Casa abierta al tiempo
UNIVERSIDAD AUTÓNOMA METROPOLITANA

PC y TI

UNIVERSIDAD AUTÓNOMA METROPOLITANA – IZTAPALAPA
DIVISIÓN DE CIENCIAS BÁSICAS E INGENIERIA

BÚSQUEDA DISPERSA PARA EL PROBLEMA DE COLORACIÓN DE GRÁFICAS SUAVES

Tesis que presenta:
Hugo Eduardo Vásquez Calderón
Para obtener el grado de
Maestro en Ciencias y Tecnologías de la Información

Correo electrónico: tlhugo.calderon@gmail.com
sagitariohevc@gmail.com

Asesores: Dr. Pedro Lara Velázquez
Dr. Sergio G. De los Cobos Silva

Jurado calificador:
Presidente: Dr. Miguel Ángel Gutiérrez Andrade
Secretario Dr. Pedro Lara Velázquez
Vocal: Dra. Hérica Sánchez Larios

México, ciudad de México, noviembre de 2017

Agradecimientos

Al Dr. Pedro Lara Velázquez por su apoyo, enseñanzas, paciencia, comprensión y amistad. Gracias por las charlas, risas y anécdotas compartidas a lo largo de este proyecto de investigación, que de no ser por su propuesta todo esto no sería posible.

Al Dr. Sergio G. De Los Cobos Silva gracias por aceptar ser asesor de este proyecto y sobre todo por la retroalimentación, consejos, apoyo y paciencia que tuvo a lo largo de este, sin los cuales no sería posible.

Al Dr. Miguel Ángel Gutiérrez Andrade por sus enseñanza, consejos, paciencia y retroalimentación para hacer de este proyecto un mejor trabajo de investigación. Y gracias por aceptar ser sinodal.

A la Dra. Hérica Sánchez Larios por aceptar ser sinodal de este trabajo.

A mis padres Agustina Calderón Bustillos y Eduardo Hugo Vásquez Lagunas por su apoyo, paciencia, comprensión, solidaridad, cariño, amor y por ser ejemplos a seguir. Son el motor de mi vida y los cimientos de mi persona, son los mejores padres, gracias por todo ya que sin ustedes no estaría cumpliendo un objetivo más en mi vida.

A mi tía Tony por los desvelos, las pláticas, las risas y los momentos que compartimos, gracias por sus cuidados, cariño, amor, paciencia y solidaridad.

A mis tíos, tías, primos y primas por su apoyo, consejos, cariño y amor.

A Alejandra Martínez por su comprensión, cariño, consejos, paciencia, apoyo y amor incondicional. Sin tu presencia, mi vida carecería de la luz que expides con tu alegría, optimismo y felicidad que contagia a todos los que te rodean, gracias por ser parte de mi vida, Te amo.

A Hortensia C. Concha y Claudia Martínez por su apoyo, optimismo, cariño, solidaridad y consejos. Gracias por su calidez, cariño y confianza por permitirme entrar a su casa y ser parte de su vida.

Al M. en CyT Daniel Urueta por su confianza, apoyo, consejos y amistad. Gracias por las pláticas, risas, anécdotas y enseñanzas a lo largo de este proyecto.

Al M. en CyT Miguel Ortiz y al próximo M. en CyT Eric Márquez por su apoyo, consejos, enseñanzas, platicas y sobre todo por su amistad.

Al posgrado en Ciencias y Tecnologías de la Información de la UAM Unidad Iztapalapa y todo el personal que lo conforma por brindarme la oportunidad de crecer académica y personalmente.

Al CONACyT por otorgarme la beca para poder cursar el posgrado y cumplir con otra meta académica en mi vida.

Contenido

Resumen

El modelo de coloración de gráficas suaves es un caso particular del problema de coloración de gráficas, donde se busca una coloración que minimice la dureza de una gráfica completa con "*n*" vértices y los cuales están unidos por aristas con penalizaciones. El modelo de gráficas suaves ha probado que se puede resolver otros problemas de coloración, se ha utilizado como clasificador no supervisado, también para problemas de reconocimiento de patrones para comparar diferencias lingüísticas. Es un problema NP-Duro, por lo cual, para instancias mayores a 20 vértices, es necesario el uso de metaheurísticas para encontrar su solución. Es por esto que en esta investigación se desarrolla un algoritmo de búsqueda dispersa con la implementación de las mejores estrategias para cada uno de sus métodos, también se desarrolla instancias para poner aprueba el algoritmo y de esta formar corroborar las mejores estrategias a implementar. A su vez se compara con otras metaheurísticas para el problema de coloración robusta utilizado el modelo de coloración de gráficas suaves y nuestro algoritmo iguala las mejores soluciones obtenidas.

Introducción

La optimización matemática se puede agrupar en 2 grandes categorías: optimización continua y optimización discreta (Lovász 2010). En los problemas continuos se busca la solución dentro de un conjunto de números reales continuos, con la principal cualidad de que si se tomando dos números del conjunto entre de ellos hay números infinitos, por su parte en los problemas discretos o también conocidos como problemas combinatorios buscas las soluciones en un conjunto finito, se pude definir una función de evaluación que determine la calidad de cada miembro del conjunto (de los Cobos Silva et al. 2010).

La teoría de grafos es un problema discreto de las ciencias de la computación que estudia la relación entre los objetos de una colección y los representa a través de estructuras matemáticas conocidas como gráficas. Una gráfica es un conjunto de vértices (puntos) conectados por medio de aristas (líneas), pueden tener dirección y peso. La teoría de grafos tiene aplicaciones en distintas áreas como: minería de datos, redes, segmentación de imagines, etc (Riaz y Ali 2011).

La familia de problemas de coloración de gráficas es parte de la teoría de grafos. Se trata de un problema combinatorio el cual consiste en asignar un color k a cada vértice "n" de una gráfica G, de tal manera que vértices unidos por una arista tengan distinto color k. Su complejidad es de k^n, es el problema de coloración de gráficas más antiguo.

Los problemas de coloración de gráficas tienen características particulares que los diferencian y que facilita o complican su implementación para solucionar ciertos problemas. La familia de coloración de gráficas está compuesta por: el problema de coloración mínima, el problema de coloración equitativa, el problema de coloración de gráficas débiles, el problema de coloración de gráficas suaves y el problema de coloración robusta.

La complejidad de los problemas de coloración de gráficas varia de NP-Completo a NP-Duro, por lo tanto, son problemas que son considerados irresolubles por algoritmos polinomiales. Para algunos problemas de coloración de gráficas cuando se tiene un número de vértices menor igual a 20 se pueden obtener soluciones exactas a través de algoritmos exactos, pero cuando se tiene más de 20 vértices los algoritmos exactos son incapaces de hallar una solución, por lo cual para estos problemas es necesario el uso de heurísticas.

El modelo de coloración de gráficas suaves es un caso particular del problema de coloración de gráficas, donde se busca una coloración que minimice la suma de las penalizaciones entre aristas incidentes cuyo vértice tiene el mismo color. La coloración de gráficas suaves a diferencia de la coloración de gráficas tradicional puede tener dos vértices unidos por una arista con el mismo color esto debido a que una de sus aplicaciones es como clasificador (Flores Cruz et al., s/f), otras aplicaciones son optimización y calendarización.

La coloración de gráficas suaves es un problema NP-Duro lo que significa que son problemas que toman como mínimo un tiempo polinomial no determinista para encontrar una solución (Wegener 2005), por lo que a partir de 20 vértices el uso de algoritmos exactos es infactible y se necesita utilizar heurísticas o metaheurísticas para hallar buenas soluciones en tiempos de ejecución aceptables. Este problema se ha demostrado que puede resolver otros problemas tales como coloración mínima, coloración equitativa, coloración robusta entre otros.

El objetivo de los algoritmos heurísticos es encontrar de manera rápida y sencilla una buena solución a un problema, no necesariamente es la solución óptima. Es por esto que las heurísticas son estrategias que son utilizadas cuando un algoritmo exacto o clásico es incapaz de encontrar la solución de un problema o para hallar dicha solución le tomaría un periodo de tiempo extremadamente prolongado y por lo tanto al finalizar la solución carecería de importancia y relevancia. Un ejemplo de una heurística para los problemas de optimización combinatoria es la búsqueda local la cual es un algoritmo sencillo que da buenos resultados de manera rápida. La cual consiste en simplemente realizar ligeros cambios a la solución para de esta forma ir encontrando una mejor solución.

Las metaheurísticas, por su parte son, algoritmos que, a diferencia de las heurísticas, van más allá de encontrar soluciones de manera rápida y sencilla, sino que buscan encontrar soluciones que sean lo más cercanas posibles al óptimo, por lo cual son soluciones de alta calidad y que son encontradas en tiempos de ejecución aceptables. Para lograr mejores resultados las metaheurísticas utilizan las soluciones encontradas por las heurísticas y utilizan otros métodos para mejorar dichos resultados, algunos ejemplos de estos algoritmos son: GRASP, búsqueda Tabú, recocido simulado, algoritmos genéticos, búsqueda dispersa, entre otros.

Búsqueda dispersa o scatter search es un algoritmo metaheurístico para resolver problemas de optimización. Su concepto y fundamentos fueron propuestos a principios de los 70, fue descrito por primera vez en 1977 por Fred Glover, pero es hasta 1998 (Glover 1998) que establece un descripción formal y completa del algoritmo y sus métodos. La búsqueda dispersa es considerada un algoritmo evolutivo, ya que genera nuevas soluciones a partir de las mejores soluciones obtenidas previamente. Se compone principalmente de cinco métodos los cuales son

extremadamente flexibles, por lo cual estos métodos pueden utilizar distintas estrategias para facilitar su implementación y dar solución a distintos problemas.

El objetivo principal de esta investigación es el desarrollo de un algoritmo de solución para el problema de coloración de gráficas suaves utilizando la técnica metaheurística de búsqueda dispersa. Los objetivos particulares son.

- Revisión del estado del arte del problema de coloración de gráficas suaves y del algoritmo de búsqueda dispersa

- Planteamiento de instancias apropiadas para verificar la calidad del algoritmo.

- Generar un algoritmo de búsqueda dispersa con las mejores estrategias encontradas en el estado del arte para de esta manera tener un algoritmo que proporcione soluciones de calidad en tiempos de ejecución aceptables.

- Implementación del algoritmo en Python y la validación a través de las instancias seleccionadas.

Este trabajo consta de seis capítulos:

El capítulo 1 corresponde a los fundamentos del problema de coloración de gráficas suaves como también del algoritmo de búsqueda dispersa. A su vez se mencionan los distintos problemas de coloración de gráficas que existe y se muestran algunos de los métodos metaheurísticos más utilizados para resolver problemas de coloración de gráficas.

El capítulo 2 consiste en la descripción detallada de las distintas estrategias seleccionadas para implementar en cada uno de los cinco métodos principales del algoritmo de búsqueda dispersa, como también se describe su algoritmo genérico y la variación del algoritmo utilizado un criterio de paro distinto.

El capítulo 3 explica el lenguaje de programación escogido, como también se detalla la implementación del algoritmo de búsqueda dispersa en forma de seudocódigo, y se hace mención de las particularidades propias del lenguaje de programación en el que fue implementado.

El capítulo 4 describe la manera en que fueron creadas las instancias utilizadas para probar las distintas estrategias implementadas para cada método del algoritmo de búsqueda dispersa, como

da el contexto de las instancias utilizadas para probar que el modelo de coloración de gráficas suaves utilizado el algoritmo de búsqueda dispersa puede dar solución a otros problemas de coloración de gráficas como en este caso es el problema de coloración robusta.

En el capítulo 5 se presentan los resultados obtenidos de las distintas estrategias implementadas para cada uno de los cinco métodos del algoritmo de búsqueda dispersa. Con estos resultados se realiza un análisis para determinar que combinación de estrategias nos brinda un mejor y eficiente algoritmo de búsqueda dispersa basándonos en los datos fundamentales del modelo de coloración de gráficas suaves como lo son dureza, solidez y resiliencia. Con la combinación ideal de estrategias analizamos los resultados obtenidos de aplicar el algoritmo de búsqueda dispersa y el modelo de coloración de gráficas suaves para el problema de coloración robusta, obteniendo los mismos resultados idóneos que algoritmos implementados con el modelo de coloración robusta.

En el capítulo 6 se dan las conclusiones de esta investigación.

Metodología

Como primer paso se realizó una investigación bibliográfica con respecto a la familia de problemas de coloración de gráficas, con lo cual se determinó que los problemas de coloración de gráficas mínima, coloración de gráficas débiles, coloración equitativa y en específico el problema de coloración robusta son las más utilizadas de esta familia. A su vez se investigó los algoritmos utilizados para darles solución y sus aplicaciones.

Los algoritmos más utilizados fueron: GRASP, búsqueda Tabú, algoritmos genéticos, búsqueda de vecindarios variable, recocido simulado, algoritmos bio-inspirados y búsqueda dispersa. Por lo cual ser realizo una investigación sobre su funcionamiento, características principales, desempeño y si se han comparado con el algoritmo de búsqueda dispersa.

Se prosiguió a investigar de lleno el algoritmo de búsqueda dispersa, en la cual se enfatizó en las recomendaciones para su correcta implementación, como también las distintas estrategias para utilizar en los métodos del algoritmo y de esta forma generar una propuesta que se adapte a las necesidades del modelo de coloración de gráficas suaves y a su vez sea confiable, eficaz y eficiente. Para garantizar estas características se investigaron instancias que se pudieran utilizar para probar el algoritmo propuesto, pero primero se concluyó que se propondrían instancias propias para de esta menara tener un control y garantizar que se implementaran las mejores estrategias para cada método del algoritmo, al obtener el algoritmo con las mejores estrategias se comparó y probo con instancias para el problema de coloración robusta las cuales ya habían sido resultas con anterioridad con el algoritmo de búsqueda dispersa y otros algoritmo que se investigaron, con el fin de garantizar la confiabilidad, eficiencia y eficacia del algoritmo propuesto.

Capítulo 1
Marco de referencia

1.1. Problemas de coloración de gráficas

Los problemas de coloración de gráficas son uno de los problemas más utilizados de optimización combinatoria debido a su flexibilidad dado que pueden dar solución a distintos tipos de problema como: reconocimiento de patrones (Lara Velázquez et al. 2015b), calendarización, asignación de recurso (Lara Velázquez et al. 2010), coloración de mapas, entre otros, como también su capacidad de dar soluciones de alta calidad. A continuación, se expondrán de manera breve algunos de los problemas de coloración de gráficas y se pondré énfasis el modelo de coloración de gráficas suaves, el cual compete a esta investigación, como también se explicare de una manera un poco más amplia el problema de coloración robusta.

1.1.1. Coloración de gráficas mínima

En este problema se tiene una gráfica G con "n" vértices, los vértices unidos por una misma arista no pueden tener el mismo color k, se busca utilizar la menor cantidad de colores para colorear todos los vértices de la gráfica y aquellos con el mismo color forman una clase, que es lo mismo que una coloración que no use más colores del número cromático (Diestel 2000). Este problema se aplica en la coloración de mapas, de tal manera que el mapa contenga el menor número de colores y en calendarización, para evitar el conflicto entre varios usuarios y un recurso único.

1.1.2. Coloración Equitativa

La coloración equitativa busca colorear los vértices "n" de una gráfica G, de tal manera que los vértices unidos por una arista tengan distinto color k. Lo que caracteriza a la coloración equitativa es que el número de vértices entre cualquier clase de color será diferente a lo mas en un vértice. Este problema de coloración se utiliza en la calendarización de rutas, en el cual se busca tener el mayor número de rutas posibles en un momento determinado (Lih 2013).

1.1.3. Coloración de gráficas débiles

En este problema se busca colorear los vértices "n" de una gráfica G de tal manera que cualquier vértice que esté conectado con más de un vértice, al menos uno de estos vértices tiene que ser de color k distinto, esto con el objetivo de crear una gráfica con el mínimo número de Incompatibilidades (Kuhn 2009). Este modelo, se colorea una gráfica con menos colores que el número cromático.

1.1.4. Coloración robusta

Buscando en la literatura, el problema de coloración robusta es uno de los problemas de coloración más utilizados junto con el problema de coloración de gráficas tradicional. Es un problema con complejidad NP-Duro y es una generalización del problema de coloración mínima. Consiste en 2 gráficas, la primera es del tipo de coloración mínima la cual es una restricción del problema, y la segunda es una gráfica complementaria a la primera y tiene penalizaciones en sus aristas. El objetivo de la coloración robusta es encontrar una solución los más estable posible, por lo que se busca minimizar la suma de los valores de las aristas en la segunda gráfica, cuyos vértices tienen el mismo color, manteniendo válidas las incompatibilidades (Ramírez Rodríguez 2001).

Sea una gráfica simple G, con conjuntos de vértices y aristas denotados por V (G) y E(G), respectivamente, y |V (G)| = n. Se define una coloración válida de G como una aplicación C: $V(G) \rightarrow \{1, 2, ..., n\}$ que identifica a C (i) como el color del vértice i, de forma tal que dos vértices adyacentes no tengan el mismo color, es decir, $(i, j) \in E(G) \Rightarrow C(i) \neq C(j)$.

Una coloración válida de una gráfica G con k colores define una partición del conjunto de vértices en subconjuntos V1, V2, ..., Vk, donde Vj denota al conjunto de vértices que tienen asignado el color j. Claramente, cada Vj es un conjunto independiente, al cual se le llama clase de color j o clase cromática j. Se dice que una k-coloración válida de G es una coloración válida que no utiliza más de k colores, y una gráfica es k-coloreable si admite una k-coloración válida. Al mínimo valor k tal que G es k-coloreable se le llama número cromático de la gráfica y se denota por $\chi(G)$ (Lara Velázquez et al. 2012).

En una gráfica G, el Problema de Coloración Mínima busca una coloración válida de G que no utilice más de $\chi(G)$ colores.

El concepto de rigidez de una k-coloración, el cual establece que dadas dos gráficas complementarias G y $\bar{G}$, y una función de peso p: $E(\bar{G}) \rightarrow R$, la cual asigna una penalización a

cada arista de $\bar{G}$, la rigidez de una k-coloración c^k de G, denotada R(c^k) es la suma de los pesos de las aristas de $\bar{G}$ que unen vértices de la misma clase cromática (Rámirez y Yáñes 2003), esto es:

$$R(C^k) = \sum_{(i,j)\in E(\bar{G}), C^k(i) = C^k(j)} pij$$

Para convertir este problema a un modelo en gráficas suaves, dada la función objetivo:

$$\min z = \sum_{(i,j)\in E} pijyij.$$

Se re-penalizan las aristas de la gráfica G = (V, E) de la siguiente manera:

$$\begin{matrix} n^2 & si & i,j \in E \\ vij & si & i,j \in \bar{E} \end{matrix}$$

Penalizando las aristas de la gráfica original con un valor igual a n^2 se garantiza que al encontrar una coloración con una dureza menor que n^2, se tiene una coloración válida mínima: Esta gráfica es válida en el problema de coloración mínima porque se han excluido todas las aristas de la gráfica E, dicha solución tiene un costo menor a n^2 ya que todas las penalizaciones son positivas y estrictamente menores que uno. Para la gráfica complementaria se conservan las penalizaciones originales. El número cromático es aquel número de colores donde se obtiene por primera vez una coloración menor que n^2 (Lara Velázquez et al. 2015). Para más colores que el número cromático, cualquier coloración óptima de gráficas suaves es una coloración óptima del problema de coloración robusta con esa cantidad específica de colores. La resiliencia es un parámetro que nos permite encontrar una cantidad adecuada de colores. Al incrementar un color a la coloración de la gráfica y tener un aumento en la resiliencia, se ha encontrado un número adecuado de clases para clasificar.

1.2. Coloración de Gráficas Suaves

El problema de coloración de gráficas suaves es una generalización del problema de coloración de robusta (Diestel 2000), consiste en dada una gráfica completa con "n" vértices y penalizaciones en cada arista (Fig. 1.1), encontrar el valor mínimo de la dureza, es decir la suma de las aristas cuyos vértices estén pintados con el mismo color. La coloración de gráficas suaves ha demostrado que puede resolver otros problemas tales como coloración mínima, coloración equitativa, entre otros.

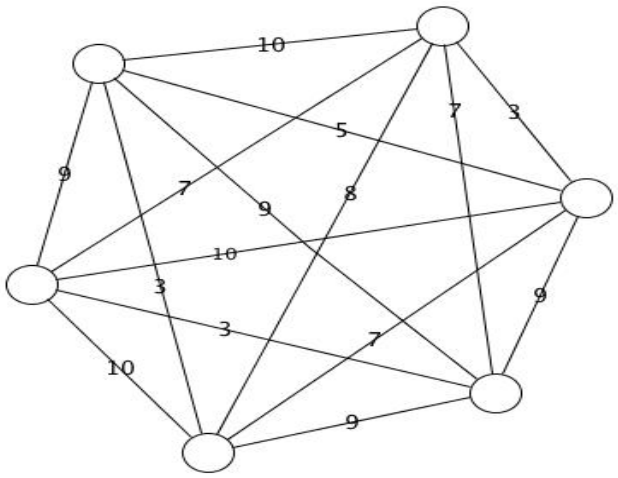

Figura 1.1. Gráfica completa con penalizaciones en cada arista.

El problema de coloración de gráficas suaves se ha utilizado para reconocimiento de patrones (Lara Velázquez et al. 2015b) y se ha utilizado para una comparación lingüística (Lara Velázquez et al. 2015a).

1.2.1. Modelo

El problema se puede definir de la siguiente forma: *G* es una gráfica completa no dirigida, formada por un conjunto de vértices *V = {1, 2, …, n}* y un conjunto de aristas E que contiene todas las aristas *(i, j)* (Lara Velázquez et al. 2015).

$$G = (V, E); \mid V \mid = n; \mid E \mid = n(n-1)\ 2.$$

Se define una penalización en cada arista (i, j), denotada por pij tal que:

$$p_{ij} \geq 0,\ \forall (i,j) \in E.$$

Una función de coloración en los vértices de G = (V, E) se define como:

$$C^k: 1, 2, ..., n \rightarrow 1, 2, ..., k,$$

Donde k es el número total de colores 1 < k < n que identifica C(i) como el color en el vértice i. Para una coloración C^k en una gráfica la *función de dureza* de la coloración C^k está dada por:

$$H(C^k) = \sum_{(i,j)\in E, C^k(i)=C^k(j)} p_{ij}$$

El objetivo del problema de CGS es encontrar la coloración C_{op}^k que minimice la dureza $H(C_{op}^k)$.

1.2.2. Solidez de una coloración

k = Número de colores en una gráfica completa con *n* vértices.
El número promedio de vértices pintado con el mismo color es *m = n/k*
El número de aristas compartidas por *m* vértices son las combinaciones *C(m , 2) = m(m-1)/2*

Por lo anterior el número promedio de penalizaciones incluídas en la función de dureza debe ser proporcional al número promedio de vértices pintados con el mismo color multiplicado por el número de colores *km(m-1)/2* (Lara Velázquez et al. 2015).

La función de solides de una coloración es la dureza de un gráfico entre el número medio de aristas que contribuyen a la dureza.

$$S(C_{op}^k) = \frac{H(C_{op}^k)}{km(m-1)/2} = \frac{2H(C_{op}^k)}{k\frac{n}{k}(\frac{n}{k}-1)} = \frac{2kH(C_{op}^k)}{n(n-k)}$$

1.2.3. Resiliencia de una coloración

La resiliencia de una coloración C^k se define como el porcentaje en que disminuye la solidez de una coloración con k – 1 colores con respecto a la realizada con k de ellos y se puede expresar como sigue (Lara Velázquez et al. 2015):

$$R(C_{op}^k) = \frac{S(C_{op}^{k-1}) - S(C_{op}^k)}{S(C_{op}^k)}$$

La resiliencia es un parámetro que nos permite encontrar una cantidad adecuada de colores. Al incrementarse colores a la coloración de la gráfica de uno en uno y tener un aumento en la resiliencia significa que es un número adecuado de clases para clasificar.

1.2.4. Modelo Binario para el problema de coloración de gráficas suaves

Dada una gráfica no dirigida *G = (V, E) con | V | = n y | E | = n(n−1)/.* se desea obtener una coloración C_{op}^{k} usando k colores. Para resolver este problema definimos el modelo de programación binaria siguiente (Lara Velázquez et al 2015):

$$\min z = \sum_{(i,j)\in E} p_{ij} y_{ij}$$

sujeto a:

$$\sum_{l=1}^{k} x_{il} = 1 \; \forall \; i \; \in \; \{1, \dots, N\}$$

$$x_{il} + \; x_{jl} - 1 \; \leq y_{ij} \; \forall \; (i,j) \in E \; y \; \forall \; l \; \in \{1, \dots, k\}$$

$$x_{il} \; \in \{0,1\} \; \forall_i \in E \; y \; \forall \; l \; \in \{1, \dots, k\} \qquad (1)$$

$$y_i \in \{0,1\} \quad \forall (i,j) \in E$$

donde:

$$x_{il} = \begin{cases} 1 & si \; C(i) = l \; \forall \; i \; \in \{1, \dots, n\} \; y \; \forall \; l \in \{1, \dots, k\} \\ 0 & en \; otro \; caso \end{cases}$$

$$y_{ij} = \begin{cases} 1 si \; para \; alguna \; l \; \in \{1, \dots, k\} \; se \; cumple \; x_{il} = \; x_{jl} = 1 \\ 0 \qquad\qquad en \; otro \; caso \end{cases}$$

El conjunto de ecuaciones del sistema (1) nos asegura que cada vértice tenga solamente asignado un color, el conjunto de desigualdades hace que y_{ij} sea igual a 1 si los vértices *i* y *j* están coloreados con el mismo color, y 0 si los colores de los vértices i y j son distintos. En el primer caso se activa la penalización p_{ij} en la función objetivo *z*. La solución al problema (1) obtiene la coloración mínima C_{op}^{k} tomando $C_{op}^{k}(i) = l$ si $x_{il} = 1$; donde $x_{il} \; \forall \; i \; \in \{1, \dots, n\}$ y $\forall \; l \; \in \{1, \dots, k\}$, son los valores de las variables de decisión obtenidas en la solución optima del modelo (1) y el valor mínimo z^* de la función objetivo es igual a $H(C_{op}^{k})$.

El modelo (1) tiene *nk* variables x_{il} y *n(n−1)/2* variables y_{ij} por lo que el número total de variables binarias del modelo es *nk + n(n−1)/2*. Por otro lado el número de ecuaciones es igual a *n* y el número de desigualdades es igual a *kn(n−1)/2,* por lo tanto, el número de restricciones es igual a: *nk + n(n−1)/2*.

1.3. Aplicaciones del modelo de coloración de gráficas suaves

1.3.1. Reconocimiento de patrones

El reconocimiento de patrones es el proceso en el cual, dentro de una población, se puede agrupar a ciertos individuos dependiendo de sus características Para el caso de la inteligencia artificial el reconocimiento de patrones es el estudio de cómo una maquina puede detectar características mediante la observación de su ambiente y tomar decisiones acertadas con esas características para formar una clasificación (Jain, Duin, y Mao 2000). Es ampliamente utilizado ya sea para clasificar nuevas especies, diagnosticar enfermedades, reconocimiento escrito y hablado, entre otras aplicaciones

Usualmente en lo problemas de reconocimiento de patrones se tiene las clasificaciones definidas previamente, el algoritmo meramente se encarga de a cada objeto asignarlo a una de estas clasificaciones previamente definidas (Lara Velázquez et al. 2015b), el algoritmo no es capaz de hacer por sí mismo una clasificación propia, por lo cual es necesario una investigación previa para definir un número de clasificaciones adecuado, de manera que se puedan agrupar nuestros objetos de la manera más óptima.

Aterrizando la coloración de gráficas suaves al problema de reconocimiento de patrones, se puede ver de la siguiente forma: cada color representa una clasificación, cada vértice representa un objeto y la ponderación de las aristas es un valor obtenido por la función objetivo que nos indica de manera numérica que tanto dos vértices tienen en cierto grado las mismas características. Todo esto se puede gracias a la cualidad de que el problema de coloración permite, a diferencia de otros tipos de coloración, que los vértices unidos por una arista tengan el mismo color.

1.3.2. Calendarización

Los problemas de calendarización se pueden interpretar de distintas formas. Como administración de recursos es básicamente un mecanismo o política usada para el manejo eficiente y efectivo del acceso a los recursos y el uso de los mismos recursos por sus varios consumidores, al igual que el caso anterior su objetivo es contar con una asignación de recurso que minimice el tiempo de espera de acceso al recurso por los consumidores (Sánchez s/f).

De igual manera se pude ver como un conjunto de trabajos debe ser procesado secuencialmente en una serie de etapas, pero no todos los trabajos tienen que pasar por todas las etapas, el objetivo es tener una asignación de tareas que minimice el tiempo de terminación. Es un problema de optimización discreta, por lo que se describe como un modelo combinatorio de programación, que minimice el tiempo del proceso, evitando la pelea por recursos entre cada uno de los trabajos en las distintas etapas del proceso (Rojas y Lama 2010). Debido a que su tiempo de solución con el incremento de recurso y consumidores crece de manera exponencial es un problema NP-Completo, así que es necesario utilizar metaheurísticas para su solución. La calendarización tiene un alto campo para su aplicación como es diseño de circuitos, logística, programación concurrente, entre otros.

Para el caso de coloración de gráficas suaves se puede, observar a cada color como un tiempo, a cada vértice como un recurso y cada arista como un consumidor, en este caso se busca la cantidad mínima de colores, de tal manera que se evite la lucha por recursos entre consumidores.

1.4. Ejemplo del modelo de coloración de gráficas suaves

Para ejemplificar de manera intuitiva los conceptos básicos del modelo de coloración de gráficas suaves, tomaremos como una gráfica completa la Fig.1.2, esto quiere decir que todos los vértices están conectados entre sí por aristas con penalizaciones, el valor de estas penalizaciones están calculadas por la fórmula de distancia entre dos vértices y los vértices se encuentra en las posiciones siguientes: (0,0), (0,2), (1,0), (1,2), (0,9), (0,11), (1,9), (1,11), (10,0), (10,2), (11,0), (11,2), (10,9), (10,11), (11,9) y (11, 11).

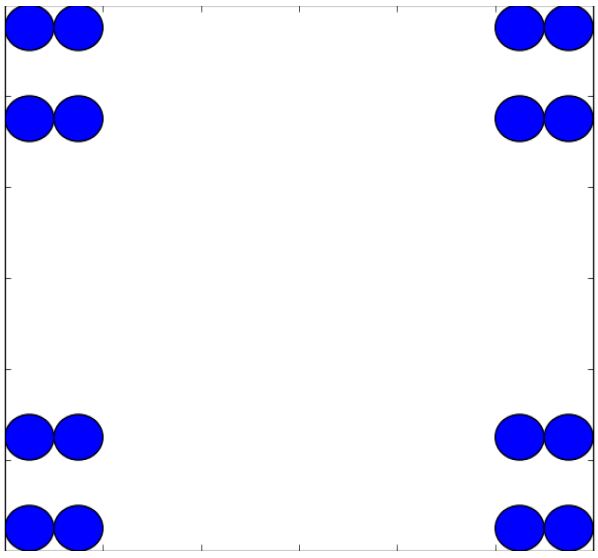

Figura 1.2. Coloración con un solo color.

Como se pude observar en la Fig.1.2, el modelo de coloración de gráficas suaves comienza pintando toda la gráfica con un solo color, con esto se prosigue a ser los cálculos de dureza y solidez respectivamente y para el caso de la resiliencia al ser una coloración de un solo color es igual a cero debido a que la fórmula de resiliencia necesita de una solidez actual y una anterior algo que no es posible al ser la primera coloración, los resultados obtenidos se encuentran en la Tabla 1.1.

TABLA 1.1
RESULTADOS PARA UN COLOR

Colores	Dureza	Solidez	Resiliencia
1	1086.081404	9.050678	0

Al incrementar los colores como se ve en la Fig.1.3, a dos colores, podemos notar una disminución en la dureza y en la solidez en la Tabla 1.2, como también podemos observar el primer resultado de resiliencia, dado que ya contamos con una solidez anterior y una actual.

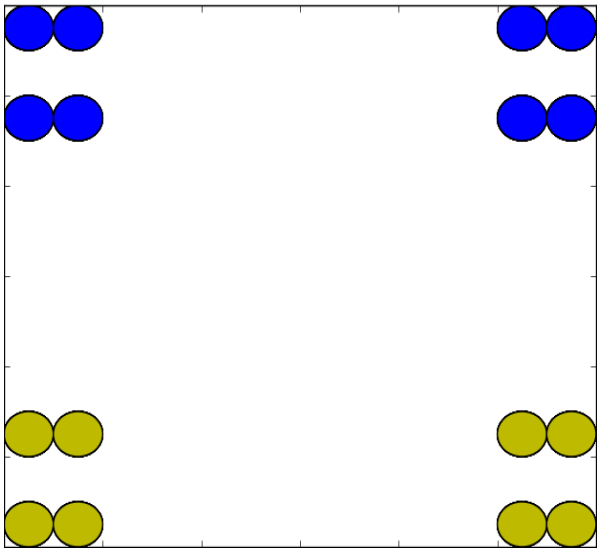

Figura 1.3. Coloración con 2 colores.

TABLA 1.2
RESULTADOS PARA DOS COLORES

Colores	Dureza	Solidez	Resiliencia
1	1086.081404	9.050678	0
2	330.79734	5.907095	0.532171

Como se mencionó en la descripción del modelo de coloración de gráficas suaves, su objetivo es disminuir la dureza a la vez que encontrar la mejor coloración posible, por lo cual, tanto el valor de dureza como el de resiliencia son sumamente importantes. En la Fig.1.4, se puede observar la coloración con tres colores de nuestro ejemplo.

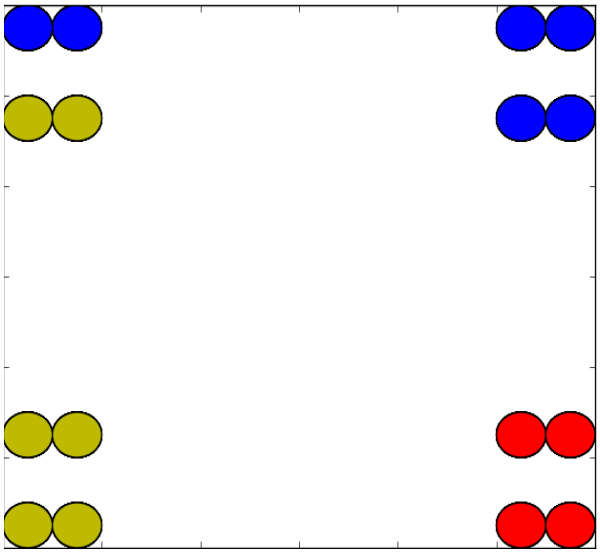

Figura 1.4. Coloración con tres colores.

Al incrementar a tres colores nuestra coloración como podremos ver en la Tabla 1.3, nuestra dureza disminuye al igual que la solidez y la resiliencia, pero como se comentó anteriormente, en el caso de la resiliencia una disminución en su valor indica una peor coloración por lo cual la agrupación de tres colores de nuestra gráfica es peor en comparación de la coloración con dos colores.

TABLA 1.3
RESULTADOS PARA DOS COLORES

Colores	Dureza	Solidez	Resiliencia
1	1086.081404	9.050678	0
2	330.79734	5.907095	0.532171
3	178.465276	5.148037	0.147446

Al agregar un color más para pintar la gráfica, obtenemos la coloración que se puede apreciar en la Fig.1.5, con esta coloración podemos constatar cómo es observable en la Tabla 1.4 que hemos encontrado el número de colores idóneo como la dureza menor, esto se debe a que los valores de dureza y solidez son los menores en comparación a las coloraciones anteriores y por su parte la resiliencia incremento en comparación a la anterior y también su valor es superior

que el valor que se obtuvo para un coloración de dos colores. Por lo que se puede concluir que la clasificación con cuatro colores para este ejemplo es la mejor ya que cuenta con la menor dureza y la mayor resiliencia, también se pude concluir que la clasificación de tres colores es la peor, al contar con el valor más bajo de resiliencia.

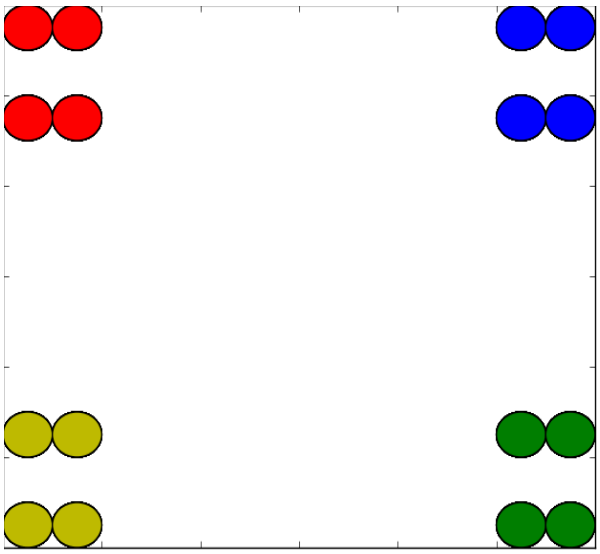

Figura 1.5. Coloración con 4 colores.

TABLA 1.4
RESULTADOS PARA DOS COLORES

Colores	Dureza	Solidez	Resiliencia
1	1086.081404	9.050678	0
2	330.79734	5.907095	0.532171
3	178.465276	5.148037	0.147446
4	41.888544	1.745356	1.949563

1.5. Metaheurísticas

Como se mencionó anteriormente los problemas de coloración de gráficas tiene una complejidad mínima de NP-Completo y en el caso del problema de coloración de gráficas suaves es de NP-Difícil, por lo cual para instancias mayores a 20 vértices es necesario el uso de una metaheurística para obtener una solución.

Hasta la fecha no existe una clasificación definitiva de las heurísticas y metaheurísticas, por lo cual se optó por utilizar la siguiente clasificación para las metaheurísticas más utilizadas para el problema de coloración de gráficas (Baghel, Agrawal, y Silakari 2013) (Musliu y Schwengerer 2013), las metaheurísticas clásicas tales como: Búsqueda Tabú, GRASP, Algoritmos Genéticos,

Recosido Simulado, Búsqueda de Vecindarios Variables. O como metaheurísticas Avanzadas como: Algoritmos Bio-Inspirados y Búsqueda Dispersa.

1.5.1. Búsqueda Tabú

El algoritmo de búsqueda tabú es una herramienta para resolver problemas de optimización combinatoria, fue creado por Fred W. Glover en 1986 forjando sus bases en 1989 (Glover 1989), permite a una heurística de búsqueda local, ir más allá de los límites del optimo local de cual otra forma seria incapaz de salir. Es una metaheurística que se ha utilizado dando soluciones de alta calidad para problemas clásicos de optimización como son: calendarización, coloración de gráficas equitativa (Lai, Hao, y Glover 2015), redes neuronales entre otros.

Este algoritmo tiene una característica principal, la cuál es que cuenta con una memoria adaptable para de esta forma encontrar una solución de manera eficiente, la memoria es adaptable porque dependiendo del problema se puede implementar una memoria a corto plazo o una a largo plazo.

La búsqueda tabú inicia como una búsqueda local o búsqueda de vecindades variables, se parte de una solución inicial y en cada iteración se busca una solución vecina a la actual y se continua hasta que se cumple un requisito de paro. Lo que cambia en búsqueda tabú es que con cada repetición del algoritmo se busca minimizar una función objetivo, por lo cual la solución que se escoja tiene que ser una solución que sea mejor que la que se tenía anteriormente, en este proceso entra la estrategia de la memoria, si se quiere tener una memoria a corto plazo o una memoria a largo plazo, estas memorias por lo general almacenan las mejores soluciones que sean ido obteniendo a lo largo del algoritmo y se puede utilizar la memoria de tal manera que limite el número de veces que una solución es aceptada ya sea por un cierto número de iteraciones o por el resto del algoritmo, también se puede utilizar la memoria para si se llega un punto en el que el algoritmo ya no encuentra una solución que minimice la función objetivo, pero el criterio de paro no se ha cumplido (Se esté en un óptimo local) se busque en la memoria una solución igual o mejor que la que se tiene y a partir de esta solución se genere un nuevo "camino", de tal forma que no se tomen las soluciones anteriores (Glover y Laguna 2007).

1.5.2. GRASP

Por sus siglas en inglés Greedy Randomized Adaptive Search Procedure es algoritmo introducido en 1989 por Feo and Resende, es una metaheurística muy utilizada debido a su gran velocidad para obtener buenas soluciones para problemas de optimización combinatoria. Es un algoritmo que tiene como característica tener 2 fases principales, una fase de construcción, seguida de una fase de mejora (Feo y Resende 1995).

A la primera fase se le conoce como Greedy Randomized Adaptive. En esta fase se genera una población de buenas soluciones mediante un algoritmo de tipo glotón (greedy), pero en lugar de escoger la mejor solución directamente se genera una lista con las mejores soluciones debido a que la característica del algoritmo glotón es escoger la que en su momento parece la mejor solución, lo que significa que escoge una solución que es un óptimo local a la espera que esta solución se convierta en un óptimo global, pero esto no es completamente seguro, por lo cual a las soluciones generadas en la lista de mejores soluciones se le aplica la segunda fase: la fase de mejora, la cual consiste en aplicar a cada uno o a cierto número de soluciones un método de búsqueda local con el objetivo de convertir las soluciones de optimo local en óptimos globales.

1.5.3. Algoritmos Genéticos

John Holland a finales de los 60s y principios de los 70s desarrolla el algoritmo genético basado en la teoría de Darwin sobre la adaptación natural (Holland 1992), Holland desarrolló este algoritmo de tal manera que su ejecución se asemejara lo más posible a el proceso evolutivo. Sus métodos son altamente influenciados con características propias de la adaptación natural.

Este algoritmo se caracteriza por tener un función de aptitud (función objetivo) con la cual las soluciones serán evaluadas en cada paso del algoritmo, primero se inicia con una población inicial (soluciones iniciales) a partir de esta población se inicia un ciclo que termina hasta que se cumpla un criterio de paro, las soluciones están representadas en formas de cromosomas por los que están compuestas de manera binaria por 0 y 1. Dentro de este ciclo se efectúan los métodos fundamentales de este algoritmo, los cuales son (de los Cobos Silva et al. 2010):

1. **Selección:** En este paso se utiliza normalmente el método de la ruleta para de esta forma simular el proceso de evolución, en el cuál los seres más aptos continúan, sin eliminar por completo a los seres menos aptos. Se encuentra una probabilidad de aptitud para cada individuo (solución) de la población, se divide entre la aptitud total y con esta nueva probabilidad acumulada mediante la generación de números aleatorias se va seleccionando la nueva población.
2. **Cruzamiento:** Aquí se simula el proceso de descendencia, en el que dos soluciones dan origen a una nueva solución que comparte características de sus dos padres, esto se realiza a través del uso de un random que determina si un par de cromosomas se combinaran, si si se combinan se utiliza otro random y a partir de este número la cadena de cromosomas delantera de una solución se junta con la cadena de cromosomas de otra solución, de esta manera hasta generar una nueva población.
3. **Mutación:** Este método busca emular el proceso natural en el cuál algunos individuos presentan características únicas, para esto se utiliza un random con el cual se decidirá si

se muta cierto individuo de la población, si un cromosoma es seleccionado se utiliza un random para cada gen que determina si muta o se mantiene igual

4. **Evaluación:** Realiza el proceso de evaluar todos los individuos de la población con la función de aptitud

Después de este paso el algoritmo se detiene o continúa dependiendo del criterio de paro, se tiene algunas estrategias para mejorar su desempeño, como el elitismo en el cual en cada iteración se guarda al mejor individuo y se compara con el mejor individuo de la nueva generación, si el mejor individuo es mejor que el anterior individuo ahora se guarda esta solución, de lo contrario se remplaza el peor individuo de esta generación por el mejor individuo de la generación pasada (Whitley 1994).

1.5.4. Búsqueda de Vecindarios Variables

Es un algoritmo desarrollado por Mladenović y Hansen en 1997 con el fin de resolver problemas de optimización combinatoria, en el cual las soluciones son propensas a quedarse en óptimos locales y no se pueda alcanzar un óptimo global.

La idea es ir cambiando las vecindades de manera controlada al momento de realizar la búsqueda, en cada iteración se trabaja con cierta vecindad en la cual se localiza un punto ya sea de manera aleatoria o con cierto criterio, y a este punto se le aplica búsqueda local para hallar el óptimo local, si este optimo local es mejor solución que la que se tiene al momento se gurda y se pasa al siguiente vecindario, si la solución no es mejor, no hay remplazo y se pasa la siguiente vecindario, de esta forma se continua iterando hasta que se cumpla el criterio de paro (Mladenović y Hansen 1997).

1.5.5. Recocido Simulado

Es una metaheurística aproxima a una solución óptima dentro una población grande se enfoca en problemas de optimización discretos, fue propuesto en 1983 por Kirkpatrick, Gellat y Vecchi basándose en el recocido de un sólido (Kirkpatrick, Gelatt, y Vecchi 1983). Para el caso de recocido simulado las soluciones se pueden ver como los estados de un sólido y los valores de la función objetivo para cada solución como los niveles de energía de un sólido, se tiene un valor de temperatura que funciona como un parámetro de control y se pude interpretar como el proceso de enfriamiento propio de un sólido.

Para explicar los conceptos del algoritmo de recocido simulado, lo compararemos con el algoritmo de búsqueda local. El algoritmo de búsqueda local parte de una solución inicial, dicha solución se va mejorando con cada iteración del algoritmo, este proceso de mejora continua

hasta que el algoritmo no encuentra una mejor solución, la solución obtenida al finalizar el algoritmo no representa necesariamente un óptimo global, y en la mayoría de los casos para problemas de mayor complejidad la solución termina estacándose en un óptimo local, por lo cual, el algoritmo de recocido simulado para impedir que una solución se estanque en un óptimo local permite que el algoritmo tomo ciertas libertades y en ciertos momentos escoja en lugar de la mejor solución una de menor calidad, claro esto debe ser de manera controlada para evitar que el algoritmo se desvirtúe (Press 2007), para lograr este control el algoritmo de recocido simulado utiliza una fórmula probabilística, la cual con el paso de las iteraciones ira disminuyendo la probabilidad de tomar una solución de menor calidad y tomar solo soluciones que mejoren a la actual, la fórmula es la siguiente:

$$P_c\{aceptar\ j\} = \begin{cases} 1 & si\ \ f(j) \leq f(i) \\ exp\left(\dfrac{f(i) - f(j)}{c}\right) & si\ \ f(j) > f(i) \end{cases}$$

Las iteraciones continúan de manera que con cada iteración la temperatura decrece hasta llegar a un estado "frío" o estable, en el cual la solución que se tiene sería lo más cercana posible al optimo global. Cabe destacar el algoritmo de recocido simulado es un algoritmo que presenta soluciones de alta calidad en un tiempo de ejecución aceptable, como tal el algoritmo es fácil de implementar, pero es complicado implementarlo de tal manera que de los resultados de alta calidad esperados y esto se debe que para cada problema su implementación es distinta y los valores de temperatura, de control de paro y otros puntos finos son difíciles de obtener de manera obvia, por lo cual para hallar estos valores es necesario realizar un proceso amplio de experimentación y de esta manera tener un algoritmo robusto.

1.5.6. Algoritmos Bio-Inspirados

Son algoritmos inspirados en la adaptación natural, cabe de estacar que en la actualidad podemos encontramos con distintos tipos de algoritmos inspirados en la naturaleza por lo cual no hay un algoritmo que los pueda englobar. Algunos de los algoritmos bio-inspirados más conocidos son: colonia de hormigas (Bessedik et al. 2005), enjambre de abejas (Tomar et al. 2013), inspirado en el sonar de los murciélagos (Djelloul, Sabba, y Chikhi 2014), partículas (Bouzidi y Riffi 2014), polinización (Bensouyad y Saidouni 2015), entre otros.

Como se puede observar estos algoritmos están inspirado en el comportamiento de la naturaleza más que en su evolución y esta es la principal "diferencia" con los algoritmos genéticos, cabe decir que su uso no solo se centra en problemas de optimización sino que también se han utilizado para el estudio de ciertos seres vivos como es el caso del enjambre de abejas y el polinización, que han sido utilizados en ciertas partes de Europa como una medida de

prevención ante la escasez de abejas y la falta polinización debido a la misma, y estos algoritmos se han probado con robots para simular el proceso de polinización de las abejas.

1.6. Búsqueda Dispersa

Como se mencionó anteriormente la búsqueda dispersa es considerada un algoritmo evolutivo debido a que con cada iteración obtiene nuevas soluciones a partir de las soluciones previamente generadas. La diferencia radica en que la búsqueda dispersa trabaja con una población pequeña de soluciones conocida como conjunto de referencia, mientras que los algoritmos evolutivos trabajan con poblaciones grandes. La metodología de búsqueda dispersa es flexible de tal manera que cada uno de su métodos puede ser implementado de distinta forma y con distintos grados de complejidad, con el objetivo de que el algoritmo se adecue a la necesidades de cada implementación de tal manera que la efectividad del algoritmo no se vea afectada (Martí, Laguna, y Glover 2006).

El algoritmo de Búsqueda Dispersa (Scatter Search) consta de 5 métodos principales:

1. **Diversification Generation Method (Generador de Soluciones Diversas).**

Este método consiste en generar una población de soluciones diversas, conocida como *P* que da inicio al algoritmo. En la literatura se recomienda que el tamaño del conjunto de soluciones diversas sea al menos 10 veces el tamaño del conjunto de referencia *b* que se quiere obtener (Martí et al. 2011) o que sea mayor a 100 elementos.

Para generas nuestra población de soluciones diversas no hay un método establecido, depende del problema al cual se quiera dar solución, lo único que se tiene que tener en mente es respetar que el método a usar de buenas soluciones pero también diversa para lograr este objetivo se utiliza algunas estrategias para tener un Diversification Generation Method de calidad, como son el uso de Memoria y utilizar un conjunto de soluciones amplio (Laguna y Armentano 2005). Esto se realiza con el fin de tener un mayor número de soluciones diversas.

Estas dos estrategias van de la mano ya que para el uso adecuado de memoria se tiene que utilizar un grupo grande de soluciones para de esta forma obtener el mejor resultado. El uso de memoria se asemeja a la búsqueda Tabú, manteniendo un control de cuantas veces se permite que la misma solución forme parte de nuestra población de búsqueda dispersa, o que definitivamente no se permita soluciones idénticas en la población *P*.

2. **Improvement Method (Método de Mejora)**:

En este método se utiliza una heurística o metaheurística para mejorar las soluciones obtenidas por el método generador de soluciones diversas como también, para las soluciones que obtiene el método de combinación de soluciones. Típicamente se utiliza el método de búsqueda local o inclusive el método de búsqueda Tabú.

Para este método se pueden utilizar dos metaheurísticas o heurísticas y de esta manera intercalar su uso o usar un método por un cierto número de iteraciones y luego otro por otro cierto número de iteraciones para generar mejores soluciones y mantener el criterio de diversidad.

3. **Reference Set Update Method (Actualización del Conjunto de Referencia):**

Método para crear y actualizar el conjunto de referencias conocido como *RefSet*, el conjunto de referencias esta ordenado de la mejor a la peor solución respecto a su calidad y su tamaño debe ser pequeño, recomendablemente su tamaño no debe ser mayor de 20 soluciones. A continuación, se describen sus dos principales funciones en el algoritmo estándar de búsqueda dispersa (Martí, Laguna, y Glover 2006):

a. *Creación*. Se inicializa el conjunto referencia con *b/2* mejores soluciones de *P*, la proporción *b/2* restante se obtiene del conjunto de soluciones diversas con el criterio de maximizar la diversidad de soluciones sin importar su calidad, por lo cual puede ser llenado con las peores soluciones del conjunto de referencia o con soluciones aleatoria que no estén ya en el *RefSet*.

b. *Actualización*. Determina si en las soluciones obtenidas al finalizar cada iteración del algoritmo, hay soluciones que mejoren el conjunto de referencia, si hay soluciones mejores que las actuales en el *Refset* son insertadas de tal manera que se mantenga el orden de la mejor a la peor solución y las soluciones peores son eliminadas del conjunto de referencia para de esta manera tener un tamaño fijo del *RefSet,* las soluciones que no mejoren el *Refset* son descartadas. De lo contrario si no se hallan mejores soluciones que las existentes en el conjunto de referencia, se considera que no se hallaron nuevas soluciones y se da por concluido el algoritmo.

4. **Subset Generation Method (Método de Generación de Subconjuntos):**

Este método especifica la forma en que se seleccionan los subconjuntos para aplicarles el método de combinación de soluciones. El tamaño de los subconjuntos puede ser desde dos

elementos hasta el mismo número de elementos del conjunto de referencia, pero se aconseja que los subconjuntos no sean mayores a cuatro elementos, pues a partir de cinco elementos la calidad de las soluciones disminuye considerablemente.

Se recomienda agrupar los subconjuntos por parejas y tercias. También se pueden utilizar las siguientes dos estrategias: se generan primero subconjuntos de dos elementos, para posteriormente incrementar a tres elementos los subconjuntos agregando la mejor solución obtenida por la iteración pasada, después se generar subconjuntos de cuatro elementos agregando las mejores soluciones obtenidas por las iteraciones pasadas, esta estrategia podía continuar hasta que los subconjuntos fueran del tamaño de *RefSet,* pero, como se mencionó anteriormente la calidad de soluciones obtenidas después de subconjuntos de cuatro elementos disminuye por lo cual se a conseja llegar hasta cuatro elementos en el subconjunto y repetir de subconjuntos de dos elementos (Martí, Laguna, y Glover 2006). Otra estrategia que se puede utilizar es intercalar la generación de subconjuntos entre pares y tercias, ya sea una iteración y una o un cierto número de iteraciones se agrupan los subconjuntos por pares y después otro cierto número de iteraciones por tercias.

5. **Solution Combination Method (Método de Combinación de Soluciones):**

El método de combinación de soluciones, se encargar de combinar los subconjuntos de soluciones obtenidos por el método anterior con el fin de obtener una nueva solución que contenga los atributos de distintas soluciones y esto permee el valor de diversidad. En este método se puede utilizar una sola estrategia combinatoria o se pueden intercalar distintas estrategias principalmente se utilizan métodos combinatorios tomados de los algoritmos genéticos tales como: Crossover, partial inversion, scramble sublist, etc (Martí, Laguna, y Campos 2005).

Las soluciones obtenidas por este método son tratadas posteriormente por el método de mejora el cual buscara incrementar la calidad de las soluciones, las soluciones obtenidas por el Método de Mejora pueden ser agregadas de manera inmediata al conjunto de referencia si son mejores a las del conjunto de referencia o esperar a que se realicen todas las combinaciones y después decidir qué soluciones entran al conjunto de referencia.

Como se puede observar, el algoritmo de búsqueda dispersa tiene grandes cualidades gracias a la flexibilidad que tiene, con lo cual permite que cada uno de sus métodos sea adaptable de manera que sea lo más conveniente para el problema a resolver, como a su vez casi todos sus métodos tiene la propiedad de que pueden utilizar heurísticas o inclusive otras metaheurísticas para de esta forma mejor su desempeño y obtener soluciones de alta calidad en un tiempo de ejecución más que aceptable.

1.6.1. Aplicaciones de búsqueda dispersa

El algoritmo de búsqueda dispersa ha sido utilizado para solucionar distintos problemas de optimización discreta demostrando así su flexibilidad para adaptarse a distintos problemas dando soluciones de calidad en tiempos de ejecución rápidos. A continuación, se enlista algunos de los problemas en los que se ha utilizado:

- Problema de coloración robusta (Lara Velázquez et al. 2012).
- Problema del agente viajero (Martí, Laguna, y Campos 2005).
- Designing effective improvement methods for scatter search: an experimental study on global optimization (Hvattum et al. 2013).
- Hybrid Scatter Search for Integer Programming Problems (Fahim y Hedar 2014)

1.6.3. Algoritmo de búsqueda dispersa propuesto por Fred Glover

---

1. Comenzar con $P = \emptyset$. Utilizar el ***método de generación de soluciones diversa*** para construir soluciones

   De *i = 1 a TamP*

   Genere una solución x_i aleatoria

   Utilice el ***método de mejora*** en x_i para obtener una mejor solución x_i^*

   Si $x_i^* \notin P$, se incluye en *P* en otro caso, rechazar x_i^*
2. Utilizando el ***método de Actualización del Conjunto de Referencia*** Construir el ***conjunto de referencia*** *RefSet*
3. Ordenar el conjunto de referencia de la mejor a la peor solución
4. Hacer *NuevaSolución = TRUE*
5. ***Mientras*** (NuevaSolución)
6. Hacer *NuevaSolución = False*
7. Generar los subconjuntos de *RefSet* utilizando el ***método de Generación de Subconjuntos***.
6. ***Mientras*** (Queden subconjuntos sin examinar).
   7. Seleccionar un subconjunto y etiquetarlo como examinado.
   8. Aplicar el ***método de combinación de soluciones*** a los subconjuntos.
   9. Aplicar el ***método de mejora*** a cada solución obtenida por la combinación.
      10. Si es x_i^* es una mejor solución y no se encuentra en el *conjunto de referencia* llamar al ***método actualización del conjunto de referencia*** para agregar la solución al *RefSet* y Hacer *NuevaSolución = TRUE*.
   11. Reordenar RefSet de la mejor a la peor solución

Capítulo 2
Propuesta de Algoritmo

Propuesta de algoritmo

Como se comentó en los capítulos anteriores, el problema de coloración de gráficas suaves es un problema discreto; cuando se tienen casos mayores a 20 vértices es necesario el uso de metaheurísticas, para obtener soluciones de alta calidad en un tiempo de ejecución aceptable. Es por esto que se propone el uso del algoritmo de búsqueda dispersa esto debido a la flexibilidad que presentan sus métodos y los buenos resultados que obtiene este algoritmo.

A continuación, se describe los 5 métodos fundamentales del algoritmo de búsqueda dispersa para el problema de coloración de gráficas suave que se puede apreciar de manera gráfica a continuación (Fig. 2.1):

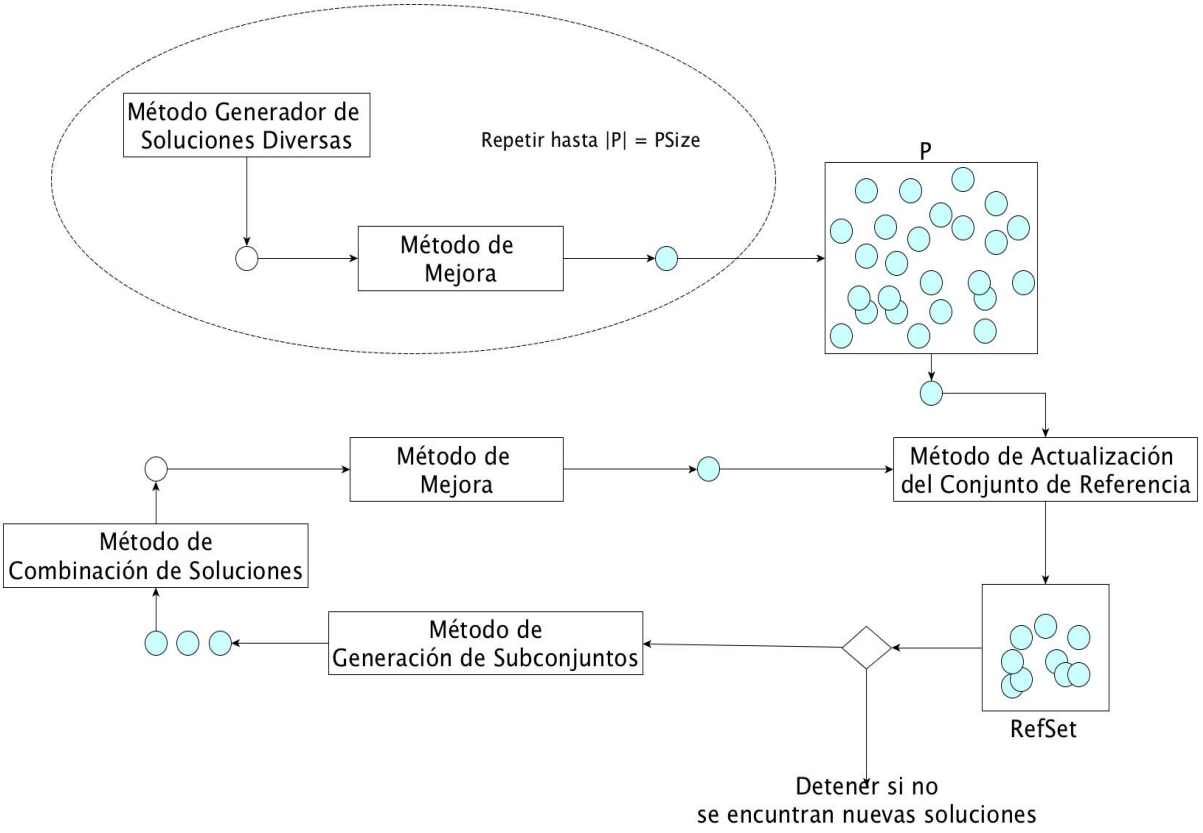

Figura 2.1. Algoritmo de búsqueda dispersa propuesto por Fred Glover en 1998.

2.1 Diversification Generation Method (Generador de Soluciones Diversas)

Este método se encarga de generar el conjunto *P* el cual es el conjunto inicial del algoritmo y está compuesto de soluciones diversas, A partir de este conjunto se procederá a obtener conjunto *b* al cual se le conoce como conjunto de referencia *RefSet* y es el conjunto con el cual se trabaja a lo largo del algoritmo. Conforme a las recomendaciones obtenidas de la literatura es aconsejable que el conjunto de referencia se no mayor a 20 soluciones y *P* sea al menos 10 veces el tamaño de *ReftSet* que se quiere obtener (Martí et al. 2011).

Para obtener soluciones diversas se propone el uso de memoria para de esta forma prevenir el uso de soluciones repetidas, se propone utilizar los siguientes métodos para obtener las soluciones:

a. Obtener una coloración de manera aleatoria, con una probabilidad de 1/*k* (*k* es el número máximo de colores), de esta manera se obtienen soluciones diversas que en promedio tienen un número equilibrado de colores, como se puede observar a continuación:

$$\begin{matrix} Color \\ Vértice \end{matrix} \quad \begin{matrix} \frac{1}{k} & \frac{1}{k} & \frac{1}{k} & \frac{1}{k} & \dots & \frac{1}{k} \\ 1 & 2 & 3 & 4 & & n \end{matrix}$$

b. Para el caso de coloración robusta se escoge un vértice al azar, para este vértice se busca en la matriz los vértices con los que intersecta y sus colores correspondientes, si ya todos los colores han sido ocupados, el vértice se pinta con un color aleatorio, de lo contrario se selecciona un color de maneara azarosa de los colores que no causen conflicto con los vértices intersectados, esto se realiza para todos los vértices

c. Tomando alguna de las soluciones obtenidas por alguno de los métodos anteriores, intercambiar los colores de tal forma que iniciando con el vértice $v_i = 1$ intercambié su color con el vértice v_n, a continuación, el $v_i = 2$ intercambié su color con el vértice v_{n-1} y así sucesivamente. A continuación, es ejemplificado de manera más clara:

$$\begin{matrix} Color \\ Vértice \end{matrix} \quad \begin{matrix} 1 & 6 & 4 & 1 & 2 & \dots & 5 & 3 \\ 1 & 2 & 3 & 4 & 5 & & n-1 & n \end{matrix} \longrightarrow \begin{matrix} 3 & 5 & 4 & 1 & 2 & \dots & 6 & 1 \\ 1 & 2 & 3 & 4 & 5 & & n-1 & n \end{matrix}$$

El uso de los métodos puede ser de manera intercalada o con un número determinado de soluciones para cada uno.

2.2 Improvement Method (Método de Mejora)

Las soluciones obtenidas por el método Generador de Soluciones Diversas, como las soluciones generadas por el método de combinación de soluciones, son procesadas por este método en busca de mejorar su calidad. Se propone utilizar las siguientes estrategias:

a. Utilizar el uso de una heurística o metaheurística para mejorar las soluciones, en el caso particular de nuestro algoritmo se propone utilizar la búsqueda local dado que se sugiere en la literatura el uso de estrategias sencillas, para no afectar los tiempos de ejecución (Hvattum et al. 2013). La búsqueda local implementa se caracteriza por escoger un vértice de manera aleatoria y a ese vértice pintarlo con un color azarosamente. Típicamente la búsqueda local termina cuando no se encuentra nuevas soluciones, pero para nuestro algoritmo el criterio de paro será un cierto número de iteraciones, con cada cambio se utiliza la dureza y la función de resiliencia $R\left(C_{op}^{k}\right)$ para determinar si la solución mejora y no es una solución igual, el funcionamiento de búsqueda local se puede observar a continuación:

$$Vertice\ Aleatorio = 4 \qquad \begin{matrix} Color \\ Vértice \end{matrix} \begin{matrix} 1 & 6 & 4 & 1 & 2 \\ 1 & 2 & 3 & 4 & 5 \end{matrix}, \dots, \begin{matrix} 5 & 3 \\ n-1 & n \end{matrix}$$

$$Color\ aleatorio = 8 \qquad \begin{matrix} Color \\ Vértice \end{matrix} \begin{matrix} 1 & 6 & 4 & 8 & 2 \\ 1 & 2 & 3 & 4 & 5 \end{matrix}, \dots, \begin{matrix} 5 & 3 \\ n-1 & n \end{matrix}$$

b. Para las instancias de coloración robustas analizadas en los capítulos posteriores se decidió utilizar la misma búsqueda local con la diferencia que para cada vértice aleatorio, se buscara en la matriz los vértices con los que intersecta y los colores con los que están pintados. Si el número total de colores ha sido utilizado el color se escoge de manera aleatoria, si no han sido utilizados todos los colores, se selecciona de manera azarosa un color que no cause conflicto con los vértices intersectados, esta estrategia es parecida a la del método generador de soluciones diversas con la diferencia de que se aplica para un color aleatorio el cual se puede repetir, en lugar de para cada vértice de la solución.

c. La búsqueda local se compara el capítulo de análisis de resultados contra una combinación de estrategias para de esta forma comparar su efectividad, las estrategias usadas son: la selección de dos vértices al azar y su cambio de color

de manera aleatoria y el intercambio de colores entre dos vértices seleccionados de manera aleatoria.

Selección de 2 vértices aleatoriamente y 2 colores al azar:

$$Vertice\ Aleatorio1 = 2\ y\ Vertice\ Aleatorio2 = 5$$

$$\begin{matrix} Color & \underline{1} & \underline{6} & \underline{4} & \underline{1} & \underline{2} & \cdots, & \underline{5} & \underline{3} \\ Vértice & \underline{1} & \underline{2} & \underline{3} & \underline{4} & \underline{5} & & \underline{n-1} & \underline{n} \end{matrix}$$

$$Color\ aleatorio1 = 7\ y\ Color\ aleatorio2 = 3$$

$$\begin{matrix} Color & \underline{1} & \underline{7} & \underline{4} & \underline{1} & \underline{3} & \cdots, & \underline{5} & \underline{3} \\ Vértice & \underline{1} & \underline{2} & \underline{3} & \underline{4} & \underline{5} & & \underline{n-1} & \underline{n} \end{matrix}$$

Intercambio de colores entre 2 vértices seleccionados de manera aleatorio:

$$Vertice\ Aleatorio1 = 1\ y\ Vertice\ Aleatorio2 = 4$$

$$\begin{matrix} Color & \underline{1} & \underline{6} & \underline{4} & \underline{8} & \underline{2} & \cdots, & \underline{5} & \underline{3} \\ Vértice & \underline{1} & \underline{2} & \underline{3} & \underline{4} & \underline{5} & & \underline{n-1} & \underline{n} \end{matrix}$$

$$\begin{matrix} Color & \underline{8} & \underline{6} & \underline{4} & \underline{1} & \underline{2} & \cdots, & \underline{5} & \underline{3} \\ Vértice & \underline{1} & \underline{2} & \underline{3} & \underline{4} & \underline{5} & & \underline{n-1} & \underline{n} \end{matrix}$$

2.3 Reference Set Update Method (Actualización del Conjunto de Referencia) (Resende et al. 2010)

Método para crear y actualizar el conjunto de referencias, el *Refset* es el conjunto de soluciones con el cual el algoritmo trabaja principalmente, esta ordenado de la mejor a la peor solución respecto a su dureza siendo la mejor solución la dureza más pequeña y la peor solución la dureza mayor, para nuestro algoritmo la función de dureza está dada por la suma de las distancias de los vértices que están pintados por un mismo color. Normalmente el método de actualización del conjunto de referencia se utiliza de dos maneras para su creación y su actualización en nuestro caso se utilizará en una tercer forma la cual la denominaremos Renovación parcial del conjunto de referencia (RPCR) y a continuación se presenta los tres usos de este método

a. *Creación*. Se utiliza después de generar la población de soluciones aleatorias *P* y funge para crear al conjunto de referencia. Para generar la primera parte del *ReftSet* se toman a las mejores soluciones del conjunto *P*, para tener el mayor

número de soluciones diversas se puede comprar el valor de resiliencia para de esta forma evitar soluciones repetidas. Para la segunda mitad del *RefSet* se utilizan las peores soluciones del conjunto *P* como se puede apreciar en la Fig. 2.2.

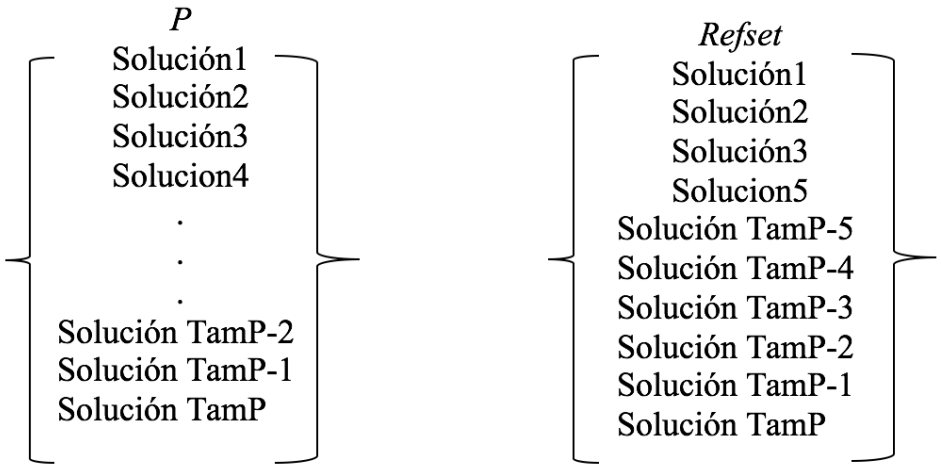

Figura 2.2. Ejemplificación de la creación del conjunto de referencia.

b. *Actualización*. Se encarga de determinar en cada iteración del algoritmo si el método de combinación de soluciones halló nuevas y mejores soluciones. Si las soluciones obtenidas por este método son mejores que las existentes en el conjunto de referencia y no existen, en este mismo se actualiza el *Refset* ubicando la solución en su lugar correspondiente con respecto a su dureza y desplazando la peor solución para, de esta forma, mantener el orden de la mejor a la peor solución. En el algoritmo original propuesto por Glover (Glover 1998) si no se hayan mejores soluciones el algoritmo da por terminado su ejecución.

c. *Renovación parcial del conjunto de referencia (RPCR)*. El criterio de paro original es remplazado por la estrategia que consiste en que el criterio de terminación este dado por un ciclo con cierto número de iteraciones, en el cual cuando el algoritmo no encuentra nuevas soluciones se prosigue a utilizar soluciones de la población de soluciones diversas *P* que no hayan sido previamente utilizadas para generar el conjunto de referencia y con estas remplazar la mitad baja del *RefSet*, cuando se hayan utilizado todas las soluciones del conjunto *P* y todavía no se cumpla con el número de iteraciones se utiliza los métodos de generación de soluciones diversas y método de mejora para generan una nueva población "P" y remplazar la mitad baja de nuestro conjunto de

referencia con soluciones diversas de “P” (Martí y Laguna 2003). Por lo cual el algoritmo ahora se vería como en la fig.2.3.

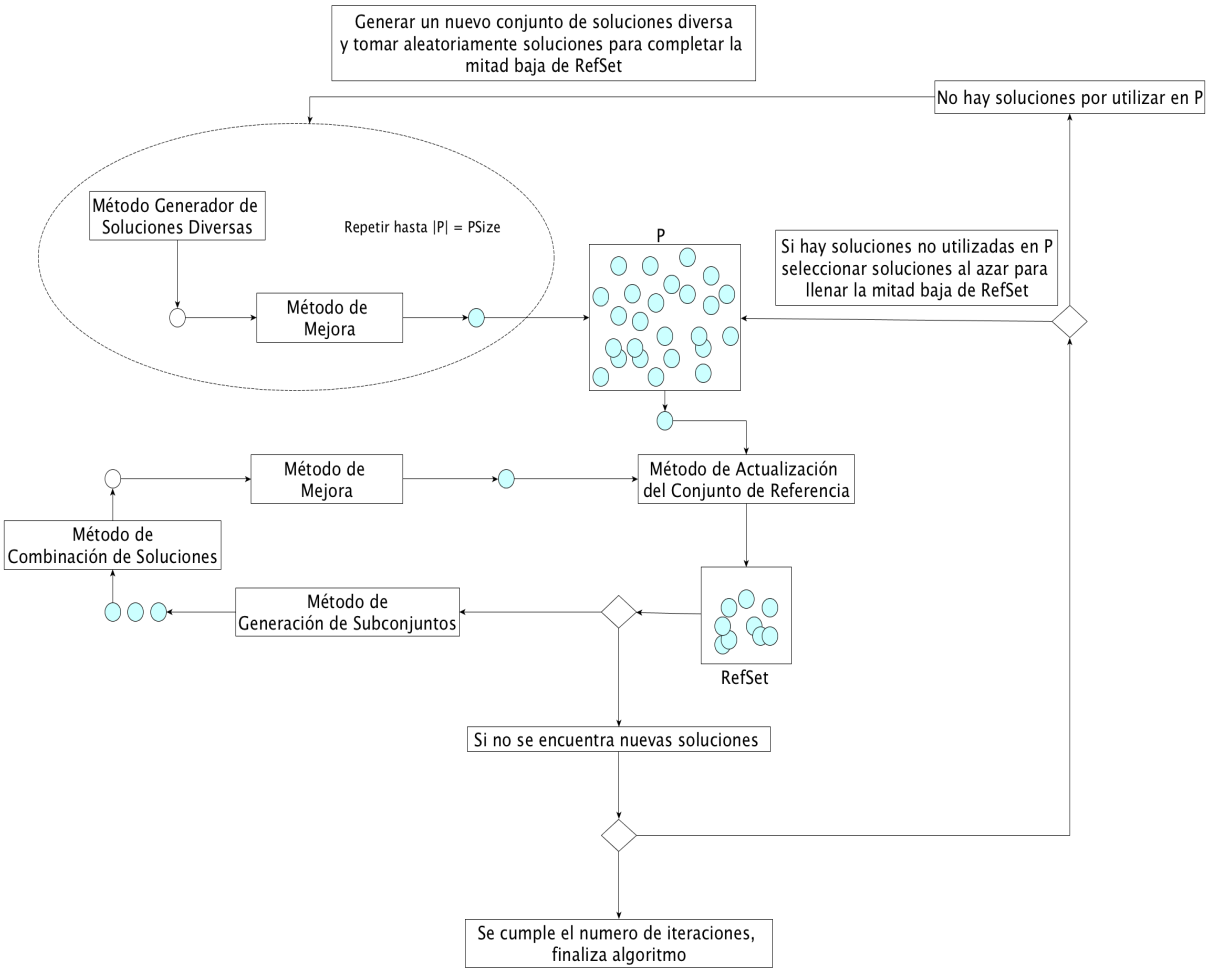

Figura 2.3. Algoritmo de búsqueda dispersa con la estrategia de *RPCR*.

2.4 Subset Generation Method (Método de Generación de Subconjuntos)

Método encargado de generar subconjuntos de soluciones del *Refset* a los cuales se les aplicara el método de combinación soluciones con el fin de obtener nuevas y mejores soluciones. Los subconjuntos pueden ser agrupados de dos en dos, de tres en tres, de cuatro en cuatro, hasta pueden ser del mismo tamaño del conjunto de referencia, pero se recomienda agrupar en parejas ya que de esta manera se obtienen los mejores resultados (Laguna y Armentano 2005), para el algoritmo propuesto se decidió agrupar los subconjuntos de la siguiente forma:

a. En parejas de tal manera que la mejor solución se combine con las segunda mejor solución, la tercera con la cuarta, así hasta la penúltima con la última solución,

de la siguiente forma: [1, 2],[2, 3],…,[b-1, b] ; *b* el tamaño de *RefSet,* como se puede observar en la Fig.2.4.

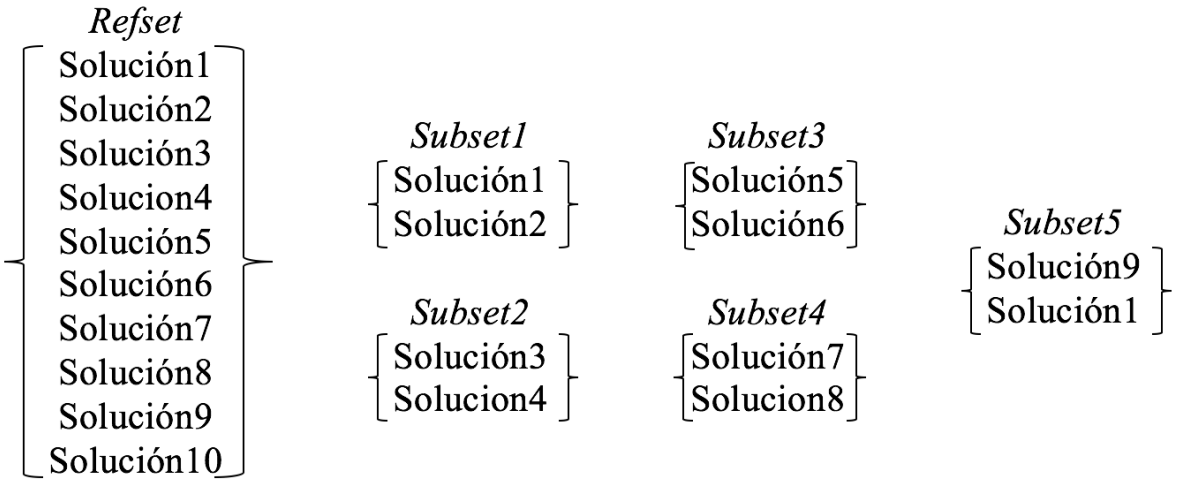

Figura 2.4. Generación de subconjuntos por parejas de manera continua.

b. En parejas utilizando un formato tipo torneo la mejor solución con la peor, segunda con la penúltima, quedando los subconjuntos de la siguiente forma: [1,b],[2, b-1],…,[b/2-1,b/2], como se puede apreciar en la Fig. 2.5.

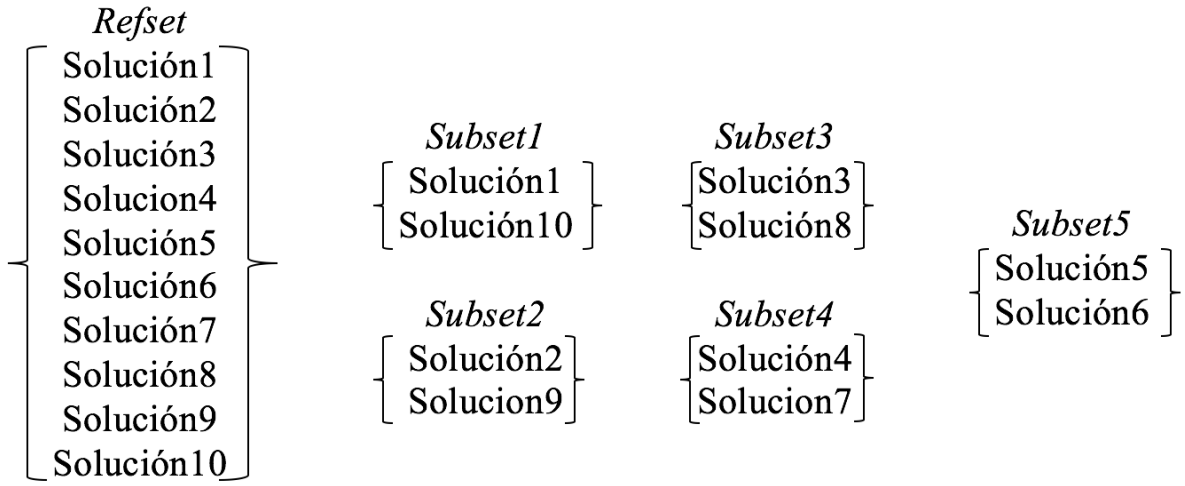

Figura 2.5. Generación de subconjuntos por parejas en modo torneo.

c. Y en tercias, agrupando la mejor solución con la segunda y la tercer mejor solución, has la antepenúltima con la penúltima y la peor solución, de tal manera que las tercias queden de la siguiente forma: [1, 2, 3],[4, 5, 6],…,[b-3,b-2,b-1] y el último grupo [1,b/2,b] como se observa en la Fig. 2.6.

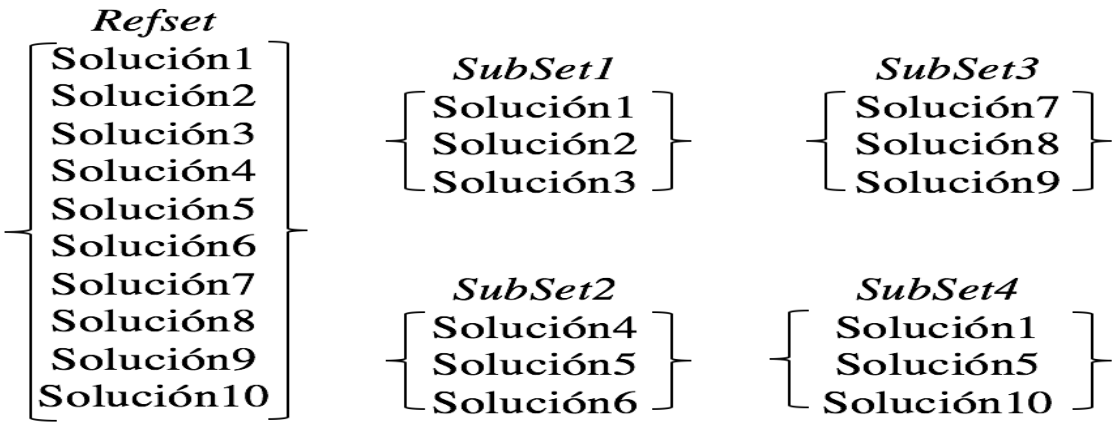

Figura 2.6. Generación de subconjuntos por tercias.

2.5 Solution Combination Method (Método de Combinación de Soluciones)

En este método se combinan soluciones a partir de los subconjuntos obtenidos por el Método de Generación de Subconjuntos, se proponen los siguientes métodos para realizar estas combinaciones:

a. *Crossover (Cruzamiento)*. Se caracteriza por combinar dos soluciones a través de un valor aleatorio r, de la primera solución se toman los elementos de la posición inicial hasta el valor aleatorio r, de la segunda solución se toman los elementos desde el valor aleatorio r hasta la posición final y con estas 2 partes se genera una nueva (de los Cobos Silva et al. 2010).

Ej. Cruzamiento con número aleatorio $r = 2$

$$Solución\ 1 = \begin{matrix} Color \\ Vértice \end{matrix} \begin{matrix} 1 & 1 & 2 & 6 & 4 \\ 1 & 2 & 3 & 4 & 5 \end{matrix}, \cdots, \begin{matrix} 2 \\ n-1 \end{matrix} \begin{matrix} 1 \\ n \end{matrix}$$

$$Solución\ 2 = \begin{matrix} Color \\ Vértice \end{matrix} \begin{matrix} 3 & 4 & 5 & 2 & 2 \\ 1 & 2 & 3 & 4 & 5 \end{matrix}, \cdots, \begin{matrix} 5 \\ n-1 \end{matrix} \begin{matrix} 6 \\ n \end{matrix}$$

$$Nueva\ Solución = \begin{matrix} Color \\ Vértice \end{matrix} \begin{matrix} 1 & 1 & 2 & 2 & 2 \\ 1 & 2 & 3 & 4 & 5 \end{matrix}, \cdots, \begin{matrix} 5 \\ n-1 \end{matrix} \begin{matrix} 6 \\ n \end{matrix}$$

b. *Local-híbrido*. Se planteó el uso de un hibrido entre búsqueda local y mutación (local-híbrido), el cual consiste en usar dos soluciones para generar una nueva solución de la siguiente manera: se genera un número aleatorio

entre (0,1) si el número aleatorio es menor a 0.5 se toma el elemento de la primera solución, de lo contrario se toma el elemento de la segunda solución y así sucesivamente para cada elemento de la nueva solución.

Ej. Local-hibrido supongamos que tenemos los siguientes números aleatorios r = 0.2, 0.8, 0.1, 0.7, 0.6, 0.7, 0.9 y 0.4, correspondiente a cada vértice de las dos soluciones agrupadas siguientes:

$$Solución\ 1 = \begin{matrix} Color \\ Vértice \end{matrix} \begin{matrix} 1 & 5 & 2 & 6 & 4 & & 2 & 5 \\ \overline{1} & \overline{2} & \overline{3} & \overline{4} & \overline{5}, \dots, & \overline{n-1} & \overline{n} \end{matrix}$$

$$Solución\ 2 = \begin{matrix} Color \\ Vértice \end{matrix} \begin{matrix} 3 & 1 & 5 & 2 & 2 & & 5 & 1 \\ \overline{1} & \overline{2} & \overline{3} & \overline{4} & \overline{5}, \dots, & \overline{n-1} & \overline{n} \end{matrix}$$

La nueva solución quedaría formada de la siguiente forma:

$$Nueva\ solución = \begin{matrix} Color \\ Vértice \end{matrix} \begin{matrix} 1 & 1 & 2 & 2 & 2 & & 5 & 5 \\ \overline{1} & \overline{2} & \overline{3} & \overline{4} & \overline{5}, \dots, & \overline{n-1} & \overline{n} \end{matrix}$$

El uso de los métodos puede ser de manera intercalada o con un número determinado de soluciones para cada uno.

Al finalizar el método de combinación de soluciones, todas las soluciones nuevas son procesadas con el método de mejora, de esta forma se mejora su calidad y busca al resultado óptimo o la mejor solución posible. Se utiliza el método de mejora antes descrito, las soluciones mejoradas son evaluadas por el método de actualización del conjunto de referencia para determinar si son mejores soluciones que las actuales del conjunto de referencia.

2.6. Algoritmo de búsqueda dispersa utilizando la estrategia de Renovación parcial del conjunto de referencia

---

1. *Comenzar con P = Ø. Utilizar el* ***método de generación de soluciones diversa*** *para construir soluciones*

De i = 1 a TamP

Genere una solución x_i *aleatoria*

Utilice el ***método de mejora*** *en* x_i *para obtener una mejor solución* x_i^*

Si $x_i^* \in / P$*, se incluye en P en otro caso, rechazar* x_i^*

2. *Utilizadando el* ***método de Actualización del Conjunto de Referencia*** *Construir el* ***conjunto de referencia*** *RefSet*
3. *Ordenar el RefSet de la mejor a la peor solución*
4. *Hacer Iteracion = 0*
5. ***Mientras*** *(Iteración < CriteriodeParo)*
6. *Generar los subconjuntos de RefSet utilizando el* ***método de Generación de Subconjuntos****.*
7. ***Mientras*** *(Se generen nuevas soluciones).*

8. Seleccionar un subconjunto y etiquetarlo como examinado.

9. Aplicar el ***método de combinación de soluciones*** *a los subconjuntos.*

10. Aplicar el ***método de mejora*** *a cada solución obtenida por la combinación*

11. Si es x_i^* es una mejor solución y no se encuentra en el *conjunto de referencia* llamar al ***método actualización del conjunto de referencia*** para agregar la solución al *RefSet*

12. Reordenar RefSet de la mejor a la peor solución

13. Iteracion = Iteracion + 1

14. Cuando no haya nuevas soluciones que mejoren el conjunto de referencia y no se haya cumplido el criterio de paro regresar al punto5 y utilizar el ***método de Actualización del Conjunto de Referencia*** *con la estrategia de* ***Renovación parcial del conjunto de referencia****, de lo contrario Terminar*

Capítulo 3
Implementación del Algoritmo

El algoritmo propuesto será implementado en el lenguaje de programación Python en su versión 2.7[1], se escogió este lenguaje de programación debido a la versatilidad que presenta, su facilidad de uso, como la amplia gama de bibliotecas con la que cuenta. Uno de los principales objetivos de este trabajo como del algoritmo de búsqueda dispersa es obtener soluciones de calidad en tiempos de ejecución adecuados por lo cual se escogió utilizar el paradigma de programación estructurado sobre el paradigma orientado a objetos, ya que el paradigma estructurado presenta mayor rapidez en tiempos de ejecución con respecto al paradigma orientado a objetos. A continuación, se describe el seudocódigo del programa y se explica a detalle las funciones que lo componen:

```
1 import random
2 import math
2 import copy
3 import time
4 start_time <- time.time()
```

Las líneas 1 a la 3 se componen de las tres bibliotecas, las cuales nos presentan utilidades importantes para el programa. La biblioteca *random* se encargará de generar números aleatorios los cuáles serán utilizados por los métodos de mejora, la biblioteca *copy* fungirá como utilidad a lo largo del programa para copiar las matrices de manera rápida y la biblioteca *time* sirve para medir los tiempos de ejecución del programa y es por ello que en la línea 4 se asigna a una variable cuando inicia la ejecución.

3.1 Función para cargar las instancias

```
1 FUNCIÓN lee ()
2       Lee el archivo de entrada
3       Hacer iteración_externa = 0
4       MIENTRAS (iteración_externa < N)
5               Hacer iteración_interna = 0
6               MIENTRAS (iteración_interna < N)
7                       Cada elemento guardarlo en la matriz dis
8                       Hacer iteración_interna += 1
```

1 El cual fue creado por Guido Van Rossum, y vio su primera versión en 1991

```
9           FINMIENTRAS
10          Hacer iteración_externa += 1
11      FINMIENTRAS
12      Regresa dis
13 FINFUNCIÓN
```

La función *lee* se encarga de almacenar en una matriz los datos de distancia que se encuentran en un archivo de texto, para posteriormente utilizar esta información y así obtener la dureza de nuestro programa. De las líneas 4 a la 11 se encargan de almacenar todos los datos en un matriz de nombre *dis*, se utiliza la siguiente instrucción ***dis <- [map(float,line.split()) for line in file]*** la cual es una cualidad de Python conocida como "comprensión de listas", que nos permite obtener datos de una matriz de manera rápida, en la cual se tendrían que utilizar los 2 mostrados en dichas líneas para obtener la información de distancia de nuestro archivo de texto. La línea 10 regresa la matriz con los datos de distancia a la función *main*.

3.2 Funciones para el problema de coloración de gráficas suaves

```
1 FUNCIÓN Repeticiones (color, solaEvaluar, numvertices)
2       Hacer iteración = 0
3       Inicializar lista temp_repe vacía
4       MIENTRAS (iteración < numvertices ):
5           SI (Elemento en solaEvaluar igual a color):
6               Guardar posición en temp_repe
7           FINSIN
8           Hacer iteración = iteración + 1
9       FIMIENTRAS
10      Regresa tempe_repe
11 FINFUNCIÓN
```

```
1 FUNCIÓN calculo (soltocal)
2       Hacer suma, iteración_externa = 0 e iteración_interma = 0
3       Almacenar el tamaño de la lista soltocal en lim_calc
4       MIENTRAS (iteración_externa < lim_calc)
            Almacenar el elemento de la lista soltocal que se encuentra en la posición
        iteración_externa, en la variable aux_posxte
5           MIENTRAS (iteración_interna <  lim_calc)
6               Almacenar el elemento de la lista soltocal que se encuentra en la posición
        iteración_interna, en la variable aux_posinter
7               suma = suma + distancia[iteración_externa][ iteración_interna]
```

```
8                   iteración_interna += 1
9               FINMIENTRAS
10              Hacer iteración_interna = 0 e iteración_externa += 1
11      FINMIENTRAS
12      Regresa suma
13 FINFUNCIÓN
```

```
1 FUNCÍON CalDureza (calsolucion, numvertices, colores)
2       Hacer color_count = 1 y dureza = 0
3       Inicializar una lista vacía llamada tempdur
4       MIENTRAS (color_count <= colores)
5               tempdur = Repeticiones (color_count, calsolucion, numvertices)
6               dureza = dureza + calculo (tempdur)
7               Hacer color_count += 1
8       FINMIENTRAS
9       Regresa dureza
10 FINFUNCIÓN
```

El cálculo de la dureza se lleva a cabo por las tres funciones anteriores, la función principal de este proceso es la función *CalDureza*, la cual se encargará en su ciclo *mientras* (ubicado en la línea 4) de buscar desde el primer color hasta el último todos los vértices que comparte cada color. En la línea 5 se llama a la función *Repeticiones,* que es la encargada de encontrar las posiciones de los vértices que comparte un color, y almacenarla en una lista temporal. Con estas posiciones se llama la función *calculo,* en la línea 6, en la cual, por medio de 2 ciclos y utilizando la lista obtenida por *Repeticiones*, encontramos las penalizaciones de los vértices que comparten un mismo color en la matriz de distancia y regresamos su suma, la cual es agregada en la variable *dureza,* para que al finalizar el ciclo de la función *CalDureza* se obtenga la dureza y regrese su valor para su posterior manipulación.

```
1 FUNCIÓN CalcSolidez (numvertices, colores, solidezdureza):
2       calcsoli = (2∗colores)∗(dureza) / (numvertices∗ (numvertices – colores)
3       Regresa calcsoli
4 FINFUNCIÓN
```

CalcSolidez es la función que se encarga de obtener el valor de solidez. Recibe el número de vértices, el número de colores que están presentes en la solución y el valor de la dureza, en la línea 2 se realiza la operación establecida en la fórmula de dureza para el problema de coloración de gráficas suaves antes mencionada.

1 ***FUNCIÓN CalcResiliencia*** (solidez_ante, solidez_actual):
2 $calcresi = \frac{solideante - solidezactual}{solidezactual}$
3 Regresa *calcresi*
4 ***FINFUNCIÓN***

Como en la anterior función, *CalcResiliencia* se encarga de realizar la operación establecida en la fórmula de resiliencia para el problema de coloración de gráficas suaves antes mencionado. En este caso se reciben los parámetros de la solidez de la coloración anterior y de la solidez actual.

3.3 Funciones del método generador de soluciones diversas

A continuación, se describen dos funciones fundamentales para el método generador de soluciones diversa y para el algoritmo.

1 ***FUNCIÓN comparacion*** (compsolucion, compdatos):
2 *Hacer intcompa = 0 y rescompara = 0*
3 ***MIENTRAS*** (intcompa sea menor que el número de elementos de compdatos):
4 ***SI*** (*compsolucion es igual a la resiliencia en la lista compdatos en su posición intcompa*)
5 Hacer *rescompara = 1*
6 Terminar ciclo
7 ***FINSI***
8 ***SINO***:
9 Hacer *intcompa += 1*
10 ***FINSINO***
11 ***FINMIENTRAS***
12 Regresar *rescompara*
13 ***FINFUNCIÓN***

La función anterior es la encargada de comparar que no haya una solución igual dentro de nuestra población *P* de soluciones diversa o del conjunto de referencia. Para lograr este objetivo recibe la resiliencia de la solución que se quiere integrar a cualquiera de estos pools y una matriz que almacena los datos de las soluciones que se encuentran actualmente en estos conjuntos, estos datos se conocen dentro de la función como *compsolucion* y *compdatos* respectivamente. Un ciclo *while* se encarga de recorrer la matriz en busca de una resiliencia igual, si la encuentra regresa un valor de uno indicado que

ya existe en la matriz por lo que se rechaza, de lo contrario regresa un valor de cero que indica que es una solución nueva y por lo tanto debe de ser agregada al pool.

```
1 FUNCIÓN Ordena (tem_poll, tem_datos, ord_tam):
2       Hacer ord_itera = 0
3       MIENTRAS (ord_itera < ord_tam):
4               Hacer ord_pos = ord_itera y ord_lugar = ord_itera + 1
5               MIENTRAS (ord_lugar < ord_tam):
6                       SI (La dureza en la posición ord_lugar < La dureza en la posición
ord_pos):
7                               Hacer ord_pos  = ord_lugar
8                       FINSI
9                       ord_lugar += 1
10              FINMIENTRAS
11              Inserta la solución en la posición ord_pos en la posición ord_intera en la lista
tem_poll
12              Inserta el dato en la posición ord_pos en la posición ord_intera en la lista
tem_poll
13              ord_itera += 1
14      FINMIENTRAS
15      Regresar tem_poll y tem_datos
16 FINFUNCIÓN
```

La función *Ordena* se encarga como su nombre lo indica de ordenar las soluciones de menor a mayor dureza, para esto recibe una lista con las soluciones, una lista con los datos y el tamaño de las listas, esta información está almacenada en las variables *tem_poll, tem_datos y ord_tam* respectivamente. El ciclo *while* exterior, que empieza en la línea 3, se encarga de controlar la posición de inserción, el *while* de la línea 5 compara las soluciones y almacena la que tiene la menor dureza, hasta recorrer todo el pool de soluciones. Al finalizar, el *while* interno en la línea 10 se quita la solución y el valor de sus posiciones antiguas y se inserta en su nueva posición. Siempre buscará la dureza menor que estará en la posición 1, después la segunda menor dureza y así sucesivamente hasta ordenar toda la lista, regresa las listas ordenadas de soluciones y datos.

```
1 FUNCIÓN CreaSol (tamP, colores, numvertices, distancia, solidez_ini, limmejora)
2       Inicializar las listas vacías unasoulucion_aleato, undato_aleto, mejor_solucion,
dato_mejor, pollP_temp y datos_poll_temp
3       DESDE ciclo_inicial = 1 a tamP
```

```
4        unasoulucion_aleato, undato_aleto = MetGenaraSolAletorias (numvertices,
    colores, solidez_ini)
5        mejor_solucion, dato_mejor = MetMejora (unasoulucion_aleato, numvertices,
    colores, solidez_ini, limmejora)
6        SI ((comparacion(dato_mejor[Resiliencia], datos_poll_temp) = 0) ó
(ciclo_inicial = 1) ):
7            ciclo_inicial += 1
8            Agregar la solución mejor_solución a la población de soluciones diversas
pollP_temp.
9            Agregar los datos dato_mejo a la población de datos de soluciones
diversas datos_poll_temp
10       FINSI
11   FINDESDE
12   pollP_temp, datos_poll_temp = Ordena (pollP_temp, datos_poll_temp, tamP)
13   Regresar pollP_temp y datos_poll_temp
14 FINFUNCIÓN
```

Esta función funge como el ciclo de búsqueda dispersa que se encarga de generar la población *P* de soluciones dispersas, el ciclo *desde* en la línea 3 se encarga de llamar a los métodos de *MetGenaraSolAletorias* y *MetMejora* para generar una solución aleatoria y después mejorarla, el *Si* de la línea 6 nos indica que si es la primera solución o la solución no se encuentra en la población *P* utilizando el método *comparación,* sí se cumple esta indicación se agrega al pool, de lo contrario se vuelven a invocar los métodos *MetGenaraSolAletorias* y *MetMejora,* hasta llenar el conjunto *P* de solucione aleatorias, al finalizar el ciclo *desde* se invoca a las función *Ordena* para tener la población de soluciones en orden de menor a mayor dureza, regresa el conjunto *P* como también una lista de datos que contiene la dureza, solidez y resiliencia respectiva de cada solución del conjunto *P*, ambas listas están ordenas.

```
1 FUNCIÓN MetGenaraSolAletorias (numvertices, colores, distancia, alea_solidez):
2    Inicializar las listas vacías garray y gdatos
3    DESDE gj = 1 a numvertices
4        Generar un color aleatorio de uno a colores y almacenarlo en rancolor
5        Agregar color en lista garray
6    FINDESDE
7    Agregar CalDureza (garray, distancia, numvertices) a gdatos
8    Agregar CalcSolidez (numvertices, colores, gdatos[Dureza]) a gdatos
9    SI (colores = 1):
10       Agregar el valor de resiliencia cero a gdatos
11   FISI
```

12 ***SINO***:
13 Agregar CalcResiliencia (alea_solidez, gdatos[Solidez]) a *gdatos*
14 ***FINSINO***
15 Regresar *garray y gdatos*
16 ***FINFUNCIÓN***

La función *MetGenaraSolAletorias* se encarga tanto de generar una solución completamente aleatoria como de obtener los datos de dureza, solidez y resiliencia para esta misma. Recibe como parámetros *numvertices* que es el número de vértices que tiene la gráfica, *colores* es la variable que contiene el número de colores que se busca, y *alea_solidez,* que es la variable que contiene la solidez obtenida anteriormente de existir. Se utilizan 2 listas *garray* las cuales almacenan la solución aleatoria generada y *gdatos* donde se almacenan los datos de dureza, solidez y resiliencia. En el ciclo *desde,* que comprende las líneas 3 a la 6, se genera la solución para cada vértice, se produce un color aleatorio y éste es agregado al arreglo *garray*. De la línea 7 a la 8 se agregan al arreglo *gdatos* los datos de dureza y solidez obtenidos por las funciones respectivas, en la línea 9, también se verifica si es una solución en la cual se utiliza un solo color pues de ser así se agrega una resiliencia igual a cero, si por el contrario es una solución de más de un color se utiliza la función de resiliencia como se observa en la línea 12. Al finalizar el método se regresa la solución y sus datos de dureza, solidez y resiliencia.

3.4 Funciones del método de mejora

1 ***FUNCIÓN Muta1color*** (numvertices, colores, solaMuta)
2 Seleccionar un vértice al azar entre 0 y numvertices y guardarlo en *randvert*
3 Seleccionar un color aleatoriamente y almacenarlo en *randcolor*
4 *solaMutam[randvert] = randcolor*
5 Regresar *solaMuta*
6 ***FINFUNCIÓN***

La función que anterior recibe como parámetros *numvertices,* que es el número de vértices, *colores,* que es el número de colores que tiene la solución, y *solaMuta,* que es un arreglo que contiene la solución a la cual se le realizara el proceso de cambiar el color de un vértice aleatorio por otro color de igual forma aleatoriamente seleccionado. Es utilizado por nuestro algoritmo de búsqueda local la cual funge como Método de Mejora del algoritmo de búsqueda dispersa, regresa al Método de Mejora la solución con los cambios realizados.

1 ***FUNCIÓN Muta2color*** (numvertices, colores, solaMuta)
2 Seleccionar dos vértices al azar entre 0 y numvertices y guardarlos en *randvert1 y randvert2*

3 Seleccionar dos colores aleatoriamente y almacenarlos en *randcolor1 y randcolor2*
4 *solaMutam[randvert1] = randcolor1*
5 *solaMutam[randvert2] = randcolor2*
6 Regresa *solaMuta*
7 ***FINFUNCIÓN***

La función Muta2color como Muta1color es utilizada por el método de mejora para realizar el cambio de dos vértices seleccionados de manera azarosa por dos colores aleatorios, para este objetivo recibe de igual forma el número de vértices que tiene la solución, los colores y la solución a la cual se le realizara el proceso.

1 ***FUNCIÓN intercambio*** (numvertices, solacambiar)
2 Seleccionar de manera aleatoria dos vértices entre 0 y *numvertices*, almacenarlos en *randvert1 y randvert2*
3 Realizar el intercambio de colores de los vértices seleccionados en *solacambiar*
4 Regresar *solacambiar*
5 ***FINFUNCIÓN***

De la misma manera que las funciones anteriores la función *intercambio* es utilizada por el método de mejora, su objetivo es intercambiar los colores entre dos vértices seleccionados aleatoriamente de una solución, en Python se utiliza una función de la biblioteca random con lo cual garantiza que se seleccionaran dos vértices diferentes. La instrucción es la siguiente: *rangointer = random.sample(range(numvertices), 2), rangointer* es una lista que almacenara los dos vértices aleatorios y de esta forma se realiza el intercambio de manera más sencilla y rápida, para esta función solo es necesario el número de vértices y la solución a la cual se realizara el intercambio

1 **FUNCION *MetMejora*** (solucionamejorar, numvertices, colores, solidezanterior, limmejora)
2 Copiar solucionamejorar en *i*
3 Hacer *mejor_solidez = 0 y mejor_resiliencia = 0*
4 Inicializar una lista vacía para almacenar los datos llamada *dato*
5 ***DESDE** itera = 1 a limmejora*
6 *j = Muta1color (numvertices-1, colores, i)*
7 fj = CalDureza (j, distancia, numvertices)
8 fi = CalDureza (i, distancia, numvertices)
9 solidezi = CalcSolidez (numvertices, colores, fi)
10 solidezj = CalcSolidez (numvertices, colores, fj)
11 **SI** (*fj* es menor o igual que *fi*):

```
12                  SI (CalcResiliencia (solidezanterior, solidezj) es diferente de
CalcResiliencia (solidezanterior, solidezi))
13                      Copiar j en i
14                      Hacer fi = fj
15                      Hacer mejor_solidez = solidezj
16                  FINSI
17              FINSI
18      FINDESDE
19      mejor_resiliencia = CalcResiliencia (solidezanterior, mejor_solidez)
20      Agregar fi a dato
21      Agregar mejor_solidez a dato
22      Agregar mejor_resiliencia a dato
23      Regresar i y dato
24 FINFUNCION
```

La función *MetMejora* es el método de mejora de búsqueda dispersa, la cual es una búsqueda local, este método recibe una solución a mejorar, el número de vértices, el número de colores que se buscan, la solidez anterior y el número de iteraciones a realizar, estos valores se almacenan en las variables *solucionamejorar*, *numvertices*, *colores*, *solidezantereio y limmejora* respectivamente, se inicializa la variable *i* con la solución a mejorar, *mejor_solidez* y *mejor_resiliencia* son variable en las cuales se almacenaran la solidez y resiliencia que se vayan obteniendo a lo largo del método y que al final junto con el valor de dureza serán almacenados en el arreglo *dato*. El ciclo *desde* que comprende de la línea 5 a la 18 se encarga de buscar mejorar la solución, *j* será la variable en la cual se almacene la solución que contiene *i* después de pasar por el método *Muta1color,* las variables *fi* y *fj* tendrán almacenados los valores de dureza de *i* y *j* respectivamente. En la línea 11 se compara si la dureza de la solución después de aplicar el método *Muta1color* (*fj*) es menor o igual a la dureza original (*fi*) si sí se comparan las resiliencias para determinar si la solución es distinta, si es distinta *j* se copia en *i* y *fi* se iguala a *fj*, este procedimiento se repite por un cierto número de iteraciones igual a *limmejora*. Al finalizar el ciclo *desde* se obtienen el valor final de resiliencia los cuales junto con la dureza son agregados al arreglo en cargado de almacenar los datos de la solución, se regresa la solución mejorada y los datos.

3.5 Funciones del método de actualización del conjunto de referencia.

```
1 FUNCIÓN ActRefSet1(sol_insert, datos_insert, tamRefset):
2       Inicializar las listas vacías act_sol y act_datos
3       Hacer act_itera = 0 y act_limite = tamaño de sol_insert - (tamRefset/2)
4       MIENTRAS (act_itera < (tamRefset/2)):
```

5 Agregar la solución de la lista sol_insert en la posición act_itera a la lista act_sol
6 Agregar los datos de la lista datos_insert en la posición act_itera a la lista act_datos
7 Hacer act_itera += 1
8 ***FINMIENTRAS***
9 Hacer act_itera = 0
10 ***MIENTRAS*** (act_itera < tamRefset/2):
11 Agregar la solución de la lista sol_insert en la posición act_limite + act_itera a la lista act_sol
12 Agregar los datos de la lista datos_insert en la posición act_limite + act_itera a la lista act_datos
13 Hacer act_itera +=1
14 ***FINMIENTRAS***
15 Regresar act_sol y act_datos
16 ***FINFUNCIÓN***

El método de Actualización del conjunto de referencia del algoritmo de búsqueda dispersa se caracteriza por ser utilizado para la creación y actualización del mismo, en nuestro caso se remplaza el criterio de terminación que el algoritmo finalice al no encontrar nuevas soluciones, por un ciclo con cierto número de iteraciones, en el cual si el algoritmo no encuentra nuevas soluciones y ya agregaron todas las soluciones de las población *P* de soluciones diversa se prosigue a utilizar los métodos de generación de soluciones diversas. El método de mejora y el método de actualización del conjunto de referencia, al invocar los 2 primero métodos para generan una nueva población *P* y con el ultimo método remplazar la mitad baja de nuestro conjunto de referencia con soluciones diversas de *P* (Martí y Laguna 2003). Por lo cual se generaron tres funciones para el método de actualización de referencia, la primera para su creación, la segunda para su actualización y la tercera para cuando no se encuentra nuevas soluciones. La función anterior se encarga de la creación del *Refset,* el primer ciclo *mientras* de la función que se encuentra en la línea 4 se encarga de llenar la primera parte del conjunto de referencia con las mejores soluciones de la población *P* de soluciones diversas, el segundo ciclo *mientras* que empieza en la línea 8 se encarga de llenar la segunda mitad del conjunto de referencia con las peores soluciones del conjunto *P* de soluciones diversas, los datos de dureza, solidez y resiliencia de las soluciones son almacenados en una matriz aparte en la cual están ordenadas de tal forma que estén en la misma posición que su solución correspondiente, se regresa el conjunto de referencia y su matriz correspondiente con sus datos.

1 ***FUNCIÓN ActRefSet2***(act_refset, act_refdata, nuevas_soluciones, datos_nuevasol, tamRefset):
2 Hacer act_iteranuevas = 0, sol_nueva = 0 y act_tope = tamaño de datos_nuevasol

```
3       MIENTRAS (act_iteranuevas < act_tope):
4               Hacer valor_nuevo = datos_nuevasol[act_iteranuevas][Resiliencia]
5               Hacer act_recorre = 0
6               SI (comparacion (valor_nuevo, act_refdata) = 0):
7                       MIENTRAS (act_recorre < tamRefset):
8                               SI (valor_nuevo < act_refdata[act_recorre][Dureza] ):
9                                       Eliminar de la lista act_refset el último elemento
10                                      Eliminar de la lista act_refdata el último elemento
11                                      Inserta la solución de la lista nuevas_soluciones en la
posición act_interanuevas en la lista act_refset en la posición act_recorre
12                                      Inserta los datos de la lista datos_nuevasol en la posición
act_interanuevas en la lista act_refdat en la posición act_recorre
13                                      Hacer sol_nueva += 1
14                                      Finalizar ciclo
15                              FINSI
16                              SINO:
17                                      Hacer act_recorre += 1
18                              FINSINO
19                      FINMIENTRAS
20              FINSI
21              Hacer act_iteranuevas += 1
22      FINMIENTRAS
159     Regresar act_refset, act_refdata y sol_nueva
160 FINFUNCIÓN
```

ActRefSet2 es la función que se encarga del proceso de actualización del conjunto de referencia, recibe como parámetros el *Refset*, los datos correspondientes al *Refset*, las nuevas soluciones que se quieren agregar, los datos correspondientes a estas soluciones y el tamaño del conjunto de referencia, son almacenadas en las variables *act_refset*, *act_refdata*, *nuevas_soluciones*, *datos_nuevasol* y *tamRefset* correspondientemente. El ciclo *mientras* en la línea 3 controla que todas las nuevas soluciones sean comparadas para determinar si ya existen en el conjunto de referencia, utilizando el método *comparacion*. Si no existen en el conjunto de referencia, el ciclo *mientras* en la línea 7 buscará la posición que le corresponde a la solución en el conjunto de referencia que esta ordenado de la menor a la mayor dureza, al encontrar su posición se quita la peor dureza y se inserta la solución en el lugar que le corresponde, a su vez también los datos de las nuevas soluciones se insertan en el orden correspondiente en una matriz alterna, este proceso se realiza para todas las nuevas soluciones, regresa el conjunto de referencia actualizado, los datos actualizados y el valor de "sol_nueva" para determinar si se encontró una nueva solución.

```
1 FUNCIÓN ActRefSet3(refset_act, datos_refset_act, tamRefset, poblacion, datos_pobla, pos_ini):
2       Hacer refset_itera = tamRefset/2 y  refset_recorre = 0
3       MIENTRAS (refset_itera < tamRefset):
4             Seleccionar aleatoriamente una solución no utilizada anteriormente de poblacion y almacenarla en rand_act
5             SI ((comparacion(datos_pobla[rand_act][Resiliencia], datos_refset_act) = 0)):
6                   Hacer refset_act[refset_itera] = poblacion[rand_act]
7                   Hacer datos_refset_act[refset_itera] <- datos_pobla[rand_act]
8                   Hacer refset_itera += 1
9                   Hacer refset_recorre += 1
10            FINSI
11            SINO:
12                  Seleccionar aleatoriamente otra solución no utilizada anteriormente de poblacion y almacenarla en rand_act
13            FINSINO
14      FINMIENTRAS
15      Regresar refset_act y datos_refset_act
16 FINFUNCIÓN
```

Esta es la última función que actualiza el conjunto de referencia cuando no se encuentra nuevas soluciones. El ciclo *mientras* de esta función ubicado en la línea 3 se cerciora de que las soluciones seleccionadas aleatoriamente para repoblar el conjunto de referencia después de no encontrar nuevas soluciones no se encuentren ya en el *Refset*, por lo cual se usa el método *comparacion*. Si no se encuentran se prosigue a llenar la mitad baja del conjunto de referencia con las nuevas soluciones, se regresa el conjunto de referencia y sus datos actualizados.

3.6 Funciones del método de generación de subconjunto

```
1 FUNCION SubGenara (divi_refset):
2       Hacer sub_ini = 0, sub_cola= tamaño divi_refset - 1 y sub_limite = tamaño divi_refset/2
3       Inicializar la lista subconjunto vacía
4       MIENTRAS (sub_ini < sub_limite):
5             Agregar a la lista subconjunto la solución de la lista divi_refset en la posición sub_ini
6             Agregar a la lista subconjunto la solución de la lista divi_refset en la posición sub_cola
7             Hacer sub_ini += 1
```

```
8           Hacer sub_cola -= 1
9       FINMIENTRAS
10      Regresar subconjunto
11 FINFUNCION
```

La función *SubGenera* representa el método de generación de subconjuntos, el cual recibe el conjunto de referencia como la variable *divi_refset*. Los subconjuntos serán creados de tal manera que la mejor solución se combine con la peor, la segunda mejor se combine con la segunda peor, así de tal manera que todas estén en parejas. El ciclo *mientras*, ubicado en la línea 4, regresa los subconjuntos obtenidos por esta combinación.

```
1 FUNCIÓN SubGenaraM (divi_refset):
2       Hacer sub_pri = 0, sub_seg =1 y sub_limite = tamaño divi_refset
3       Inicializar la lista subconjunto vacía
4       MIENTRAS (sub_seg < sub_limite):
5           Agregar a la lista subconjunto la solución de la lista divi_refset en la posición
sub_pri
6           Agregar a la lista subconjunto la solución de la lista divi_refset en la posición
sub_seg
7           Hacer sub_pri += 2
8           Hacer sub_seg += 2
9       FINMIENTRAS
10      Regresar subconjunto
11 FINFUNCION
```

SubGeneraM es la función que genera los subconjuntos en parejas pero agrupando la mejor solución con la segunda mejor, la tercer mejor con la cuarta mejor hasta la penúltima mejor solución con la última solución, representa también el metodo de generación de subconjuntos.

```
1 FUNCIÓN SubGenaraTri (divi_refset):
2       Hacer sub_pri = 0, sub_seg =1, sub_ter = 2 y sub_limite = tamaño divi_refset
3       Inicializar la lista subconjunto vacía
4       MIENTRAS (sub_ter < sub_limite):
5           Agregar a la lista subconjunto la solución de la lista divi_refset en la posición
sub_pri
6           Agregar a la lista subconjunto la solución de la lista divi_refset en la posición
sub_seg
7           Agregar a la lista subconjunto la solución de la lista divi_refset en la posición
sub_ter
```

```
8           Hacer sub_pri += 3
9           Hacer sub_seg += 3
10          Hacer sub_ter += 3
11      FINMIENTRAS
12      Agregar a la lista subconjunto la solución de la lista divi_refset en la posición 0
13      Agregar a la lista subconjunto la solución de la lista divi_refset en la posición
sub_limite/2
14      Agregar a la lista subconjunto la solución de la lista divi_refset en la posición sub_limite
15      Regresar subconjunto
16 FINFUNCIÓN
```

Esta última función se encarga de agrupar las soluciones en tercias, esto se realiza en el ciclo *mientras* de la siguiente forma: la mejor solución con la segunda y tercer mejor solución hasta la ante antepenúltimo, la antepenúltima y la penúltima solución, para al final agregar una tercia con la mejor solución, la solución de en medio y la última solución.

3.7 Funciones del método de combinación de soluciones.

```
1 FUNCIÓN hibrido (hib_inicio, hib_final, numvertices):
2       Hacer hib_recorre = 0;
3       Inicializar l alista hib_tem vacía
4       MIENTRAS (hib_recorre < numvertices):
5           Seleccionar un nuemero aleatorio entre todos los número posibles 0 y 1
6           SI (hib_rand < 0.5):
7               Agregar a la lista hib_tem el color en la lista hib_inicio en la posición
hib_recorre
8           FINSI
9           SINO:
10              Agregar a la lista hib_tem el color en la lista hib_final en la posición
hib_recorre
11          FINSINO
12          Hacer hib_recorre += 1
13      FINMIENTRAS
14      Regresar hib_tem
15 FINFUNCIÓN
```

La función *hibrido* es una herramienta de la función *MetComHibus*, la cual es la representación del método de combinación de soluciones. Consiste en combinar los dos miembros del subconjunto, en el algoritmo se plantea el uso de un método hibrido entre búsqueda local y

mutación, el cual consiste en los siguiente: la función recibe como parámetros las dos soluciones del subconjunto que se encuentra en las variables *hib_inicio* y *hib_final*, como también recibe la variable *numvertices* el cual es el número de vértices de las soluciones, el ciclo *mientras* de la línea 4 va a generar un número aleatorio entre (0, 1) si el número aleatorio es menor a 0.5 se tomara para ese vértice el color de la solución que esta almacenada en *hib_inicio*, de lo contrario se tomará el color de la solución almacenada en *hib_final*. Este procedimiento se realizará el número de veces igual al número de vértices, para de esta forma generar una nueva solución y regresar la nueva solución.

```
1 FUNCIÓN MetCombHibus (hibsub_combina, lim_hib, numvertices):
2       Hacer hib_intera = 0
3       Inicializar la lista hib_newsol vacía
4       MIENTRAS (hib_intera < lim_hib):
5             Agregar a la lista hib_newsol la solución obtenida por hibrido
      (hibsub_combina[hib_intera], hibsub_combina[hib_intera+1], numvertices)
6             Hacer hib_intera += 2
7       FINMIENTRAS
8       Regresar hib_newsol
9 FINFUNCION
```

Esta función se encarga en su ciclo *mientras,* ubicado en la línea 4, de enviar los subconjuntos obtenidos por el método de generación de subconjuntos a la función *hibrido* para generar nuevas soluciones, estas nuevas soluciones son almacenadas en una matriz, esta matriz regresa la función *MetCombHibus*.

```
1 FUNCIÓN hibridotri (hib_inicio, hib_mid, hib_final, numvertices):
2       Hacer hib_recorre = 0;
3       Inicializar l alista hib_tem vacía
4       MIENTRAS (hib_recorre < numvertices):
5             Seleccionar un nuemero aleatorio entre todos los número posibles 0 y 1
6             SI (hib_rand <= 0.34):
7                   Agregar a la lista hib_tem el color en la lista hib_inicio en la posición
hib_recorre
8             FINSI
9             SI (hib_rand > 0.34 y hib_rand <= 0.67)
10                  Agregar a la lista hib_tem el color en la lista hib_mid en la posición
hib_recorre
9             SINO:
```

10 Agregar a la lista hib_tem el color en la lista hib_final en la posición hib_recorre
11 ***FINSINO***
12 Hacer hib_recorre += 1
13 ***FINMIENTRAS***
14 Regresar hib_tem
15 ***FINFUNCIÓN***

1 ***FUNCIÓN MetCombHibus*** (hibsub_combina, tamRefset, numvertices):
2 Hacer hib_intera = 0
3 Inicializar la lista hib_newsol vacía
4 **MIENTRAS** (hib_intera < tamRefset):
5 Agregar a la lista hib_newsol la solución obtenida por hibrido (hibsub_combina[hib_intera], hibsub_combina[hib_intera+1], hibsub_combina[hib_intera+2], numvertices)
6 Hacer hib_intera += 2
7 **FINMIENTRAS**
8 Regresar hib_newsol
9 **FINFUNCION**

Las dos funciones anteriores son el equivalente del método local-hibrido para combinar soluciones para el caso de tercias.

1 ***FUNCIÓN cruzamiento*** (cruz_pos, cruz_inicio, cruz_final):
2 Agregar a cruz_tem los elementos de la solución cruz_inicio desde el primer elemento hasta cruz_pos y los elementos de la solución cruz_final desde cruz_pos hasta el último elemento
3 Regresar cruz_tem
4 ***FINFUNCIÓN***

1 ***FUNCIÓN MetCombinaCruz*** (sub_combina, lim_cruz, numvertices):
2 Hacer cruz_pos = 0 y cruz_itera = 0
3 Inicializar la lista vacía cruz_newsol
4 ***MIENTRAS*** (cruz_itera < lim_cruz):
5 Se selecciona de manera aleatroia el vértice donde se realizare el corte y se almacena en cruz_pos
6 Se agrega la solución obtenida por cruzamiento (cruz_pos, sub_combina[cruz_itera],sub_combina[cruz_itera+1]) en la lista cruz_newsol
7 Hacer cruz_itera +=2

```
8       FINMIENTRAS
9       Regresar cruz_newsol
10 FINFUNCIÓN
```

La función *MetCombinaCruz* representa al método de *cruzamiento* el cual es utilizado como método de combinación de soluciones y el cual es un método común de los algoritmos genéticos, el ciclo *mientras* de este método se encarga de generar para cada subconjunto un número aleatorio el cual representa el vértice en el cual se realizará el corte, esta información junto con el par de soluciones son enviados al método de cruzamiento el cual se encarga de realizar el proceso de generar una nueva solución de la siguiente manera: De la primer solución se tomara del primer elemento de la solución hasta el elemento que se encuentre en el vértice seleccionado aleatoriamente y de la segunda solución se tomaran desde el vértice siguiente del vértice seleccionado azarosamente hasta el último elemento de esta solución. Con estos elementos se obtiene una nueva solución, esta nueva solución es regresada.

3.8 Función principal de búsqueda dispersa

```
1 FUNCIÓN BusquedaDispersa (colores, numvertices, tamP, tamRefset, solidez_ini):
2       Hacer ciclo_principal = 0 y bandera = 1
3       Inicializar listas vacías pollP, datos_poll, refset, datos_refset, sub_refset, comb_solu,
nuevas_soluciones, datos_nuevasol, new_sol y new_data
4       pollP, datos_poll = CreaSol (tamP, colores, numvertices, solidez_ini, limmejora)
5       MIENTRAS (ciclo_principal < limcicloprin):
6               SI (ciclo_principal = 0):
7                       refset, datos_refset = ActRefSet1(pollP, datos_poll, tamRefset)
8               FINSI
9               SINO (bandera = 0):
10                      SI ( Tamaño pollP < tamRefset/2):
11                              Hacer listas vacías pollP y datos_poll <- []
12                              pollP, datos_poll = CreaSol (tamP, colores, numvertices,
distancia, solidez_ini, limmejora)
13                      FINSI
14                      refset, datos_refset = ActRefSet3(refset, datos_refset, tamRefset, pollP,
        datos_poll)
15                      Hacer listas vacías nuevas_soluciones y datos_nuevasol
16                      Hacer bandera = 1
17              FINSINO
18              SINO:
```

```
19                  refset, datos_refset, bandera = ActRefSet2 (refset, datos_refset,
    nuevas_soluciones, datos_nuevasol, tamRefset)
20                  Hacer listas vacías nuevas_soluciones y datos_nuevasol
21          FINSINO
22          sub_refset = SubGenara (refset)
23          comb_solu = MetCombHibus (sub_refset, tamRefset, numvertices)
24          Hacer ciclo_mejora = 0
25          MIENTRAS (ciclo_mejora < tamaño comb_solu):
26                  new_sol, new_data = MetMejora (comb_solu[ciclo_mejora],
    numvertices, distancia, colores, solidez_ini, limmejora)
27                  SI ((comparacion(new_data[Resiliencia], datos_nuevasol) = 0) OR
(ciclo_mejora== 0) ):
28                          Agregar new_sol a la lista nuevas_soluciones
29                          Agregar new_data a la lista datos_nuevasol
30                  FINSI
31                  Hacer ciclo_mejora += 1
32          FINMIENTRAS
33          nuevas_soluciones, datos_nuevasol = Ordena (nuevas_soluciones,
    datos_nuevasol, tamaño datos_nuevasol)
34          Hacer ciclo_principal += 1
35      FINMINETRAS
36      Regresar refset[0], datos_refset[0]
37 FINFUNCIÓN
```

La función *BusquedaDispersa* es la representación principal del algoritmo de búsqueda dispersa. Es la función que se encarga de llamar a las funciones que fungen como métodos del algoritmo, recibe como parámetros el número de colores que se buscarán, el número de vértices de la gráfica, el tamaño que se quiere de la población *P* y del conjunto de referencia y la solidez que se obtuvo de la coloración anterior estos datos están almacenados en las variables *colores*, *numvertices*, *tamP*, *tamRefset* y *solide_ini* respectivamente. Se inicializan las matrices y las listas vacías para de esta forma evitar problemas con información anterior, como también se inicializan las variables *ciclo_principal* en cero el cual controla número de iteraciones que realizara el algoritmo para obtener nuevas soluciones y la variable bandera a uno, que cambiará a cero cuando no se encuentren nuevas soluciones y de esta forma llamar nuestra tercer función de actualización del conjunto de referencia. Se llama a la función *CreaSol* para generar el conjunto *P* de soluciones diversas con su matriz de datos correspondientes, después en el ciclo *mientras,* ubicado en la línea 5, se pondrán en marcha los procedimientos fundamentales del algoritmo de búsqueda dispersa, en la línea 7 si es la primer iteración del algoritmo se generara el conjunto de referencia, por lo cual se llamara a la función *ActRefSet1*, con la cual obtendremos

el conjunto de referencia ordenado como también la matriz con los datos correspondientes al *Refset*, si por el contrario no es la primer instancia entonces invocaremos a la función *ActRefSet2*, la cual se encargara de actualizar nuestro conjunto de referencia y sus datos con las nuevas soluciones. Si esta función determina que las soluciones no son nuevas ni tampoco mejores a las ya existentes en el *Refset*, prosigue a cambiar la variable bandera a cero, con lo cual si ya no hay más soluciones en la población de soluciones diversas *P* se volverá a llamar al método *CreaSol* para generar un nuevo conjunto *P* de soluciones diversa y con este conjunto utilizar la función *ActRefSet3* o si todavía hay soluciones no utilizadas en *P* usar la función *ActRefSet3* la cual nos sirve para sustituir la mitad baja del conjunto de referencia y su matriz de datos con nuevas y probablemente peores soluciones, como también cambiar nuestra bandera a uno. Continuando en la línea 22 se llama a la función *SubGenera o SubGeneraM o SubGeneraTri* para de esta forma obtener nuestros subconjuntos que serán combinados por el método *MetCombHibus, MetCombHibusTri o MetCombinaCruz* en la línea 23, con las nuevas soluciones obtenidas de la combinación de subconjuntos se proseguirá mejorarlas por lo cual se inicializa un ciclo que se encargará de mejorar todas las soluciones nuevas en la línea 25 en este ciclo *mientras* todas las soluciones trataran de ser mejoradas por la función *MetMejora*. Después de ser procesadas por este método las soluciones son agregadas a una matriz temporal como también sus datos, en esta matriz no puede haber soluciones iguales por lo que se utiliza la función *comparacion* para cerciorase de esto, con esta matriz se utiliza la función *ActRefSet2* que determina si son nuevas y mejores soluciones, de lo contrario se utilizara el método *ActRefSet3*, este proceso se realizara un cierto número de iteraciones, al finalizar el ciclo la función regresa la mejor solución del conjunto de referencia y sus datos a la función *main* que se describe a continuación

```
1  FUNCIÓN main ():
2       Hacer colores = 2, total_colores = 20, numvertices = 100, tamP = 100, tamRefset = 10
3       Inicializar listas vacías coloración, datos_coloracion, tabla_coloracion, tabla_datos
4       Hacer distancia una variable global
5       distancia = lee ()
6       coloracion, datos_coloracion= MetGenaraSolAletorias (numvertices, 1, distancia, 0)
7       Agregar coloracion a tabla_coloracion
8       Agregar datos_coloracion a tabla_datos
9       MIENTRAS (colores <= total_colores):
10            coloracion, datos_coloracion = BusquedaDispersa (colores, numvertices,
        tamP, tamRefset, distancia, tabla_datos[colores-2][Solidez])
11            Agregar coloracion a tabla_coloracion
12            Agregar datos_coloracion a tabla_datos
13            Hacer colores += 1
14      FINMIENTRAS
```

15 *tiempo = (time.time() - start_time*)
16 ***FINFUNCIÓN***

La función *main* se encarga de inicializar el algoritmo y es donde se determinan los parámetros, se estipula el número total de colores que se buscará, el número de vértices que tienen las soluciones, el tamaño del conjunto *P* de soluciones diversa y el conjunto de referencia, se inicializan las matrices que contendrán las mejores soluciones para cada coloración y sus datos, en la línea 5 se utiliza la función *lee* para obtener las distancias entre todos los vértices y almacenarla en una matriz para su uso a lo largo del programa. En la línea 6 se llama a la función *MetGenaraSolAletorias* en lugar de la función *CreaSol* dado que para la coloración de un solo color es innecesario generar un conjunto *P* de soluciones diversa, ya que el resultado de la dureza será el mismo porque su cálculo es la sumas de las distancias de los vértices con el mismo color, por esta misma razón los colores que funge como contador y parámetro se inicializa en 2, la solución es almacena en la matriz *tabla_coloracion* y sus datos en *tabla_datos*, en el ciclo *mientras* en la línea 10 se llama a la función *BusquedaDispersa* para obtener la mejor solución con sus datos para cada color los cuales serán almacenados en *tabla_coloracion* y *tabla_datos* respectivamente y de esta forma poder determinar que coloración es la adecuada, para finalizar en la línea 15 se obtiene el tiempo de ejecución del programa

3.9 Funciones para las instancias de coloración robusta

```
1  FUNCIÓN intersecciones(numvertices):
2       Hacer aux_interext = 0 y aux_interno = 0
3       Inicializar listas vacías intersec y tempinter
4       valor_inter = numvertices²
5       MIENTRAS (aux_interext < numvertices):
6            MIENTRAS (aux_interno < numvertices):
7                 SI (distancia[aux_interext][aux_interno] igual a valor_inter):
8                      Agregar el vértice aux_interno a la lista tempinter
9                      Hacer aux_interno += 1
10                SINO
11                     Hacer aux_interno +=1
12                FINSINO
13           Agregar la lista tempinter a la lista intersec
14           Hacer aux_interext += 1 y aux_interno = 0
15           Vaciar la lista tempinter
16      FINMIENTRAS
17      Regresar intersec
```

```
18 FINFUNCIÓN
```

La función *intersecciones* se encarga de encontrar para cada vértice los vértices con los que intersecta, es decir, los vértices que tienen valores extremadamente altos representados en la línea 4, es una herramienta que nos servirá para mejora el método de generación de soluciones y el método de mejora para las instancias de coloración robusta.

```
1 FUNCIÓN ChecaIntersec (chec_vert, checa_sol):
2       Hacer lim_chec = tamaño de la lista interseccion para el vértice chec_vert y cont_chec = 0
3       Inicializar lista vacía temp_chec
4       MIENTRAS (cont_chec < lim_chec):
5               color_checa = checa_sol[ interseccion[chec_vert][cont_chec] ]
6               SI (color_checa se encunetra en la lista temp_chec o color_checa es igual a None):
7                       Hacer cont_chec += 1
8               FINSI
9               SINO
10                      Agregar colores_checa - 1 en la lista temp_chec
11                      Hacer cont_chec += 1
12              FINSINO
13      Regresa temp_chec
14 FINFUNCIÓN
```

Esta función se encarga de obtener lo colores con los que se encuentran pintados los vértices que intersectan al vértice *chec_vert*, es por eso que utiliza la matriz *interseccion* que es generada por la función ***intercecciones*** y contiene todas las intersecciones para cada vértice. La matriz está declarada como global para que pueda ser utilizados por todo el programa, en la línea 5 obtiene los colores que tienen las intersecciones del vértice *chec_vert* y en la línea 10 agrega los colores a la lista *temp_chec* si es un color que no está ya en la lista, regresa esta lista de colores.

```
1 FUNCIÓN SelecColor (nocolores, colores):
2       Generar una lista de colores que no se encentren en la lista nocolores y guardarla en list_colores
4       Regresa list_colores
5 FINFUNCIÓN
```

La función ***SelecColor*** tiene como objetivo regresar una lista de colores disponibles para usar, esto significa eliminar los colores con los cuales ya están pintados los vértices con los que

intersecta un vértice. En *Python* este procedimiento es relativamente sencillo: primero se crea una lista con todos los colores que se pueden utilizar usando la siguiente instrucción *list_colores = list(xrange(colores)),* después utilizando la propiedad de comprensión de listas se eliminan los colores que ya están ocupados por los vértices con los que intersecta nuestro vertice usando la siguiente instrucción *list_colores = [cont_color for cont_color in list_colores if cont_color not in nocolores]*, se regresa la lista *list_colores*.

```
1 FUNCIÓN MetGeneraSolRobus (numvertices, colores, solidez_ante):
2       Inicializa lista vacía gdatos
3       Inicializar la lista con tamaño igual numvertices sol_robust con elementos None
4       Inicializar lista list_vert con todos los vértices disponibles
5       DESDE conrobus = 1 a numvertices
6             Seleccionar un vértice aleatorio de la lista de vértices list_vert y almacenarlo en
rand_vert
7             colores_ocupados = ChecaIntersec (rand_vert, sol_robust)
8             SI ((Tamaño de la lista colores_ocupados es igual a colores) o (Tamaño de la
lista colores_ocupados es igual a 0)):
9                   Seleccionar un color aleatorio de entre todos los colores y guárdalo en la
lista sol_robust en su posición rand_vert
10            FINSI
11            SINO:
12                  colores_posibles = SelecColor (colores_ocupados, colores)
13                  Seleccionar un color aleatorio de la lista colores_posibles y guárdalo en
la lista sol_robust en su posición rand_vert
14            FINSINO
15            Quitar el vértice rand_vert de la lista list_vert
16      FINDESDE
17      Agregar CalDureza (sol_robust, numvertices, colores) a gdatos
18      Agregar CalcSolidez (numvertices, colores, gdatos[Dureza]) a gdatos
19      SI (colores igual a 1):
20            Agregar el valor de resiliencia cero a gdatos
21      FISI
22      SINO:
23            Agregar CalcResiliencia (solidez_ante, gdatos[Solidez]) a gdatos
24      FINSINO
25      Regresa sol_robust y gdatos
21 FINFUNCIÓN
```

La función de generación de soluciones robustas cambia en comparación la utilizada para las instancias anteriores, esto se debe a las limitantes de las instancias de coloración robusta, por lo que las soluciones se generan de esta forma se escoge un vértice al azar como en la línea 6, para este vértice se buscan en la matriz los vértices con los que intersecta y sus colores correspondientes. Este paso se realiza en la línea 7, si ya todos los colores han sido ocupados, el vértice se pinta con un color aleatorio esto se ejecuta en la línea 8 y 9, de lo contrario se selecciona un color de maneara azarosa de entre los colores que no causen conflicto con los vértices intersectados este procedimiento se hace de la línea 11 a la línea 14, esto se realiza para todos los vértices, por lo cual se usa el ciclo desde que va de la línea 5 a la 16, después de estos procedimientos el proceso para obtener los datos de dureza, solidez y resiliencia son los mismo que en el método de generación de soluciones usado para las instancias anteriores.

1 FUNCIÓN Muta1colorRobus (numvertices, colores, temMuta1):
2 Seleccionar un vértice aleatorio y almacenarlo en *muta_rand*
3 *muta_ocupado* = ***ChecaIntersec*** (muta_rand, temMuta1)
4 ***SI*** ((Tamaño de la lista *muta_ocupado* es igual a *colores*) o (Tamaño de la lista *muta_ocupado* es igual a 0)):
5 Seleccionar un color aleatorio de entre todos los colores y guárdalo en la lista *temMuta1* en su posición *muta_rand*
6 ***FINSI***
7 ***SINO***:
8 *muta_posibles* = ***SelecColor*** (muta_ocupado, colores)
9 Seleccionar un color aleatorio de la lista *muta_posibles* y guárdalo en la lista *temMuta1* en su posición *muta_rand*
10 ***FINSINO***
11 Regresa temMuta1
12 ***FINFUNCIÓN***

Esta función remplazaría en el método de mejora a la función ***Muta1color*** a diferencia de esta función ***Muta1colorRobus*** utiliza la estrategia descrita en el método ***MetGeneraSolRobus*** para de esta forma tener mejores soluciones y converger más rápido con la mejor solución.

Capítulo 4
Instancias a utilizar

Se crearon 4 instancias para probar nuestro algoritmo, estas instancias son matrices seudo-aleatorias, que fueron generadas al igual que nuestro algoritmo en el lenguaje de programación *Python* en su versión 2.7. Las matrices fueron creadas de tal manera que nuestros resultados fueran controlados para de esta forma poder probar el comportamiento de nuestro algoritmo y demostrar los conceptos básicos del problema de coloración de gráficas suaves.

Las matrices fueron rellenadas de la siguiente manera: se generó una lista ordenada de menor a mayor con valores aleatorios entre de (0,1), el tamaño de la lista es igual al número de casillas de las matrices, los valores más pequeños son agrupados de manera aleatoria en subconjuntos que representan el número de colores adecuados para cierta instancia por lo cual estos subconjuntos son agrupados en las diagonales de cada instancia como se puede observar en la Fig. 4.1 y en la Fig. 4.2, de esta forma obtener los valores más altos de resiliencia cuando se presente este número de colores.

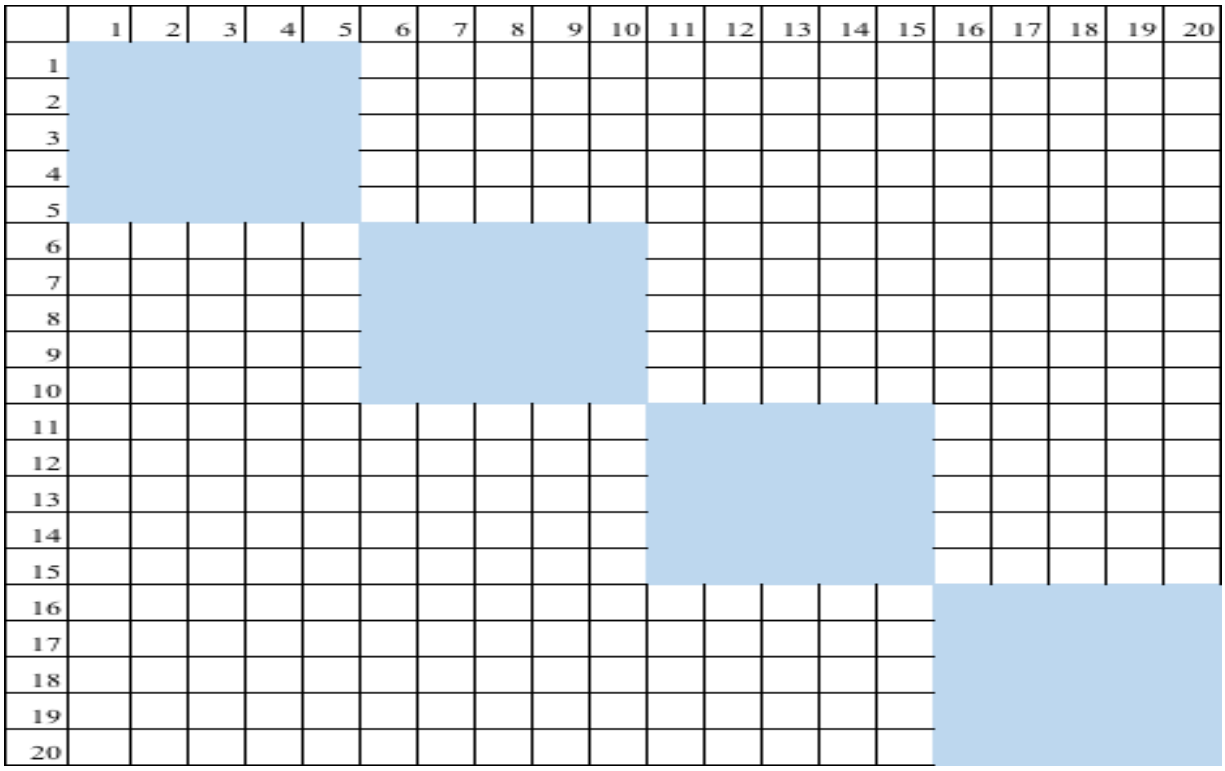

Figura 4.1. Generación de matriz de 20x20 para un óptimo de 4 colores.

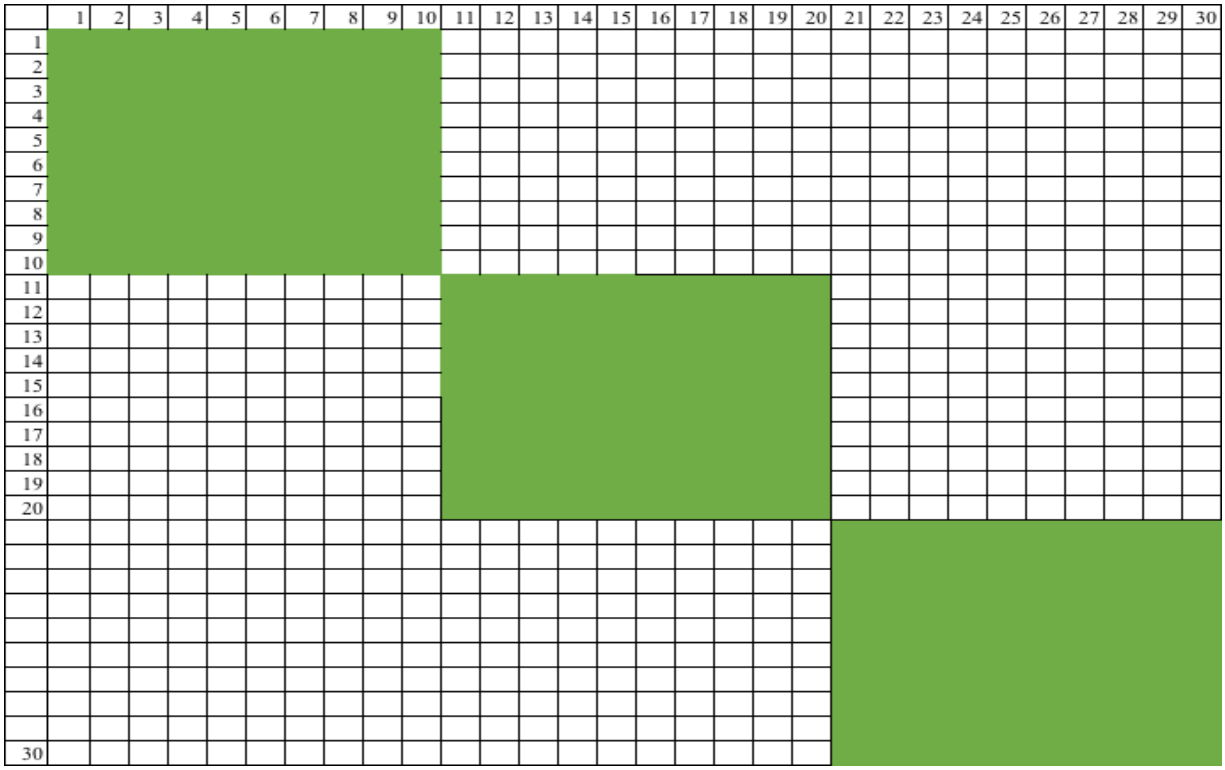

Figura 4.2. Generación de matriz de 30x30 para un óptimo de 3 colores.

Se crearon una matriz de 20x20 para un óptimo de 4 colores, una de 30x30 para un óptimo de 3 colores, una de 60x60 para un óptimo de 6 y 3 colores y una matriz de 100x100 para un óptimo de 20, 10 y 50 colores. Para el caso de las matrices de 60x60 y de 100x100 que tiene más de un óptimo de colores, se generaron subconjuntos para cada óptimo de colores, para el caso de la matriz de 60 x 60 para un óptimo de 3 y 6 colores, se llenaron primero las diagonales con 3 subconjuntos de tamaño 20x20 que representa el óptimo de 3 colores pintado de color rojo, después con 6 subconjuntos de 10x10 para el óptimo de 6 colores pintado de color verde que abarca el rojo como se puede ver en la Fig. 4.3 y para el caso de la matriz de 100x100 para un óptimo de 20, 10 y 5 colores, se llenaron primero las diagonales con 20 subconjuntos de tamaño 5x5 que representa el óptimo de 20 colores pintado de colore azul, después con 10 subconjuntos de 10x10 para el óptimo de 10 colores pintado de verde que abarca al azul y al final con 5 subconjuntos de 20x20 para un óptimo de 5 colores pintado de naranja que abarca al azul y verde, como se aprecia en la Fig. 4.4.

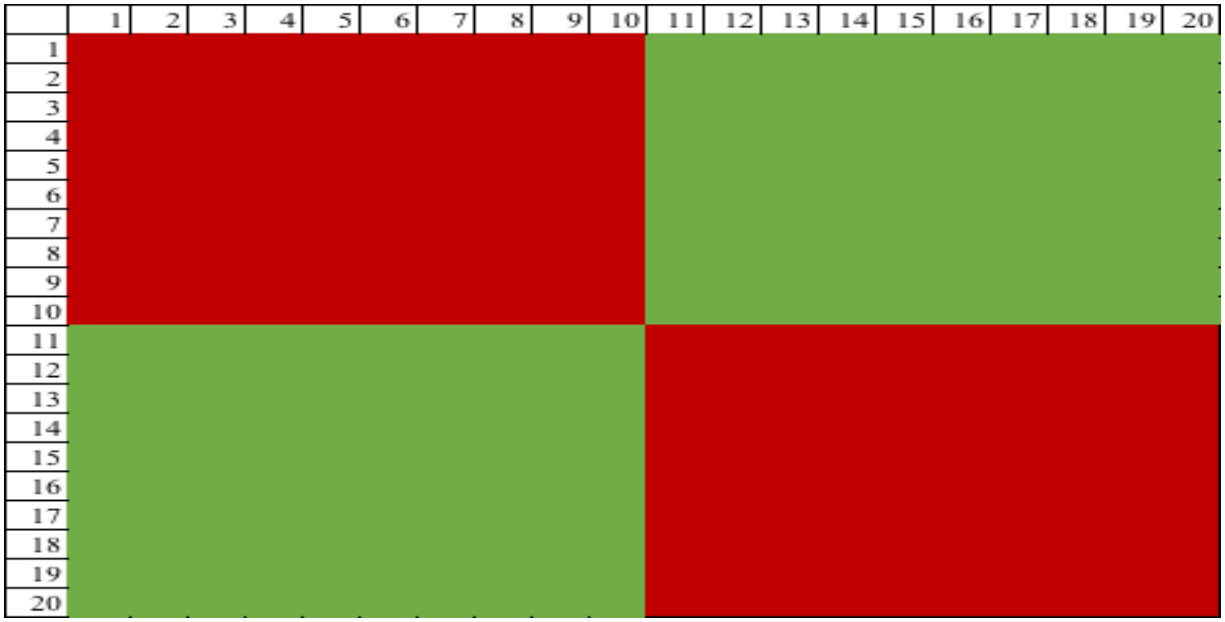

Figura 4.3. Generación de matriz de 60x60 para un óptimo de 6 y 3 colores.

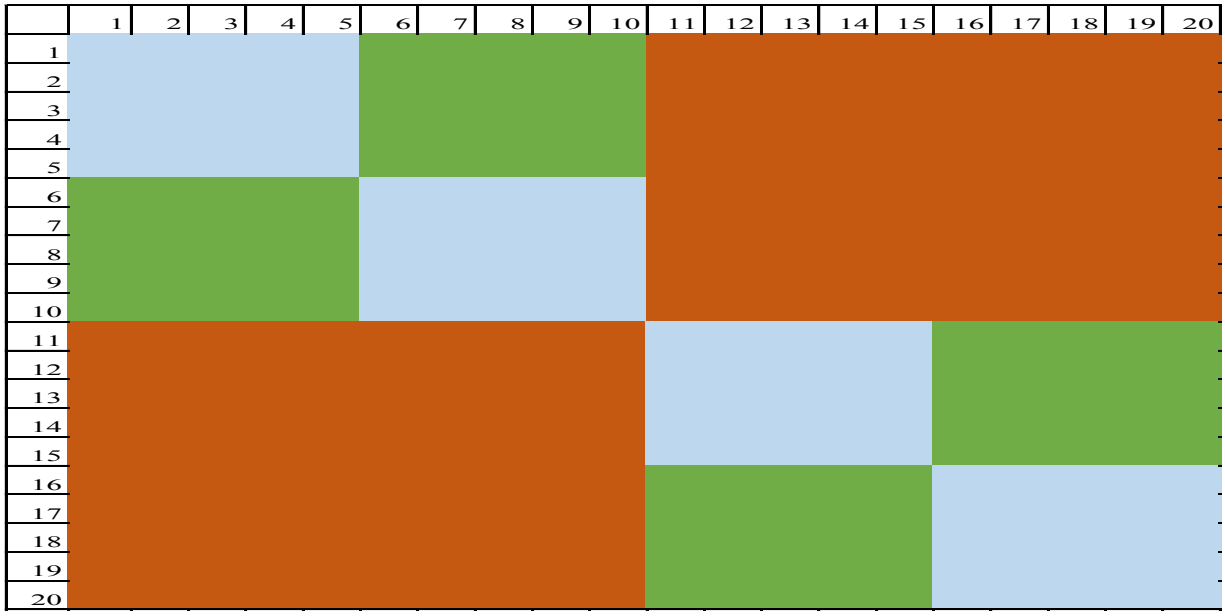

Figura 4.4. Generación de matriz de 100x100 para un óptimo de 20, 10 y 5 colores.

Las instancias de coloración robusta fueron generadas de manera equivalente a las propuestas en (Ramírez Rodríguez et al. 2003) usando las instancias de 20 a 100 vértices. Las instancias originales estaban por 10 elementos por renglón, por lo que en nuestro caso se cuadraron para que fueran el número de vértices, elementos por renglón y elementos por columna, también como se explicó en el capítulo 1, los valores que equivalen a uno fueron sustituidos por el $numero\ de\ vertices^{\,2}$.

Capítulo 5
Análisis de Resultados

La primera parte de nuestras pruebas se enfoca en poner a prueba el algoritmo de búsqueda dispersa puro, tal como lo plantea Fred Glover en 1998 (Glover 1998) para probar de esta forma sus limitantes. También se implementa el algoritmo utilizado los métodos recomendados, como se mencionó anteriormente y se utilizan las instancias descritas para las pruebas.

Como se ha mencionado con anterioridad, el algoritmo de búsqueda dispersa tiene como cualidad la flexibilidad de sus métodos, por lo cual para la segunda parte de las pruebas es fundamental encontrar las mejores variantes para cada método y de esta forma obtener un algoritmo que entregue resultados de calidad en un tiempo de ejecución aceptable. Para realizar la comparación entre las variantes del algoritmo se utilizó el análisis de varianza (ANOVA) con el apoyo del software Minitab, para de esta forma encontrar si las variantes presentaban resultados significativamente diferentes o iguales y de esta forma tomar una decisión de cual es mejor. Para todas las pruebas se realizaron 25 iteraciones.

Todas las variantes del algoritmo fueron implementadas en el lenguaje de programación Python 2.7, y los resultados fueron obtenidos con el siguiente hardware: Procesador 2.6 GHz Intel Core i7, 8 Gb de memoria RAM DDR3, con el sistema operativo macOS 10.12.1. Las pruebas ANOVA fueron realizadas utilizando el software Minitab 17.1, en las cuales se compararon las durezas obtenidas (coloración) y los tiempos de ejecución de cada variante.

5.1 Matriz de 20x20 para 4 colores

Al ser la instancia más pequeña propuesta, no representa un problema para el algoritmo de búsqueda dispersa, pero sirve para mostrar cómo funcionan los conceptos básicos de coloración de gráficas suaves, cundo tenemos una coloración que no es adecuada nuestros valores de dureza y solidez serán altos, por su parte la resiliencia tendrá valores pequeños, al ir incrementado el número de colores uno por uno y acercarnos a una coloración adecuada los valores de dureza y solidez disminuirán, mientras por su parte la resiliencia aumentara, cuando el número de colores es igual a 1 nuestra resiliencia es cero y cuando encontramos una coloración adecuado y agregamos otro color el valor de la resiliencia disminuirá drásticamente.

TABLA 5.1
RESULTADOS OBTENIDOS DE LA MATRIZ DE 20X20 PARA 4 COLORES

Nº Colores	Dureza	Solidez	Resiliencia
1	108.94725	0.573407	0
2	39.349267	0.437214	0.311501
3	20.079337	0.354341	0.233879
4	6.661847	0.166546	1.127586

En la tabla 5.1 se observa los resultados obtenidos por el algoritmo de búsqueda dispersa, estos resultados se presentaron en las 25 iteraciones realizadas por lo cual se obtiene una desviación estándar de 0. El tiempo de ejecución promedio fue de 177.5836184 segundos, lo cual demuestra que se cumple el objetivo principal del problema de coloración de gráficas suaves esto es reducir la dureza como se observa para el caso de 4 colores. También constatamos que el planteamiento de nuestras instancias es correcto al presentar el valor de resiliencia más alto para 4 colores.

5.2 Matriz de 30x30 para 3 colores

Al igual que en la instancia anterior, los resultados que se muestran en la tabla 5.2 son los que se obtuvieron la 25 veces que se ejecutó el algoritmo, esta vez con un promedio de tiempo de ejecución de 130.3172786 segundos.

TABLA 5.2
RESULTADOS OBTENIDOS DE LA MATRIZ DE 30X30 PARA 3 COLORES

Nº Colores	Dureza	Solidez	Resiliencia
1	169.876104	0.39052	0
2	55.580111	0.264667	0.475513
3	9.555272	0.07078	2.739304

En este caso el tiempo ejecución es menor debido a que son menos colores los que se calculan, pero nos muestra que al igual que la instancia anterior, se cumple el comportamiento esperado de las instancias y el objetivo de coloración de gráficas suaves de reducir la dureza.

5.3 Matriz de 60x60 para 3 y 6 colores

Para esta instancia se incrementa el tiempo promedio de ejecución siendo 229.1007935 segundos. Al igual que en las instancias anteriores los resultados que se presentan en la tabla 5.3 son los mismos, obtenidos las 25 veces que se ejecutó el algoritmo.

TABLA 5.3
RESULTADOS OBTENIDOS DE LA MATRIZ DE 60X60 PARA 3 Y 6 COLORES

Nº Colores	Dureza	Solidez	Resiliencia
1	1200.152998	0.678053	0
2	479.490993	0.551139	0.230275
3	195.410682	0.342826	0.607636
4	144.384816	0.343773	-0.002757
5	94.133245	0.285252	0.205156
6	45.082208	0.166971	0.708393

En esta instancia se aprecia el comportamiento de la resiliencia, de tal manera que cuando se tiene 3 colores la resiliencia aumenta, pero al pasar a 4 colores vemos como la resiliencia se vuelve negativa. De esta forma se demuestra que es una peor coloración, y al llegar a 6 colores la resiliencia llega a su valor máximo, indicando, que es la coloración adecuada para esta instancia al igual que se obtiene el valor menor de dureza.

5.4 Matriz de 100x100 para 5, 10 y 20 colores

Esta es la instancia más grande propuesta, la cual por la cantidad de vértices y de colores representa el mayor reto para el algoritmo de búsqueda dispersa, En la tabla 5.4 se muestran los resultados obtenidos.

TABLA 5.4
RESULTADOS OBTENIDOS DE LA MATRIZ DE 100X100 PARA 5, 10 Y 20 COLORES

Nº Colores	Dureza	Solidez	Resiliencia
1	3559.259499	0.719042323	0
2	1532.859647	0.625656999	0.149259617
3	872.610995	0.539759378	0.15914058
4	538.487263	0.448739386	0.202834864
5	267.868045	0.281966363	0.591464247

6	223.87709	0.28580054	-0.01341557
7	179.935157	0.270870129	0.055120185
8	137.212661	0.238630715	0.135101695
9	92.676291	0.18331574	0.301746998
10	49.695086	0.110433524	0.659964593
11	45.206796	0.111747136	-0.011755216
12	41.069866	0.112008725	-0.002335437
13	36.356959	0.108652981	0.030884974
14	32.340149	0.105293508	0.03190579
15	28.128903	0.099278481	0.060587422
16	29.44675	0.112178095	-0.114992272
17	29.697228	0.121651295	-0.077871758
18	15.151705	0.06651968	0.82880156
19	19.809403	0.092933002	-0.284218962
20	7.610984	0.03805492	1.442075866

Como se observa en los resultados, muestra mejores coloraciones cuando se tienen 5, 10 y 20 colores. Es en estos colores donde las resiliencias son mayores, siendo la coloración optima cuando se tiene 20 colores. Cuando se tiene la menor dureza y la mayor resiliencia, para esta instancia el tiempo de ejecución es de 3482.39114594 segundos que es igual a 58.039 minutos, al realizar otras ejecuciones del algoritmo para esta instancia se encontró que no siempre se obtenían los valores óptimos por lo cual se prosiguió a la segunda parte de las pruebas para encontrar las mejores estrategias del algoritmo de búsqueda dispersa, de esta forma mejorar los tiempos de ejecución e incrementar la calidad de nuestras soluciones.

Debido al tiempo de ejecución de la instancia anterior, para la segunda parte de las pruebas en lugar de que el algoritmo encuentre los datos de dureza, solidez y resiliencia para todas las coloraciones posibles nos enfocaremos en el cálculo de la dureza para la coloración correspondiente a 20 colores, la cual es la más demandante.

5.5 Búsqueda dispersa: Estándar vs Renovación parcial del conjunto de referencia (RPCR)

En esta prueba se comparó el algoritmo de búsqueda dispersa puro (Estándar) (Glover 1998) en el cual está estipulado que el algoritmo terminara su ejecución cuando el método de combinación de soluciones deje de encontrar nuevas soluciones para nuestro conjunto de referencia, contra el algoritmo de búsqueda dispersa en el cual se remplaza el criterio de terminación anterior por un ciclo con cierto número de iteraciones, en el cual cuando el algoritmo no encuentra nuevas soluciones se prosigue a utilizar los métodos de generación de soluciones diversas, método de

mejora y el método de actualización del conjunto de referencia, al invocar los 2 primero métodos para generan una nueva población "*P*" y con el ultimo método remplazar la mitad baja de nuestro conjunto de referencia con soluciones diversas de "*P*" (Martí y Laguna 2003).

5.5.1. Búsqueda dispersa: Estándar vs RPCR: *Dureza*

Se compararon las durezas obtenidas por las dos variantes utilizando el análisis de varianza, los resultados obtenidos se muestran en la tabla 5.5.

TABLA 5.5
RESULTADOS OBTENIDOS POR EL ANÁLISIS DE VARIANZA REFERENTES A LA DUREZA

Factor	Número	Media	Desv. Est.	Optimo	Valor Mayor
Estándar	25	10.46	5.54	7.610984	25.6948
RPCR	25	8.322	0.975	7.610984	10.621212

La dureza optima es de 7.610984, ya que va de la mano con la coloración correcta para 20 colores, eso significa que tenemos exactamente 5 vértices pintados con cada color, como se muestra, las dos variantes obtienen este resultado, pero no el 100% de las veces. Con la información anterior nos percatamos que la variante RCPR presenta resultados más cercanos al óptimo ya que su desviación estándar es menor y su valor más alto es menor que los obtenidos por el algoritmo estándar. Esto se pude observar mejor en la Fig. 5.1.

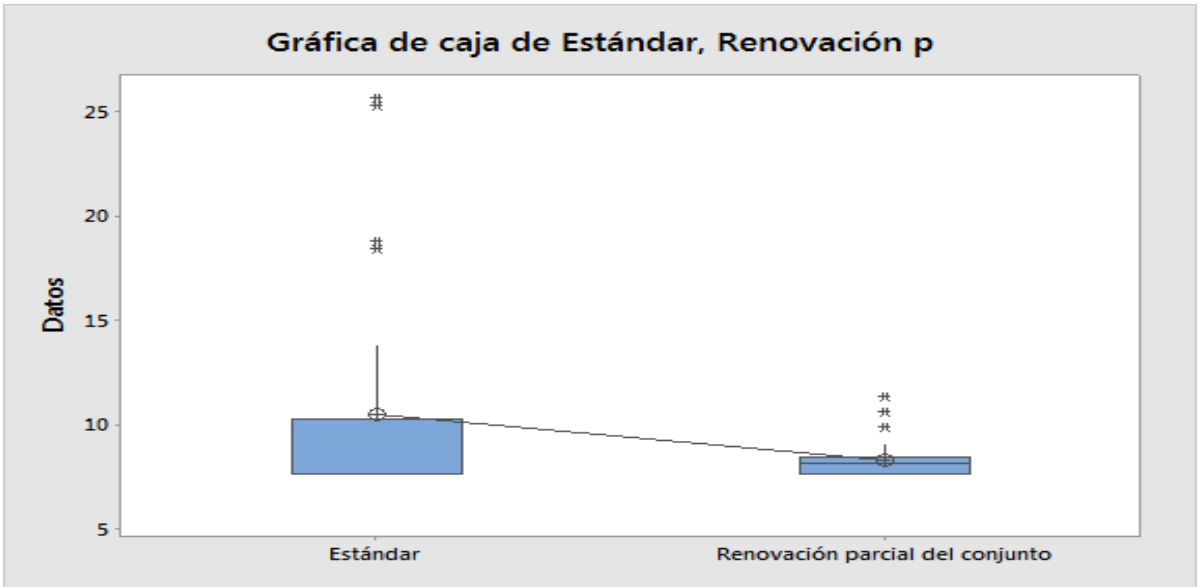

Figura 5.1. Comparación de los intervalos referentes a la dureza entre Estándar y RPCR.

Con la información de la tabla 5.5 y la Fig. 5.1, podemos concluir que las durezas obtenidas por el algoritmo estándar son demasiado dispersas, por lo tanto, nos pueden dar el valor óptimo, como puede darnos valores extremadamente alejados del óptimo. Por su parte la variante del algoritmo de búsqueda dispersa con renovación parcial del conjunto de referencia nos presenta tanto una media como una desviación estándar pequeñas, por lo que nos entrega durezas más cercanas al óptimo y esto lo hace más confiable. Usado la herramienta comparativa Games-Howell del software Minitab nos indica que no hay diferencias significativas entre las dos, como para categorizarlas de manera distinta.

5.5.2. Búsqueda dispersa: Estándar vs RPCR: *Tiempo*

Con la prueba ANOVA se comparan los tiempos de ejecución en segundos del algoritmo estándar y su variante RPCR. En la tabla 5.6 y en la Fig. 5.2 se muestran los resultados obtenidos.

TABLA 5.6
RESULTADOS OBTENIDOS POR EL ANÁLISIS DE VARIANZA REFERENTES AL TIEMPO

Factor	Número	Media	Desv. Est.
Estándar	25	257	51.3
RPCR	25	159.41	12.48

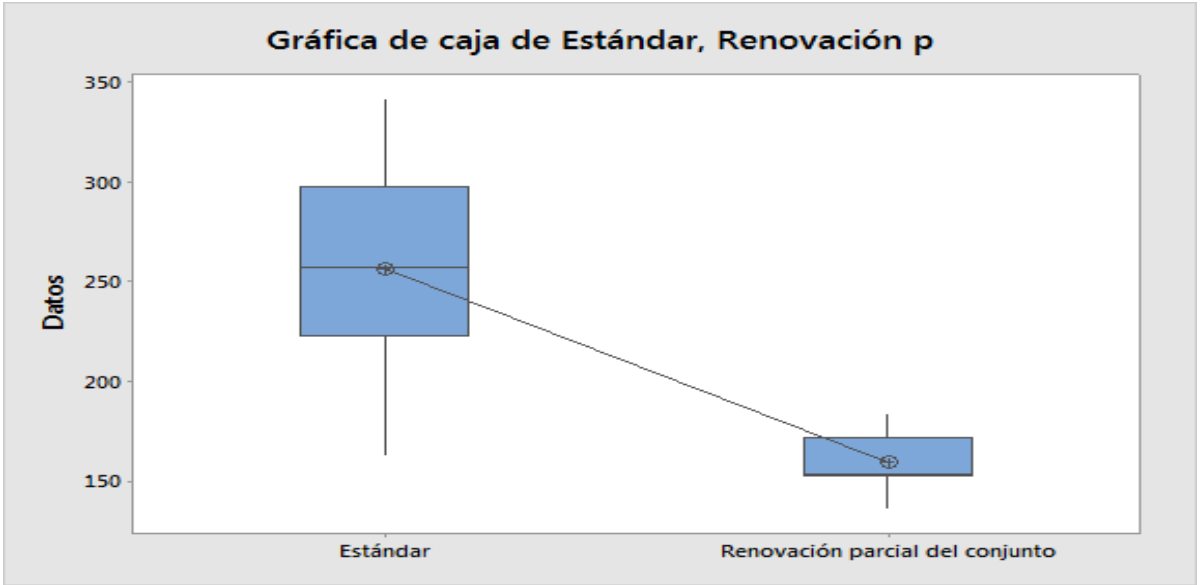

Figura 5.2. Comparación de los intervalos referentes al tiempo entre Estándar y RPCR.

Como se puede observar, la diferencia entre el algoritmo estándar y la variante RPCR referente al tiempo es notable favoreciendo a la variante con RPCR, siendo esta última 97.59 segundos más rápida que el algoritmo estándar. Utilizando la comparativa Games-Howell también nos indica que hay una diferencia significativa entre los dos. Con estos resultados, aunado a los resultados del análisis de varianza de la dureza, nos indicaban que no había diferencia significativa entre los dos algoritmos y que por su parte el algoritmo estándar tiende a presentar resultados extremos, se concluye que el algoritmo de búsqueda dispersa con renovación parcial del conjunto de referencia, es un algoritmo más rápido y más confiable por lo que para las pruebas siguientes este será el algoritmo base.

5.6 Método de combinación de soluciones

En este apartado se comparan las variantes que se pueden utilizar en el método de combinación de soluciones del algoritmo de búsqueda dispersa con ciclo. En la literatura se sugiere el uso del método de cruzamiento utilizado comúnmente en los algoritmos genéticos. Mismo que se caracteriza por combinar dos soluciones a través de un valor aleatorio, de la primera solución se toman los elementos de la posición inicial hasta el valor aleatorio, de la segunda solución se toman los elementos desde el valor aleatorio hasta la posición final y con estas 2 partes se genera una nueva solución, se planteó el uso de un hibrido entre búsqueda local y mutación (local-híbrido), el cual consiste en usando dos soluciones generar una nueva solución de la siguiente manera: se genera un número aleatorio entre (0,1) si el número aleatorio es menor a .5 se toma el elemento de la primera solución de lo contrario se toma el elemento de la segunda solución así sucesivamente para cada elemento de la nueva solución.

5.6.1. Método de combinación de soluciones: Dureza

Al comparar los datos obtenidos por cada una de las variantes con la prueba ANOVA se obtuvieron los siguientes resultados mostrados en la tabla 5.7 y la Fig. 5.3.

TABLA 5.7
RESULTADOS OBTENIDOS POR EL ANÁLISIS DE VARIANZA REFERENTES A LA DUREZA

Factor	Número	Media	Desv. Est.	Optimo	Valor Mayor
Cruzamiento	25	8.322	0.975	7.610984	11.36157
Local_hibrido	25	8.174	1.069	7.610984	11.183382

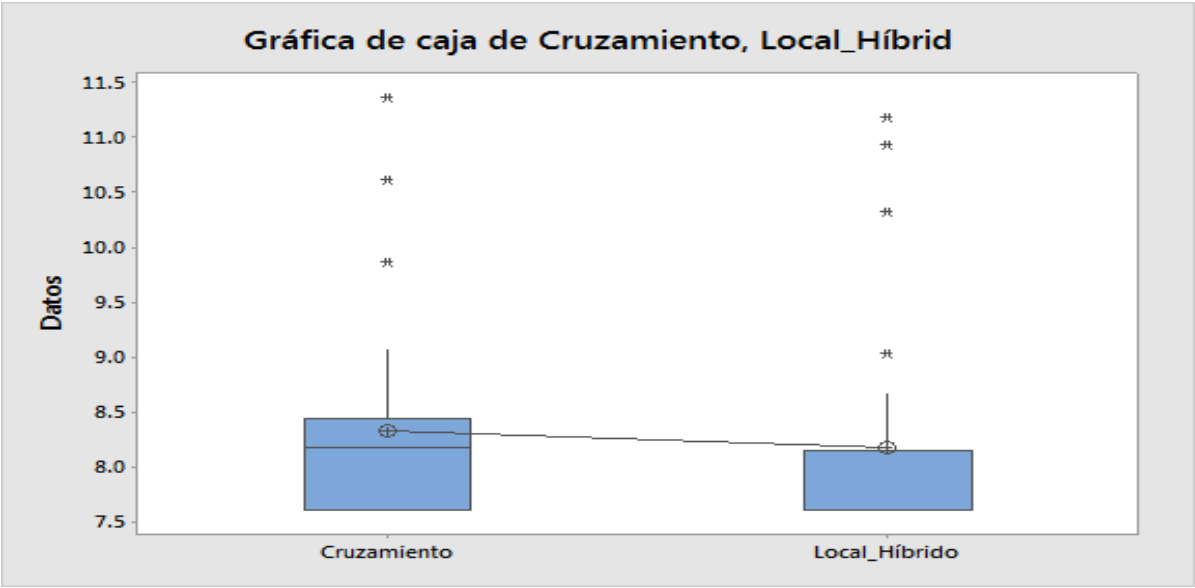

Figura 5.3. Comparación de los intervalos referentes a la dureza entre Cruzamiento y Local_hibrido.

Como se puede observar la diferencia entre los dos métodos es apenas perceptible. Al analizar los resultados con el método Games-Howell también llega a la misma conclusión que no hay una diferencia significativa entre los dos métodos.

5.6.2. Método de combinación de soluciones: Tiempo

Utilizando el análisis de varianza para comparar los tiempos de ejecución obtenidos por las dos variantes obtenemos los siguientes resultados mostrados en la tabla 5.8.

TABLA 5.8
RESULTADOS OBTENIDOS POR EL ANÁLISIS DE VARIANZA REFERENTES AL TIEMPO

Factor	Número	Media	Desv. Est.
Cruzamiento	25	159.41	12.48
Local_hibrido	25	141.59	13.75

Con estos resultados se puede apreciar que la variante "*local_hibrido*" es más rápida que usar "*cruzamiento*", utilizando la comparativa Games-Howell nos indica que son significativamente diferentes favoreciendo a "*local_hibrido*", esto se puede apreciar mejor en la Fig. 5.4.

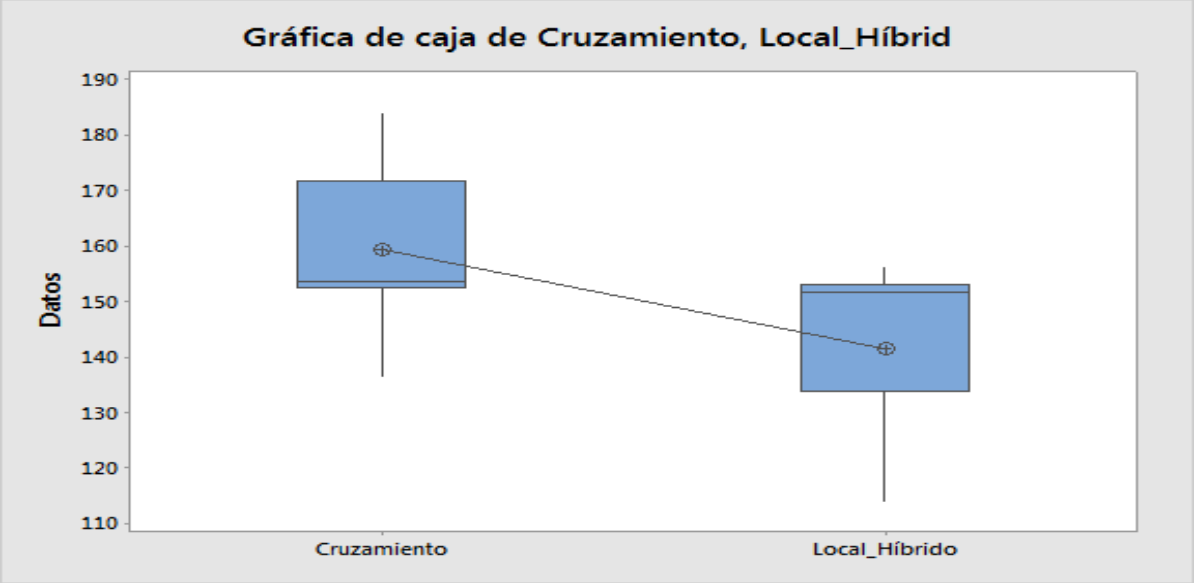

Figura 5.4. Comparación de los intervalos referentes al tiempo entre *Cruzamiento* y *Local_hibrido*.

Con los resultados referentes a la dureza se puede concluir que usar el método "*local_hibrido*" para la combinación de resultados es adecuado ya que es la variante más rápida y no compromete la calidad de los resultados al presentar resultados similares a los del método de cruzamiento. Para las siguientes pruebas se utilizará la variante "*local_hibrido*" en el método de combinación de soluciones.

5.7 Método de mejora

A continuación, compararemos diferentes estrategias para optimizar el método de mejora, se utilizará como modelo base que se cambien de color un vértice aleatoriamente (Búsqueda local simple), se comparará contra una combinación de métodos en la cual dependiendo de un valor aleatorio "L" se escoge uno de los siguientes técnicas: si "L" < 0.3 se cambia de color un vértice aleatoriamente (Búsqueda local simple), si 0.3<"L"<0.6 se cambian dos vértices de color aleatoriamente (Vecindades variables), si "L">= 0.6 se intercambia los colores de dos vértices (Intercambio). Se plantean 2 variantes más para el uso de esta combinación, en la cual se cambia el orden de las técnicas.

5.7.1. Método de mejora: Dureza

Con los datos recabados de las distintas iteraciones, se realiza la prueba ANOVA y se obtienen los siguientes resultados desplegados en la tabla 5.8.

TABLA 5.8
RESULTADOS OBTENIDOS POR EL ANÁLISIS DE VARIANZA REFERENTES A LA DUREZA

Factor	Número	Media	Desv. Est.	Optimo	Valor Mayor
Simple	25	8.174	1.069	7.610984	11.183382
Combina1	25	11.708	1.984	7.610984	17.551765
Combina2	25	11.939	2.17	7.610984	17.4209
Combina3	25	10.55	1.761	7.610984	17.551765

Como se puede apreciar, los valores que nos arroja el método simple (búsqueda local) son valores más cercanos al optimo, en comparación con el resto. Esto se comprueba con la comparación Games-Howell, en la cual entre las combinaciones de estrategias y el método simple se encuentran diferencias significativas, dicha comparación favorece al método simple. Esta diferencia se observa más notoriamente en la Fig. 5.5. en la cual nos podemos percatar que el intervalo de valores con respecto a la dureza del método simple es más pequeño y se encuentra cerca del óptimo a diferencia de las demás estrategias.

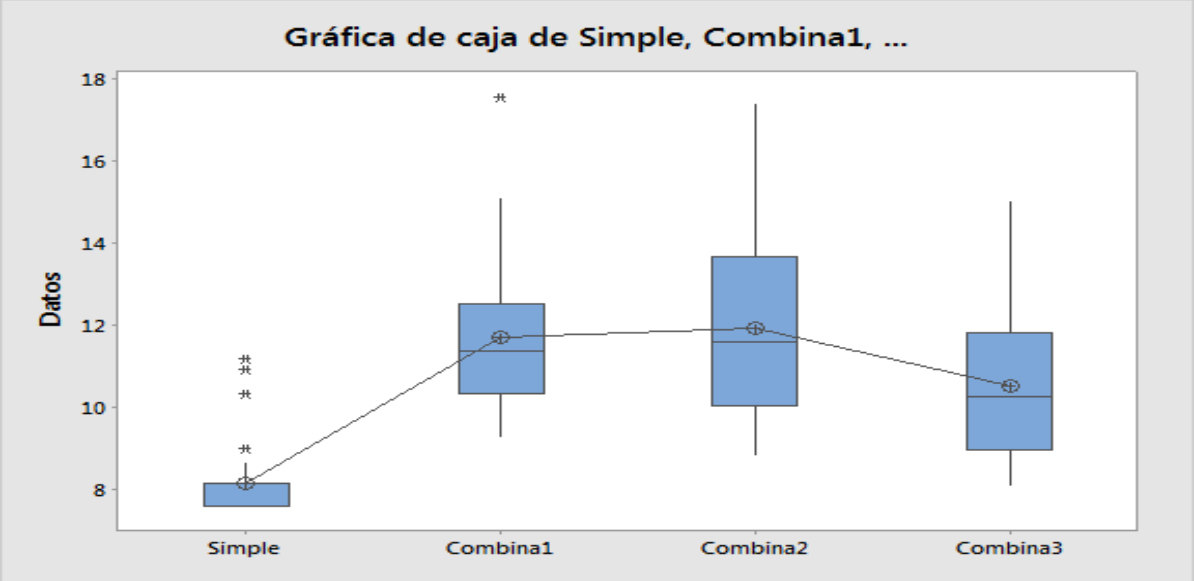

Figura 5.5. Comparación de los intervalos referentes a la dureza entré las distintas estrategias para el método de mejora.

5.7.2. Método de mejora: Tiempo

Al realizar el análisis de varianza con respecto al tiempo de ejecución para las distintas estrategias del método de mejora se obtienen los siguientes resultados en la tabla 5.9.

TABLA 5.9
RESULTADOS OBTENIDOS POR EL ANÁLISIS DE VARIANZA REFERENTES AL TIEMPO

Factor	Número	Media	Desv. Est.
Simple	25	141.59	13.76
Combina1	25	131.36	15.31
Combina2	25	135	15.65
Combina3	25	131.75	13.95

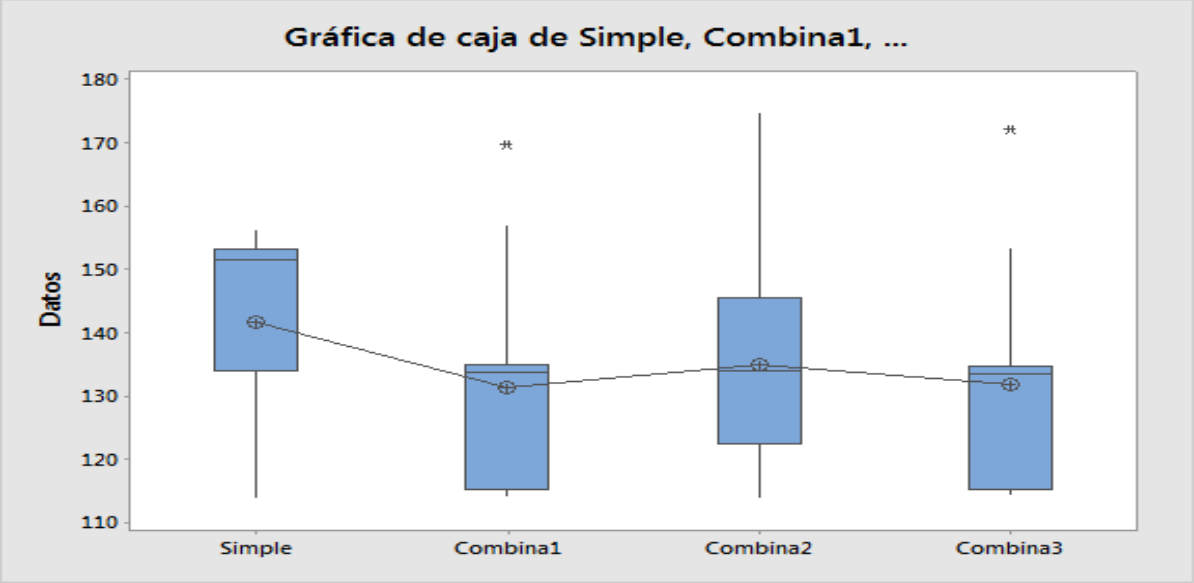

Figura 5.6. Comparación de los intervalos referentes al tiempo entré las distintas estrategias para el método de mejora.

Como podemos observar en la tabla 5.9 y la Fig. 5.6, la estrategia simple es ligeramente la más lenta de las 4, el método comparativo Games-Howell nos indica que entre las distintas estrategias respecto al tiempo para el método de mejora no hay diferencias significativas. Por lo que la estrategia simple es la mejor opción debido a que nos da mejores soluciones en un tiempo de ejecución equiparable a las demás estrategias.

5.8 Método mejora 350 iteraciones

Para las pruebas realizadas hasta el momento del algoritmo de búsqueda dispersa con la estrategia de ciclo, se han utilizado 200 iteraciones para el método de mejora y para el ciclo principal del algoritmo de búsqueda dispersa 100 iteraciones. Por las características de las estrategias se incrementará el número de iteraciones del método de mejora a 350 para de esta forma evitar que el algoritmo se quede en un óptimo local.

5.8.1. Método mejora 350 iteraciones: Dureza

Con los datos obtenidos se realizó la prueba ANOVA y se obtuvieron los siguientes resultados referentes a la dureza que se muestran en la tabla 5.10.

TABLA 5.10
RESULTADOS OBTENIDOS POR EL ANÁLISIS DE VARIANZA REFERENTES A LA DUREZA

Factor	Número	Media	Desv. Est.	Optimo	Valor Mayor
Simple	25	7.611	0	7.610984	7.610984
Combina1	25	8.841	1.799	7.610984	14.505964
Combina2	25	8.556	1.532	7.610984	13.258003
Combina3	25	8.043	0.885	7.610984	11.519531

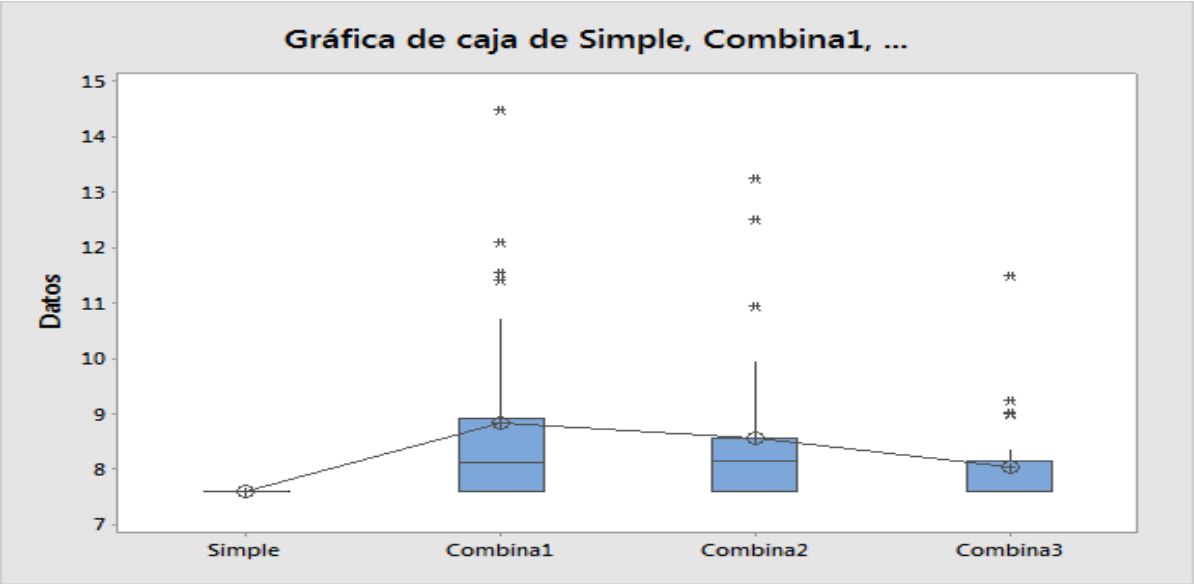

Figura 5.7. Comparación de los intervalos referentes a la dureza entré las distintas estrategias para el método de mejora.

Al incrementar el número de iteraciones del método de mejora de 200 a 350, se pude observar en la tabla 5.10 que la estrategia simple (Búsqueda local) obtiene un 100% de las 25 pruebas realizadas el valor óptimo de dureza, mientras que los métodos combinatorios reducen sus intervalos obteniendo un mayor número de veces el valor óptimo de la dureza y disminuyendo el valor máximo que obtienen, pero no logran obtener un 100% de veces el óptimo como lo logra el método simple, esto se pude ver en la Fig. 5.7.

5.8.2. Método mejora 350 iteraciones: Tiempo

El análisis de varianza de los datos con respecto al tiempo para las estrategias de mejora se aprecia en la tabla 5.11.

TABLA 5.11
RESULTADOS OBTENIDOS POR EL ANÁLISIS DE VARIANZA REFERENTES AL TIEMPO

Factor	Número	Media	Desv. Est.
Simple	25	320.03	27.15
Combina1	25	258.35	27.23
Combina2	25	234.84	23.08
Combina3	25	263.67	25.09

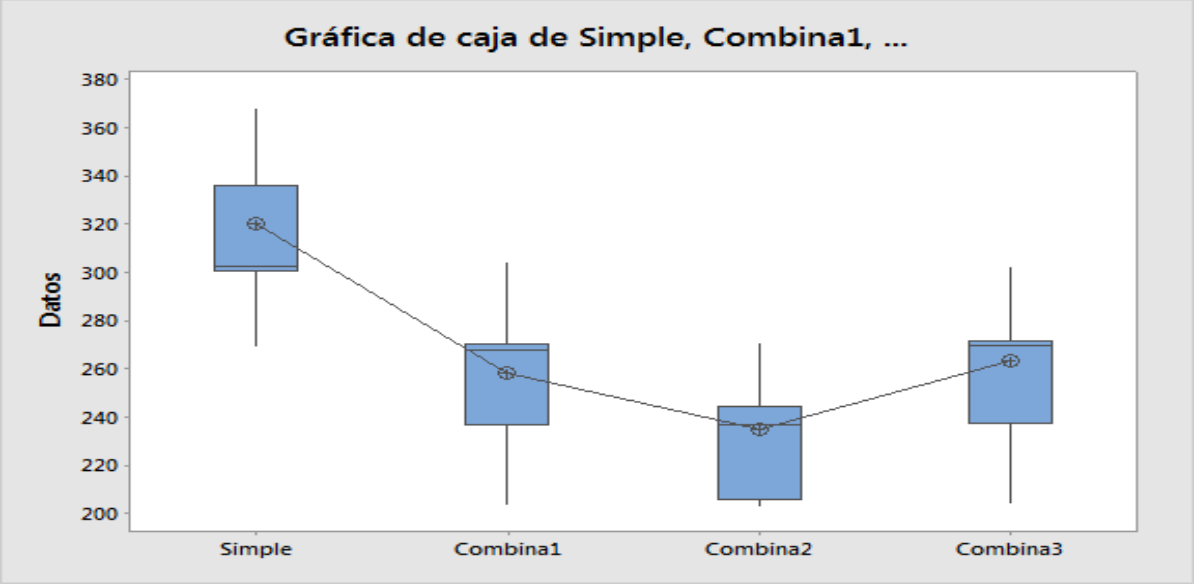

Figura 5.8. Comparación de los intervalos referentes al tiempo entré las distintas estrategias para el método de mejora.

Como se puede observar en la tabla 5.11 y la Fig. 5.8 la estrategia simple es la más lenta de las 4, mientras que de los métodos combinatorio combina 2 es la estrategia más rápida, como se puede observar en los datos obtenidos en esta sección tanto de dureza como de tiempos de ejecución el método simple (búsqueda local) es la mejor estrategia para utilizar en el método de mejora, ya que obtiene el óptimo un 100% de la veces y solamente es 56.36 segundo más lento que el método combina 3, el cual es el que presenta mejores resultados de los métodos combinatorios.

5.9 Búsqueda dispersa estándar vs búsqueda dispersa mejorada

Para finalizar comparamos el algoritmo de búsqueda dispersa estándar contra el algoritmo de búsqueda dispersa con las mejoras encontradas en las pruebas anteriores, como en el caso del algoritmo de búsqueda dispersa estándar el algoritmo mejorado no tuvo inconvenientes para encontrar la dureza, solidez, resiliencia y coloración optimas de las primeras 3 instancias, y para la instancia de 100 x 100 para 20, 10 y 5 colores los resultados se muestran a continuación en las Fig. 5.9 y 5.10.

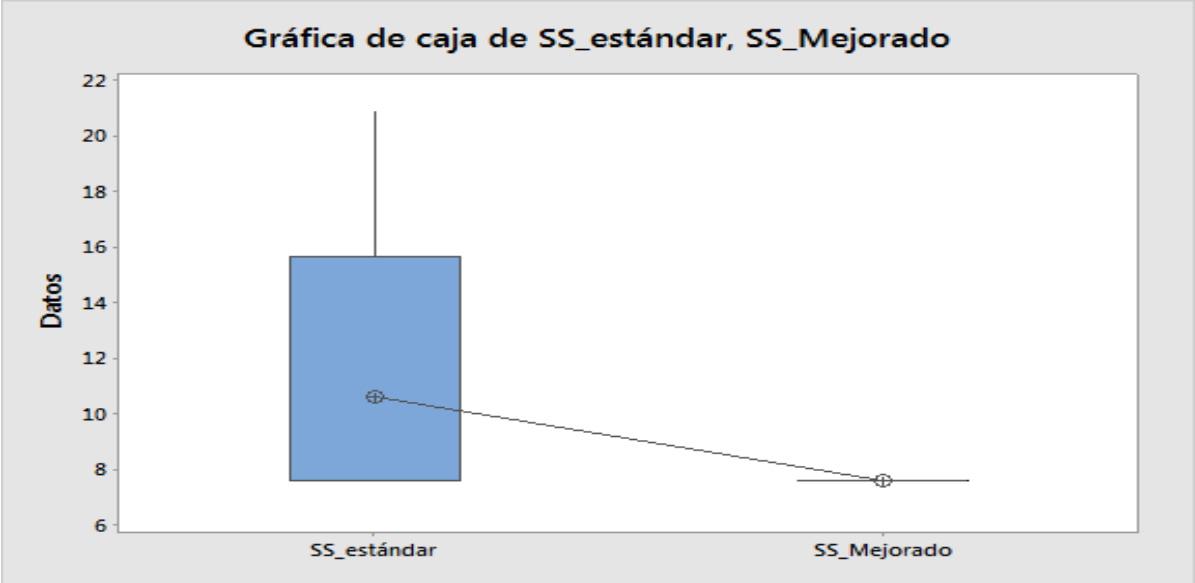

Figura 5.9. Comparación de los intervalos referentes a la dureza entré el algoritmo estándar y el mejorado de búsqueda dispersa.

Como se puede observar en las Fig. 5.9 y 5.10 el algoritmo de búsqueda dispersa mejorado obtiene la dureza optima el 100% de las 25 iteraciones realizadas por lo cual las mejoras obtenidas presentan una mejoría notable con respecto al algoritmo estándar y nos proporcionan un algoritmo más robusto y rápido.

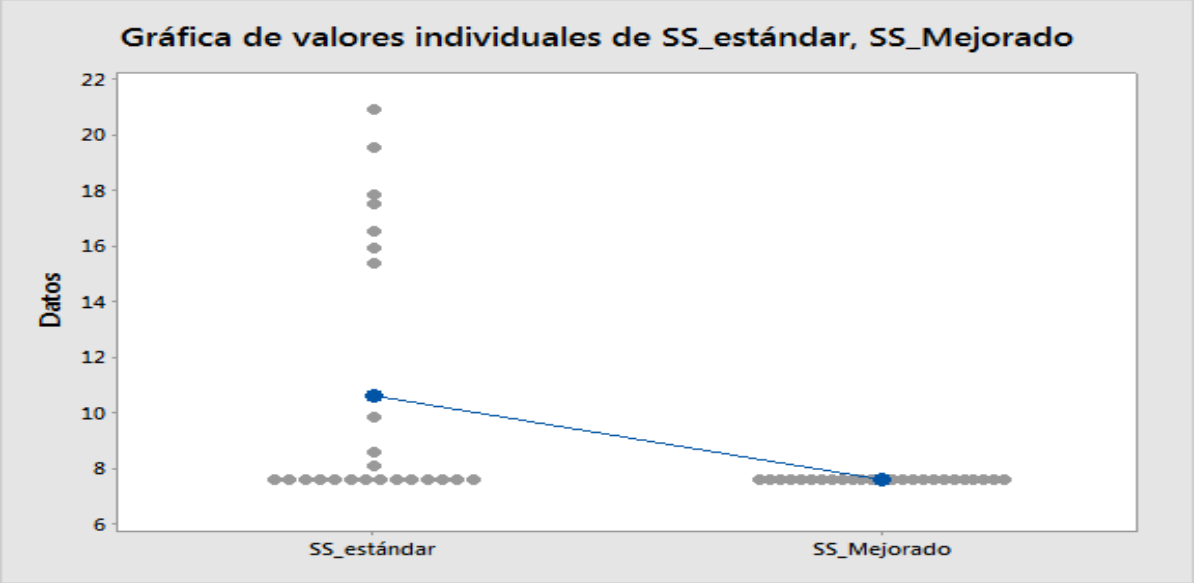

Figura 5.10. Comparación de los valores individuales entré el algoritmo estándar y el mejorado de búsqueda dispersa.

5.10 Instancias coloración robusta

Para las instancias de coloración robusta que como se mencionó en el capítulo anterior fueron tomadas de (Ramírez Rodríguez 2001) y los resultados obtenidos se compraron con los resultados obtenidos en (Ramírez Rodríguez et al. 2011), para esta instancias fue necesario aplicar distintas funciones para los métodos de generación de soluciones diversa y mejora, estas estrategias fueron ya explicadas en los capítulos 3 y 4. En la tabla 5.12 se muestran los resultados obtenidos por distintas metaheurísticas en el artículo antes mencionado, como a su vez el algoritmo de búsqueda propuesto, cabe mencionar que las metaheurísticas incluyendo una de búsqueda dispersa trabaja directamente con el problema de coloración robusta, mientras que nuestro algoritmo de búsqueda dispersa da solución al problema de coloración robusta a través del modelo de coloración de gráficas suaves, también vale la pena aclarar que el valor de rigidez de coloración robusta es igual al valor de dureza de coloración de gráficas suaves.

TABLA 5.12
Comparación del algoritmo propuesto de búsqueda dispersa con otras metaheurísticas (Ramírez Rodríguez et al. 2011) utilizadas para las instancias de coloración robusta propuestas por Ramírez (Ramírez Rodríguez 2001)

Instancias		Rigidez				Dureza
Instancia	Colores	Búsqueda tabú	GRASP	Recocido Simulado	Búsqueda Dispersa Col. Robusta	Búsqueda dispersa Col.Graf. Suaves (Algoritmo Propuesto)
Al30	10	8.0623	7.5749	7.5749	7.5749	7.5749
	11	6.0565	5.9318	5.889	5.889	5.889
Al40	14	7.1709	7.395	6.9901	7.0837	7.0837
	15	5.8173	6.3117	4.9947	5.6708	5.6708
Al50	17	9.8259	8.9531	8.2587	8.2587	8.2587
	18	7.4966	7.1464	6.7164	6.7164	6.7164
Al60	20	9.8331	9.9687	8.7568	8.8676	8.8676
	21	8.2181	8.143	7.2048	7.238	7.238
Al70	24	11.1307	11.2388	9.3555	9.2634	9.2634
	25	9.5478	9.2145	7.3799	7.7048	7.7048
Al80	27	11.1946	11.7512	9.7132	9.835	9.835
	28	10.5845	10.2631	8.6118	8.5961	8.5961
Al90	30	12.2832	13.4919	10.64	10.8911	10.8911
	31	11.3699	11.506	9.557	9.5008	9.5008
AL100	34	12.1932	12.8675	9.9658	10.047	10.047
	35	12.065	11.1317	9.0303	9.4259	9.4259

Como se puede observar en la tabla 5.12 los resultados obtenidos por el algoritmo propuesto son de la misma calidad que los mejores resultados obtenidos en el artículo (Ramírez Rodríguez et al. 2011), por lo que se muestra que el algoritmo propuesto es un algoritmo robusto, eficiente y que da soluciones de calidad, como a su vez que el modelo de coloración de gráficas suaves rectifica su eficacia para resolver otros problemas de coloración como en este caso el problema de coloración robusta.

Capítulo 6
Conclusiones

En este trabajo se realizó una investigación sobre el modelo de coloración de gráficas suaves en el que se explica en que consiste, sus fundamentos principales, los distintos problemas de coloración de gráficas que hay como también se investigaron las distintas metaheurísticas que existen para resolver los problemas de coloración de gráficas, con lo cual se propone un algoritmo de búsqueda dispersa, el cual fue implementado en el lenguaje de programación Python en su versión 2.7.

En la literatura se encontraron las mejores estrategias para cada método del algoritmo de búsqueda dispersa siendo los métodos de combinación de soluciones y de mejora en los cuales se encontraron diferencias significativas dependiendo de la estrategia utilizada. A su vez para probar estas estrategias y determinar cuáles son las mejores, se generaron instancias de matrices seudo aleatorias desde 20 a 100 vértices. En las cuales se evaluó la calidad de las soluciones como también los tiempos de ejecución, para de esta forma obtener un algoritmo que no comprometa la calidad de las soluciones con periodos de ejecución sumamente grandes y así obtener un algoritmo eficiente y que presente soluciones de alta calidad.

Como otro medio de comprobación para el algoritmo propuesto, se implementó el algoritmo de búsqueda dispersa como medio de solución para el problema de coloración de robusta utilizado el modelo de coloración de gráficas suaves, comparando los resultados con metaheurísticas que trabajaron directamente con el problema de coloración robusta y que actualmente entregan los mejores resultados, el algoritmo propuesto iguala estos resultados demostrando su eficiencia, robustez y capacidad de generar soluciones de alta calidad. Con esta comparación también se comprueba que el problema de coloración de gráficas suaves es capaz de dar solución a distintos problemas de coloración de gráficas de manera eficaz y eficiente.

Referencias

Baghel, Malti, Shikha Agrawal, y Sanjay Silakari. 2013. “Recent Trends and Developments in Graph Coloring”. En *Proceedings of the International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA)*, editado por Suresh Chandra Satapathy, Siba K. Udgata, y Bhabendra Narayan Biswal, 199:431–39. Berlin, Heidelberg: Springer Berlin Heidelberg. http://link.springer.com/10.1007/978-3-642-35314-7_49.

Bensouyad, Meriem, y Djamel Eddine Saidouni. 2015. “A Hybrid Discrete Flower Pollination Algorithm for Graph Coloring Problem”. En , 1–6. ACM Press. doi:10.1145/2832987.2833022.

Bessedik, M., R. Laib, A. Boulmerka, y H. Drias. 2005. “Ant Colony System for Graph Coloring Problem”. En *Proceedings of the 2005 International Conference on Computational Intelligence for Modelling*, 1:786–91. IEEE. doi:10.1109/CIMCA.2005.1631360.

Bouzidi, Morad, y Mohammed Essaid Riffi. 2014. “Discrete novel hybrid particle swarm optimization to solve travelling salesman problem”. En , 17–20. El Jadida: IEEE. doi:10.1109/WCCCS.2014.7107912.

De los Cobos Silva, Sergio Gerardo, John Goddard Close, Miguel Ángel Gutiérrez Andrade, Alma Edith Martínez Licona, y Universidad Autónoma Metropolitana. 2010. *Búsqueda y exploración estocástica*. México, D.F.: Universidad Autónoma Metropolitana, Unidad Iztapalapa.

Diestel, R. 2000. *Graph Theory*. New York, NY: Springer-Verlag.

Djelloul, Halima, Sara Sabba, y Salim Chikhi. 2014. “Binary bat algorithm for graph coloring problem”. En , 481–86. IEEE. doi:10.1109/ICoCS.2014.7060988.

Fahim, Alaa, y Abdel-Rahman Hedar. 2014. “Hybrid scatter search for integer programming problems”. En , ORDS – 61 – ORDS – 69. IEEE. doi:10.1109/INFOS.2014.7036698.

Feo, T. A., y MGC Resende. 1995. “Greedy Randomized Adaptative Search Procedures”. *Journal of Global Optimization*, 109–33.

Flores Cruz, Jorge, Pedro Lara Velázquez, Miguel Ángel Gutiérrez Andrade, Sergio G. De los Cobos Silva, y Eric Alfredo Rincón García. s/f. “Un sistema Clasificador Utilizando Coloracioón de Gráficas Suaves”. *Revista de Matemática: Teoría y Aplicaciones*. http://revistas.ucr.ac.cr/index.php/matematica/article/view/20838.

Glover, Fred. 1989. “Tabu Search—Part I”. *ORSA Journal on Computing* 1 (3): 190–206. doi:10.1287/ijoc.1.3.190.

———. 1998. “A template for scatter search and path relinking”. En *Artificial Evolution*, editado por Jin-Kao Hao, Evelyne Lutton, Edmund Ronald, Marc Schoenauer, y Dominique Snyers, 1363:1–51. Berlin, Heidelberg: Springer Berlin Heidelberg. http://link.springer.com/10.1007/BFb0026589.

Glover, Fred, y Manuel Laguna. 2007. “Principles of tabu search”, 1–12.

Holland, John H. 1992. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. 1st MIT Press ed. Complex adaptive systems. Cambridge, Mass: MIT Press.

Hvattum, Lars Magnus, Abraham Duarte, Fred Glover, y Rafael Martí. 2013. "Designing Effective Improvement Methods for Scatter Search: An Experimental Study on Global Optimization". *Soft Computing* 17 (1): 49–62. doi:10.1007/s00500-012-0902-9.

Jain, Anil K, Robert PW Duin, y Jianchang Mao. 2000. "Statistical pattern recognition: A review". *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 22 (1): 4–37.

Kirkpatrick, S., C. D. Gelatt, y M. P. Vecchi. 1983. "Optimization by Simulated Annealing". *Science* 220 (4598): 671–80. doi:10.1126/science.220.4598.671.

Kuhn, F. 2009. "Weak graph colorings: distributed algorithms and applications". *SPAA'09*, 138–44.

Laguna, Manuel, y Viníicius A. Armentano. 2005. "Lessons from Applying and Experimenting with Scatter Search". En *Metaheuristic Optimization via Memory and Evolution*, editado por Ramesh Sharda, Stefan Voß, César Rego, y Bahram Alidaee, 30:229–46. Boston: Kluwer Academic Publishers. http://link.springer.com/10.1007/0-387-23667-8_10.

Lai, Xiangjing, Jin-Kao Hao, y Fred Glover. 2015. "Backtracking Based Iterated Tabu Search for Equitable Coloring". *Engineering Applications of Artificial Intelligence* 46 (noviembre): 269–78. doi:10.1016/j.engappai.2015.09.008.

Lara Velázquez, Pedro, Sergio G. De los Cobos Silva, Miguel Ángel Gutiérrez Andrade, y Eric Alfredo Rincón García. 2015. "Coloración de gráficas suaves". *Revista de Matemática: Teoría y Aplicaciones* 22 (2): 311. doi:10.15517/rmta.v22i2.20838.

Lara Velázquez, Pedro, Sergio G. De los Cobos Silva, Miguel Ángel Gutiérrez Andrade, Eric Alfredo Rincón García, Antonin Ponsih, y Roman Anselmo Mora Gutiérrez. 2015a. "COMPARATIVE PHILOLOGY AMONG IBERIAN LANGUAGES USING SOFT GRAPH COLORING". En *New Techniques for Decision Making under Uncertainty*, 39–48. Girona.

———. 2015b. "PATTERN RECOGNITION USING SOFT GRAPH COLORING". En *New Techniques for Decision Making under Uncertainty*, 1–11. Girona.

Lara Velázquez, Pedro, Lizbeth Gallardo-López, Miguel Ángel Gutiérrez Andrade, y Sergio Gerardo De los Cobos Silva. 2010. "Asignación de frecuencias en telefonía celular aplicando el problema de coloracion robusta". *Revista de Matemática: Teoría y Aplicaciones* 16 (2): 231. doi:10.15517/rmta.v16i2.303.

Lara Velázquez, Pedro, Miguel Ángel Gutiérrez Andrade, Javier Ramírez Rodríguez, y Rafael López Bracho. 2012. "Un Algoritmo Evolutivo para Resolver el Problema de Coloración Robusta". *Revista de Matemática: Teoría y Aplicaciones* 12 (1-2): 111. doi:10.15517/rmta.v12i1-2.255.

Lih, Ko-Wei. 2013. "Equitable Coloring of Graphs". En *Handbook of Combinatorial Optimization*, editado por Panos M. Pardalos, Ding-Zhu Du, y Ronald L. Graham, 1199–1248. New York, NY: Springer New York. http://link.springer.com/10.1007/978-1-4419-7997-1_25.

Lovász, László. 2010. “Discrete and Continuous: Two Sides of the Same?” En *Visions in Mathematics*, editado por N. Alon, J. Bourgain, A. Connes, M. Gromov, y V. Milman, 359–82. Basel: Birkhäuser Basel. http://link.springer.com/10.1007/978-3-0346-0422-2_13.

Martí, Rafael, y Manuel Laguna. 2003. “Scatter Search: Diseño Básico y Estrategias Avanzadas”. *Inteligencia Artificial. Revista Iberoamericana de Inteligencia Artificial* 7 (19): 0.

Martí, Rafael, Manuel Laguna, y Vicente Campos. 2005. “Scatter Search vs. Genetic Algorithms”. En *Metaheuristic Optimization via Memory and Evolution*, editado por Ramesh Sharda, Stefan Voß, César Rego, y Bahram Alidaee, 30:263–82. Boston: Kluwer Academic Publishers. http://link.springer.com/10.1007/0-387-23667-8_12.

Martí, Rafael, Manuel Laguna, y Fred Glover. 2006. “Principles of Scatter Search”. *European Journal of Operational Research* 169 (2): 359–72. doi:10.1016/j.ejor.2004.08.004.

Martí, Rafael, Juan-José Pantrigo, Abraham Duarte, Vicente Campos, y Fred Glover. 2011. “Scatter Search and Path Relinking: A Tutorial on the Linear Arrangement Problem”. *International Journal of Swarm Intelligence Research* 2 (2): 1–21. doi:10.4018/jsir.2011040101.

Mladenović, N., y P. Hansen. 1997. “Variable Neighborhood Search”. *Computers & Operations Research* 24 (11): 1097–1100. doi:10.1016/S0305-0548(97)00031-2.

Musliu, Nysret, y Martin Schwengerer. 2013. “Algorithm Selection for the Graph Coloring Problem”. En *Learning and Intelligent Optimization*, editado por Giuseppe Nicosia y Panos Pardalos, 7997:389–403. Berlin, Heidelberg: Springer Berlin Heidelberg. http://link.springer.com/10.1007/978-3-642-44973-4_42.

Press, William H., ed. 2007. *Numerical recipes: the art of scientific computing*. 3rd ed. Cambridge, UK ; New York: Cambridge University Press.

Rámirez, J., y J. Yáñes. 2003. “The robust coloring problem”. *European Journal of Operational Research* 148: 546–58.

Ramírez Rodríguez, J. 2001. *Extensiones del Problema de Coloración de Grafos. Tesis de Doctorado*. Madrid, España: Universidad Complutense de Madrid.

Ramírez Rodríguez, Javier, M.A Gutiérrez Andrade, R López Bracho, y J Yañes Gestoso. 2003. “Heuristics for the robust coloring problem” 18.

Ramírez Rodríguez, Javier, Miguel Ángel Gutiérrez Andrade, Pedro Lara Velázquez, y Rafael Lopez Bracho. 2011. “Heurísticas para el Problema de Coloracion Robusta”. *Revista de Matemática: Teoría y Aplicaciones* 18 (1): 137. doi:10.15517/rmta.v18i1.2119.

Resende, Mauricio G.C., Celso C. Ribeiro, Fred Glover, y Rafael Martí. 2010. “Scatter Search and Path-Relinking: Fundamentals, Advances, and Applications”. En *Handbook of Metaheuristics*, editado por Michel Gendreau y Jean-Yves Potvin, 146:87–107. Boston, MA: Springer US. http://link.springer.com/10.1007/978-1-4419-1665-5_4.

Riaz, Ferozuddin, y Khidir M. Ali. 2011. “Applications of Graph Theory in Computer Science”. En , 142–45. IEEE. doi:10.1109/CICSyN.2011.40.

Rojas, Beatriz Pérez, y María Auxilio Osorio Lama. 2010. *Análisis Comparativo de Heurísticas para el Problema de Calendarización de Trabajos con Transferencia Cero*.

Sánchez, Johnatan E Pecero. s/f. “Sobre la Calendarización en la Grid”.

Tomar, Ranjeet Singh, Sonali Singh, Shekhar Verma, y Geetam Singh Tomar. 2013. “A Novel ABC Optimization Algorithm for Graph Coloring Problem”. En , 257–61. Mathura: IEEE. doi:10.1109/CICN.2013.61.

Wegener, Ingo. 2005. *Complexity Theory: Exploring the Limits of Efficient Algorithms*. Berlin ; New York: Springer.

Whitley, Darrell. 1994. “A Genetic Algorithm Tutorial”. *Statistics and Computing* 4 (2). doi:10.1007/BF00175354.

Casa abierta al tiempo
UNIVERSIDAD AUTÓNOMA METROPOLITANA

PCyTI

UNIVERSIDAD AUTÓNOMA METROPOLITANA – IZTAPALAPA
DIVISIÓN DE CIENCIAS BÁSICAS E INGENIERIA

BÚSQUEDA DISPERSA PARA EL PROBLEMA DE COLORACIÓN DE GRÁFICAS SUAVES

Tesis que presenta:
Hugo Eduardo Vásquez Calderón
Para obtener el grado de
Maestro en Ciencias y Tecnologías de la Información

Asesores: Dr. Pedro Lara Velázquez
Dr. Sergio G. De los Cobos Silva

Jurado calificador:

Presidente: Dr. Miguel Ángel Gutiérrez Andrade

Secretario Dr. Pedro Lara Velázquez

Vocal: Dra. Hérica Sánchez Larios

México, ciudad de México, noviembre de 2017

Casa abierta al tiempo
UNIVERSIDAD AUTÓNOMA METROPOLITANA

ACTA DE EXAMEN DE GRADO

No. 00067
Matrícula: 2153805515

BÚSQUEDA DISPERSA PARA EL PROBLEMA DE COLORACIÓN DE GRÁFICAS SUAVES

HUGO EDUARDO VASQUEZ CALDERON
ALUMNO

REVISÓ

LIC. JULIO CESAR DE LARA ISASSI
DIRECTOR DE SISTEMAS ESCOLARES

En la Ciudad de México, se presentaron a las 12:00 horas del día 24 del mes de noviembre del año 2017 en la Unidad Iztapalapa de la Universidad Autónoma Metropolitana, los suscritos miembros del jurado:

DR. MIGUEL ANGEL GUTIERREZ ANDRADE
DRA. HÉRICA SÁNCHEZ LARIOS
DR. PEDRO LARA VELAZQUEZ

Bajo la Presidencia del primero y con carácter de Secretario el último, se reunieron para proceder al Examen de Grado cuya denominación aparece al margen, para la obtención del grado de:

MAESTRO EN CIENCIAS (CIENCIAS Y TECNOLOGIAS DE LA INFORMACION)

DE: HUGO EDUARDO VASQUEZ CALDERON

y de acuerdo con el artículo 78 fracción III del Reglamento de Estudios Superiores de la Universidad Autónoma Metropolitana, los miembros del jurado resolvieron:

Aprobar

Acto continuo, el presidente del jurado comunicó al interesado el resultado de la evaluación y, en caso aprobatorio, le fue tomada la protesta.

DIRECTOR DE LA DIVISIÓN DE CBI

DR. JOSE GILBERTO CORDOBA HERRERA

PRESIDENTE

DR. MIGUEL ANGEL GUTIERREZ ANDRADE

VOCAL

DRA. HÉRICA SÁNCHEZ LARIOS

SECRETARIO

DR. PEDRO LARA VELAZQUEZ