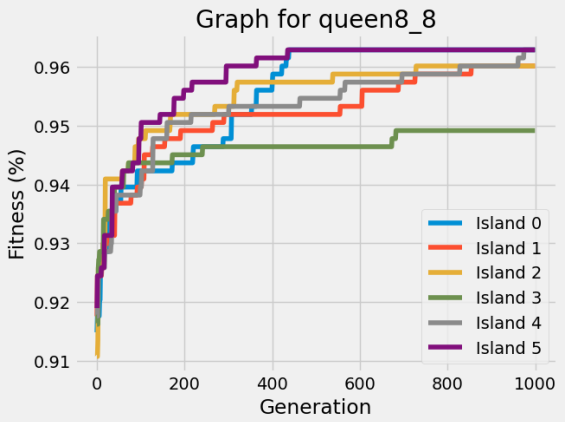

Graph for miles1000
Fitness (%)
Generation
Island 0
Island 1
Island 2
Island 3
Island 4
Island 5


Graph for myciel4
Fitness (%)
Generation
Island 0
Island 1
Island 2
Island 3
Island 4
Island 5









6.2. Comparación para cada tipo de gráfica.

La siguiente tabla nos muestra, que porcentaje de aristas fueron coloreadas para las gráficas de prueba (Tabla 9). Se utilizaron los siguientes parámetros para el modelo:

- n_generations = 4000
- population_size = 750
- n_islands = 15
- percentage_to_keep = 30 %

Gráfica	$\|V\|$	$\|E\|$	colores	$\|E\|$ coloreadas	Porcentaje
myciel3	11	10	4	10	100
myciel4	23	35	5	35	100
myciel5	36	118	6	118	100
queen5_5	23	160	5	160	100
queen6_6	25	290	7	288	99.31
queen7_7	49	476	7	461	96.8
huck	74	301	11	301	100
jean	80	254	10	254	100
david	87	406	11	406	100
games120	120	638	9	638	100
miles250	128	387	8	387	100
miles1000	128	3216	42	3212	99.8
anna	138	493	11	493	100
homer	561	1629	13	1628	100

[10]

6.3. Tiempos de ejecución.

Realizamos las ejecuciones correspondientes a cada gráfica con el algoritmo concurrente y su versión secuencial, y comparamos los tiempos de ejecución para hacer una verificación empírica de la eficiencia de nuestra programa. Para las gráficas conocidas, obtuvimos los siguientes tiempos de ejecución en segundos para ambas versiones del algoritmo la implementación concurrente y la secuencial.

Gráfica	$\|V\|$	$\|E\|$	colores	T Paralelo	T Secuencial
myciel3	11	10	4	13.04	61.80
myciel4	23	71	5	20.72	86.81
myciel5	36	236	6	38.43	219.74
queen5_5	23	160	5	103.66	708.95
queen6_6	25	290	7	140.51	1182.72
queen7_7	49	476	7	277.09	1949.53
huck	74	301	1	53.63	396.7
jean	80	254	10	50.17	352.58
david	87	406	11	53.59	377.58
games120	120	638	9	113.55	729.09
miles250	128	387	8	66.13	384.33
miles1000	128	3216	42	444.3	3321.67
anna	138	493	11	68.15	479.34
homer	561	1629	13	361.60	2745.17

6.4. Análisis de convergencia, tiempos de ejecución y soluciones.

6.4.1. Convergencia

De las gráficas para las cuales implementamos nuestro programa se pueden realizar las siguientes observaciones:

- La mayoría de las gráficas con las que implementamos nuestro programa lograron llegar a una coloración óptima o casi óptima.
- La mejor solución de la población inicial (aleatoria) suele tener un buen fitness (por arriba del 90 % en la mayoría de los casos)
- La convergencia puede llegar a depender fuertemente de las población inicial (en nuestro caso es aleatoria) como se muestra en *queen*8_8, donde la isla 3 quedo por debajo de todas las demás, mientras que la isla 5 convergió mucho más rápido que las demás.
- Por el punto anteriormente mencionado, el utilizar un mayor número de islas nos asegurara una convergencia más rápida.

6.4.2. Tiempos de Ejecución

Se muestra una gráfica que compara los tiempos de ejecución del algoritmo secuencial contra el concurrente, algoritmo concurrente en naranja y paralelo en azul

[10] $|V|$Representa los vértices y $|E|$ representa las aristas.

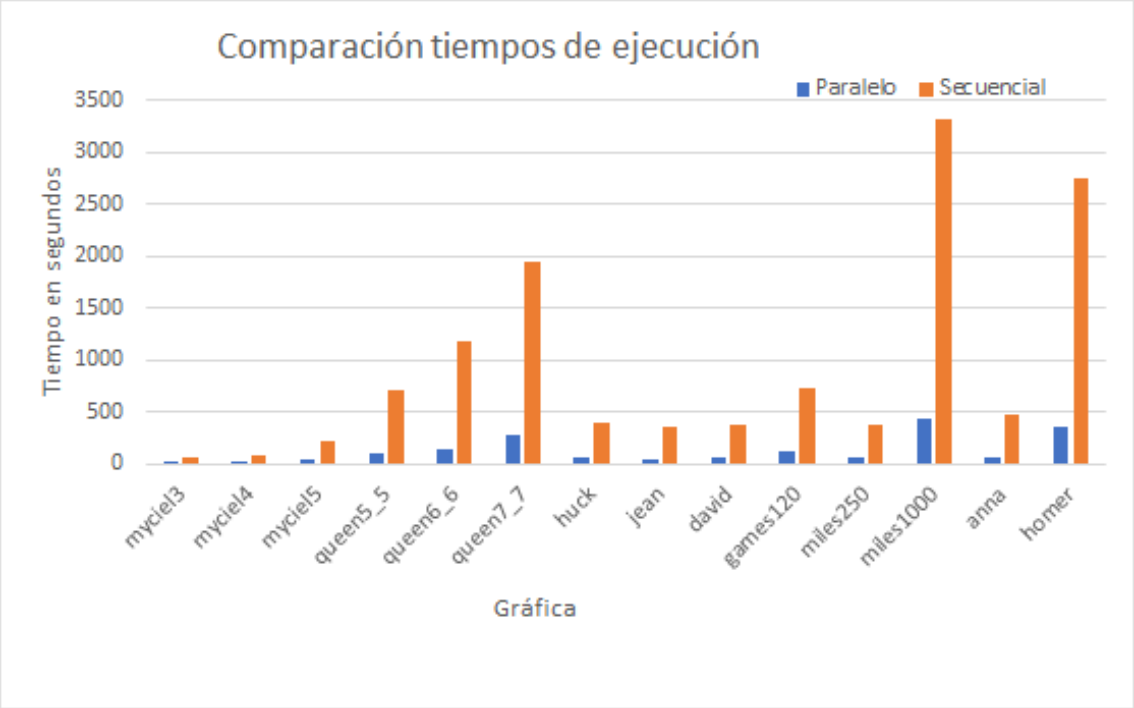
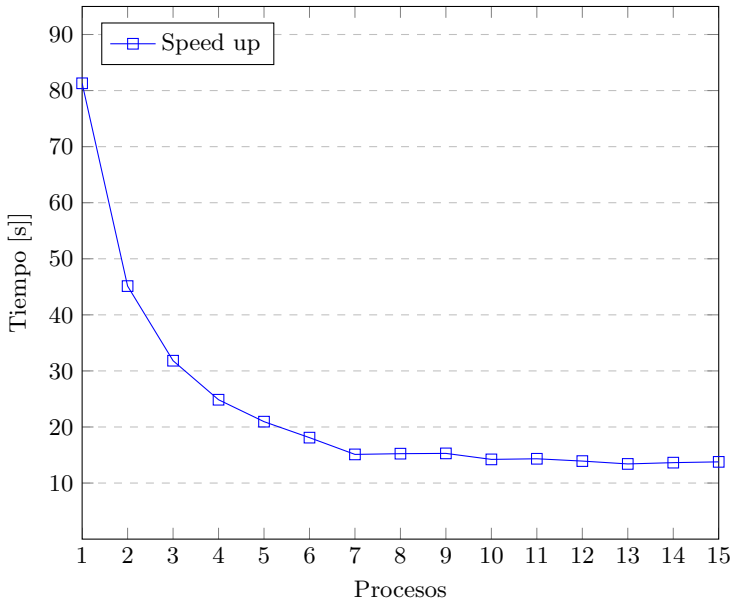


Figura 15: Comparación de tiempos de ejecución de algoritmo concurrente y secuencial.

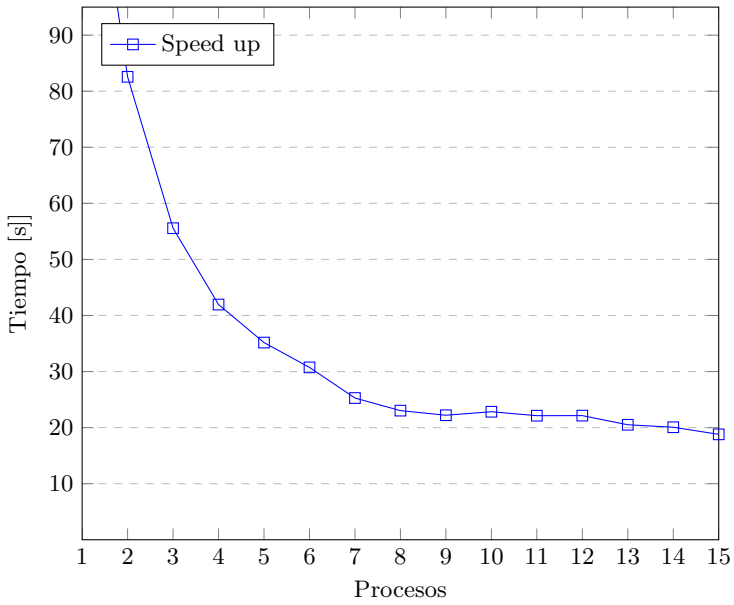
Observe que el tiempo de ejecución es de 3 a 6 veces mejor dependiendo de la gráfica que estemos coloreando. Por otra parte podemos hacer un análisis de los tiempos de ejecución al variar el número de procesos con respecto a las gráficas *myciel3, myciel4, myciel5*, como se puede ver en los siguientes diagramas. [11]

Gráfica: *myciel3*

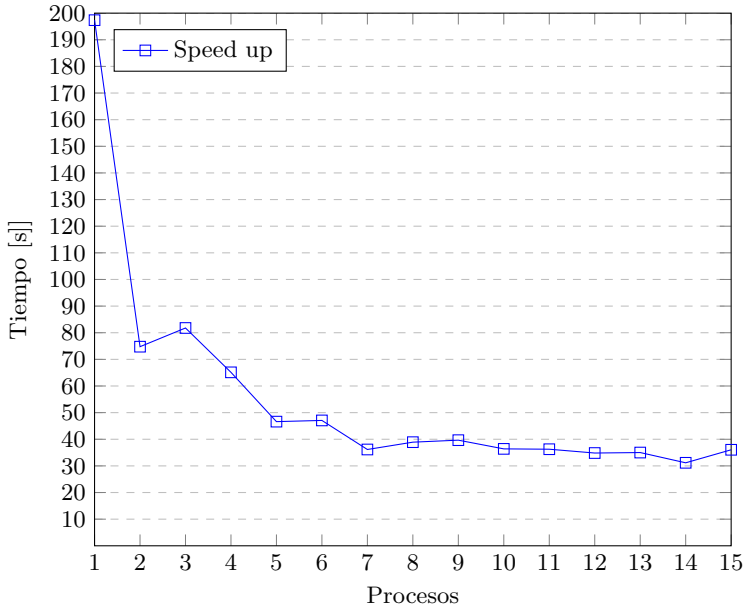


[11] Todas las mediciones de comparación se realizaron en un equipo de computo personal, con procesador Ryzen 7 2700 y memoria RAM HyperX Fury DDR4, 3466MHz

Gráfica: *myciel4*



Gráfica: *myciel5*



Observe que el tiempo de ejecución parece no disminuir una vez que aumentamos el número de procesos más allá de 7

7. Conclusiones

Como dijimos anteriormente el problema de colorear una gráfica aunque tiene un gran número de aplicaciones algunas de las cuales se mencionaron en el abstract, el problema de colorear una gráfica es un problema NP-completo es decir que no se conoce por el momento ningún algoritmo que sea capaz de encontrar una coloración de la gráfica en tiempo polinomial por lo que para un número pequeño de vértices y aristas, cualquier algoritmo que nos asegure la coloración de la gráfica se vuelve imposible de ejecutar en un corto periodo de tiempo.

- Como se pudo ver durante el desarrollo del proyecto aún cuando el problema sea NP es posible llegar a la aproximación de una solución a partir de una algoritmo evolutivo, en este caso un algoritmo genético y aunque no siempre es posible llegar a una solución como es el caso de las gráficas tipo *myciel*, muchas veces si podemos asegurar la solución o una aproximación a ella en un corto periodo de tiempo por ejemplo para la gráfica $miles1000$ que tiene 3216 aristas, seria imposible de colorear con un algoritmo con complejidad no polinomial, sin embargo nuestro algoritmo evolutivo fue capaz de aproximar una solución en tan solo $3321{,}67s$ es decir aproximadamente cincuenta y cinco minutos.
- Usando una implementación concurrente se logro reducir nuestro tiempo de ejecución de 3 a 6 veces, por ejemplo para la gráfica $miles1000$ el tiempo paso de ser de 55 minutos a un poco menos de 8 minutos.
- Al analizar lo que pasa cuando aumentábamos el número de procesos al colorear cierta gráfica, observamos que el tiempo tiende a reducirse pero después de cierto número de procesos aproximadamente 7 ya no vemos una mejoría notable en el tiempo de ejecución, esto puede deberse muy probablemente a la ley de Amdahl. [12]

[12] La ley de Amdahl se usa a menudo en computación paralela para predecir la aceleración teórica del tiempo de ejecución cuando se usan múltiples procesadores

Referencias

[1] D. A. Coley, An introduction to genetic algorithms for scientists and engineers, World Scientific Publishing Company, 199.

[2] Musa M. Hindi and Roman V. Yampolskiy.2011 Genetic Algorithm Applied to the Graph Coloring Problem, Speed School of Engineering Louisville, Kentucky URL: http://ceur-ws.org/Vol-841/submission_10.pdf

[3] Graph color instances, consultado 5 de febrero de 2021, base de datos de distintas gráficas https://mat.tepper.cmu.edu/COLOR/instances.html

[4] S. Poddar, Parallel Genetic Algorithm, 2020. URL: https://medium.com/swlh/parallel-genetic-algorithm-3d3314c8373c

[5] J. Krepl. Graph Colouring Problem, github URL: https://github.com/jankrepl/Graph_Colouring

[6] Evolutionary Algorithms Review Arm Inc., Bellevue MAANA Inc., Bellevue

[7] E. Cuevas, J. Osuna, D. Olivia, M. Diaz Optimización:Algoritmos programados con MATLAB,Alfaomega, 2016

[8] Back, T., Hammel, U., and Schwefel, H. P. 1997. Evolutionary computation: comments on the history and current state. IEEE Transactions on Evolutionary Computation 1: 3-17

[9] Diaz, Isabel Méndez, and Zabala, Paula. 1999, A Generalization of the Graph Coloring Problem, Departamento de Computacion, Universidad de Buenes Aires.

[10] Yampolskiy, Roman V., and El-Barkouky, Ahmed. 2011. Wisdom of Artificial Crowds Algorithm for Solving NP-Hard Problems. International Journal of Bio-Inspired Computation (IJBIC) 6: 358-369

[11] Glass, C. A., and Prugel-Bennett, A. 2003. Genetic algorithm for graph coloring: Exploration of Galinier and Hao's algorithm. Journal of Combinatorial Optimization 3: 229-236

[12] Yi, Sheng Kung Michael, Steyvers, Mark, Lee, Michael D., and Dry, Matthew J. 2010b. Wisdom of the Crowds in Traveling Salesman Problems. URL: http://www.socsci.uci.edu/~mdlee/YiEtAl2010.pdf

[13] Back, T., Hammel, U., and Schwefel, H. P. 1997. Evolutionary computation: comments on the history and current state. IEEE Transactions on Evolutionary Computation 1: 3-17

[14] D. Gutierrez, A. Tapia ,Algoritmos Genéticos con Python: Un enfoque práctico para resolver problemas de ingeniería. 2020