UNIVERSIDAD AUTÓNOMA DE SINALOA

FACULTAD DE INFORMÁTICA CULIACÁN
FACULTAD DE CIENCIAS DE LA TIERRA Y EL ESPACIO

MAESTRIA EN CIENCIAS DE LA INFORMACIÓN

ALGORITMOS EVOLUTIVOS PARALELOS PARA RESOLVER EL PROBLEMA DE COLOREADO DE GRAFOS

TESIS

QUE COMO REQUISITO PARCIAL PARA OBTENER EL GRADO DE MAESTRO EN CIENCIAS DE LA INFORMACIÓN

PRESENTA:
JESSICA GUADALUPE URREA GUERRERO

DIRECTORES DE TESIS:
DRA. XIOMARA PENÉLOPE ZALDÍVAR COLADO
DR. ROLANDO MENCHACA MÉNDEZ

Culiacán, Sinaloa México. Febrero de 2019

Dedicatoria y agradecimientos

Gracias a mis padres, a mi hermano y a Eduardo,
a mis compañeros de maestría que
juntos logramos atravesar múltiples adversidades,
pero sobre todo a Mauro.

También quiero agradecer al
Posgrado en Ciencias de la Información,
a la Universidad Autónoma de Sinaloa,
y al CONACyT.

De la Facultad de Informática Culiacán al
M.C. Gerardo Galvez, M.C. Gerardo Beltrán,
Dr. Jorge Navarro y Dr. Arturo Yee.

En especial a mis asesores de tesis que
me brindaron el apoyo necesario para lograr mis objetivos.

Le dedico esta tesis a Rolando, por todo su apoyo
y confianza que me brindó durante este tiempo,
***gracias por creer en mi**.*

La presente tesis está basada en trabajo realizado gracias al financiamiento del Consejo Nacional de Ciencia y Tecnología de México (CONACyT).

Abstract

The decision version of the graph coloring problem is one of the 21 problems that Richard Karp [1] showed belong to the NP-Complete class. The problems that belong to this class are characterized by their complexity and by the fact that an algorithm that solves any of them could be used to solve all the other problems that belong to it [1].

There is another class of problems known as NP-Hard [3] [4] which is composed of even more difficult problems than NP-Complete problems. The extra complexity lies in the fact that unlike NP-Complete problems, NP-Hard problems can not be verified in polynomial time. This means that even if an oracle provides us the answer to any of these problems, we do not know an algorithm that can verify in polynomial time that said answer is correct.

Another difference between NP-Hard problems, and NP-Complete problems, is that the first are usually problems of combinatorial optimization, while the second are decision problems in which the algorithm only has to respond if an input string is or not an instance of the problem.

Unfortunately, an algorithm capable of solving in polynomial time any of the problems belonging to either of these two classes (NP-Complete and NP-Hard) is not known, so it is necessary to use alternatives as metaheuristics [2], that, although they do not provide theoretical guarantees, in practice they have shown that in many cases they are capable of finding the best known solutions. The development of this type of algorithms is of great importance due to the fact that NP-Hard problems appear in almost all areas of computer science such as machine learning, computer networks, operating systems, information systems, others.

In this work, we propose the parallel implementation, based on CUDA, of a number of different versions of an hybrid evolutionary algorithm [5] [6] for the problem of graph coloring. This algorithm is characterized by mixing evolutionary computation techniques with local search heuristics with the aim of finding solutions close to the optimum. Our experimental results show that our parallel implementation is capable of consistently reducing the running time of the evolutionary algorithm.

Key words: *NP-Hard, graph coloring, combinatorial optimization, metaheuristics, parallel algorithms, CUDA.*

Resumen

La clase de problemas *NP-Difícil* [3] [4] está compuesta por problemas de búsqueda que no se pueden verificar en tiempo polinomial, es decir, que aún y cuando un oráculo nos proporcione la respuesta a alguno de estos problemas, no se conoce un algoritmo que pueda verificar en tiempo polinomial que dicha respuesta sea correcta.

A la fecha no se conoce un algoritmo capaz de resolver en tiempo polinomial alguno de los problemas pertenecientes a la clase NP-Difícil, por lo que es necesario utilizar alternativas como metaheurísticas [2], que aunque no proveen garantías teóricas, en la práctica han demostrado que en muchos casos son capaces de encontrar las mejores soluciones conocidas. El desarrollo de este tipo de algoritmos es de suma importancia por el hecho de que los problemas NP-Difícil aparecen en casi todas las áreas de las ciencias de la computación como aprendizaje de máquina, redes de computadoras, sistemas operativos, sistemas de información, entre otras.

En este trabajo, proponemos una implementación paralela, basada en CUDA, con un conjunto de versiones diferentes de un algoritmo evolutivo híbrido [5] [6] para el problema de coloreado de grafos. Dicho algoritmo se caracteriza por mezclar técnicas de cómputo evolutivo con heurísticas de búsqueda local con el objetivo de encontrar soluciones cercanas a la óptima. Los resultados experimentales muestran que nuestra propuesta de implementación paralela es capaz de consistentemente reducir el tiempo de ejecución de los algoritmos evolutivos.

Palabras clave: *problemas NP-Difícil, colorado de grafos, optimización combinatoria, metaheurísticas, algoritmos paralelos, CUDA.*

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

La versión de decisión del problema de coloreado de grafos es uno de los 21 problemas que Richard Karp [1] demostró pertenecen a la clase NP-Completa. Los problemas que pertenecen a esta clase se caracterizan por su complejidad y por el hecho de que un algoritmo que resuelva cualquiera de ellos podría ser usado para resolver a todos los demás problemas que pertenecen a dicha clase [1].

Existe otra clase de problemas conocida como NP-Difícil [3] [4] que está compuesta por problemas aún más difíciles que los problemas NP-Completos. La complejidad extra radica en el hecho de que a diferencia de los problemas NP-Completos, los problemas NP-Difícil no se pueden verificar en tiempo polinomial. Esto quiere decir, que aún y cuando un oráculo nos proporcione la respuesta a alguno de estos problemas, no se conoce un algoritmo que pueda verificar en tiempo polinomial que dicha respuesta sea correcta.

Otra diferencia importante entre los problemas NP-Difícil, y los problemas NP-Completos, es que los primeros suelen ser problemas de optimización combinatoria, mientras que los segundos son problemas de decisión en los que el algoritmo sólo tiene que responder si una cadena de entrada es o no una instancia del problema en cuestión.

Este trabajo propone un sistema paralelo para que produce buenas soluciones a instancias de la versión de búsqueda del problema de coloreado de grafos, el cual pertenece a la clase NP-Difícil. Dada la importancia del problema, a la fecha existen múltiples propuestas para su solución. Una particularmente prometedora consiste en utilizar *algoritmos híbridos* [9] [12] [14] que combinan algoritmos evolutivos con esquemas de búsqueda local. En la literatura se muestra que este tipo de herramientas han dado buenos resultados [18], sin embargo, este tipo de algoritmos suelen ser extremadamente costosos en términos de su tiempo de ejecución, por lo que se propone desarrollar una versión paralela que utilice la tecnología de tarjetas CUDA con las cuales se espera una mejora significativa.

En la siguiente sección se presenta el planteamiento del problema que se pretende resolver con este trabajo de tesis, posteriormente se presentan las hipótesis de investigación, así como el objetivo general y particulares. Finalmente se presenta la justificación de porque es pertinente la realización del presente trabajo.

1.1. Planteamiento del problema

El problema de coloreado de grafos es posible verla como una asignación de clases a los elementos de un conjunto de objetos que pueden estar relacionados entre sí. El objetivo es asignar los objetos al menor número de clases, con la restricción que dos objetos que pertenezcan a la misma clase no pueden estar relacionados.

En el contexto de la teoría de grafos, los objetos del problema de coloreado de grafos, se representa por medio de un conjunto de vértices V, y las relaciones son modeladas por un conjunto E de aristas entre vértices. De esta forma, dos nodos $u, v \in V$ tal que $(u, v) \in E$ no pueden pertenecer a la misma clase. Una forma de distinguir las clases es utilizando un conjunto de colores, y a la división en clases se le conoce como coloreado [40].

Formalmente, la versión de búsqueda del problema de coloreado de grafos se define por un grafo no dirigido $G = (V, E)$, un número de colores disponibles k y consiste en encontrar una función $f : V \to \{1, 2, 3, .., k\}$ tal que el tamaño del conjunto de *aristas monocromáticas* $|M|$ es minimizado, donde $M = \{(u, v) : (u, v) \in E \text{ y } f(u) = f(v)\}$, es decir que el conjunto M contiene todos los pares de nodos adyacentes cuya función de asignación de color sea la misma.

Existen dos variantes de este problema, la primera es que el valor k (número de colores) es fijo y se desea minimizar el número de aristas monocromáticas. En la segunda versión, k es variable y se requiere encontrar el valor mínimo de k, tal que el número de aristas monocromáticas es igual a cero. Se dice que un coloreado es *propio* cuando el número de aristas monocromáticas es igual a cero.

El número cromático de un grafo G es el entero más pequeño k tal que el grafo tiene un coloreado propio; y se denota por $\chi(G)$. Se dice que un grafo G con $\chi(G) = k$ es *k-cromático*, y cuando $\chi(G) \leq k$, decimos que es *k-coloreable* [7] [40].

Debido a que el problema de coloreado de grafos pertenece a la clase NP-Difícil, a menos que la clase de problemas P sea igual a la NP, no es posible diseñar un algoritmo que se ejecute en tiempo polinomial capaz de resolverlo. Por lo anterior, se han utilizado técnicas metaheurísticas [8] [11] no polinomales para intentar encontrar buenas soluciones. Desafortunadamente, el tiempo de ejecución de la mayoría de las metaheurísticas usadas en la actualidad es demasiado elevado, restringiendo su uso a instancias pequeñas y por lo tanto, haciéndolas poco prácticas. Por lo anterior, es indispensable desarrollar implementaciones eficientes que sean capaz de resolver instancias del problema en tiempo razonable.

En el caso particular de los algoritmos híbridos, el principal cuello de botella radica en que cada individuo realiza una etapa de búsqueda local de manera secuencial. Para resolver este problema, se propone paralelizar la búsqueda local para que todos los individuos puedan progresar al mismo tiempo.

1.2. Hipótesis

1.2.1. Hipótesis 1

Es posible implementar algoritmos evolutivos híbridos paralelos que superen el desempeño de algoritmos voraces para resolver problemas combinatorios como el de coloreado de grafos.

1.2.2. Hipótesis 2

La etapa de búsqueda local de los algoritmos evolutivos híbridos puede ser paralelizada para reducir su tiempo de ejecución.

1.3. Objetivos

1.3.1. Objetivo general

Desarrollar y caracterizar un algoritmo evolutivo híbrido paralelo para resolver el problema de coloreado de grafos.

1.3.2. Objetivos específicos

- Realizar un análisis detallado de los algoritmos evolutivos propuestos en la literatura para resolver el problema de coloreado de grafos.
- Diseñar un algoritmo evolutivo híbrido paralelo para resolver el problema de coloreado de grafos.
- Implementar el algoritmo propuesto sobre la plataforma *CUDA* (*Compute Unified Device Architecture*).
- Caracterizar experimentalmente el desempeño del algoritmo propuesto utilizando instancias de prueba estandarizadas.
- Caracterizar experimentalmente el impacto de incrementar el número de núcleos cuda en el tiempo de ejecución del algoritmo propuesto.
- Caracterizar experimentalmente el impacto de utilizar diferentes modelos de memoria en el tiempo de ejecución del algoritmo propuesto.
- Analizar los resultados obtenidos para proponer investigaciones futuras.

1.4. Justificación

La relevancia del problema de coloreado de grafos se debe, por un lado, a sus múltiples aplicaciones prácticas en áreas como planificación de tareas [20], planificación de grupos de trabajo [21], diseño de compiladores [22], reconocimiento de patrones [23], redes de computadoras [24] [25], entre otros.

Por otro lado, al pertenecer a la clase NP-Difícil, el problema de coloreado de grafos tiene una gran importancia teórica ya que de encontrarse un algoritmo que lo resuelva en tiempo polinomial, se demostraría que la clase de problemas P es igual a la clase NP. Lo anterior le daría respuesta a la pregunta que es considerada como una de las más importantes en teoría de la computación.

Finalmente, con el desarrollo de modelos de programación paralela basado en tarjetas gráficas es posible extender un poco la barrera de problemas que pueden ser resueltos con grandes cantidades de información, y bajar su costo temporal. La contribución que se espera es poder resolver problemas que podemos observar en la vida cotidiana, además de brindar la posibilidad de una mejor calidad de vida ya que pueden ser aplicadas en medicina [26], química [27], aprendizaje de máquina [28], entre otras.

Capítulo 2

Marco teórico

2.1. Introducción

En este capítulo se presentan los principales conceptos relacionados con este tema de tesis, ya que con ellos será posible entender toda la teoría que existe detrás del problema de coloreado de grafos, y desarrollar un análisis de los conceptos así como de las técnicas utilizadas según la literatura para tratar de resolverlo. En primera instancia se introduce una pequeña introducción de los temas relacionados como la teoría de clases de complejidad computacional, en particular la clase NP-Difícil, historia del problema de coloreado de grafos a la cual pertenece la versión de búsqueda de la clase NP-Difícil, algoritmos genéticos híbridos, cómputo paralelo, y la plataforma CUDA. Posteriormente se presenta un análisis detallado de cada uno de estos temas, además de incluir diversos temas relacionados como teoría de grafos, optimización combinatoria, búsquedas estocásticas. Finalmente, concluyendo con un panorama general de sobre algoritmos paralelos, analizando a detalle la plataforma $CUDA$, así como su arquitectura, modelo y manejo de memoria (jerarquías de memoria).

2.2. Problemas P, NP-Completos y NP-Difíciles

En la vida diaria podemos observar que es más difícil verificar si la solución a un problema es correcta que resolverlo. Lo cual nos lleva a preguntarnos si esto es ¿coincidencia o representa un hecho fundamental de la vida (o propiedad del mundo)? Esto representa la esencia de la pregunta P contra NP. Donde P representa los problemas de búsqueda que pueden ser resueltos de manera eficiente (en tiempo polinomial) y NP representa los problemas de búsqueda para los cuales sus soluciones pueden ser verificadas de manera eficiente. La clasificación concreta de estos problemas son: P (polinomial time), NP (*nondeterministic polynomial time*), NP-Completo y NP-Difícil, todas estas se basan en el modelo de máquinas de Turing y por lo tanto pueden ser tratadas como un lenguaje. La primera se refiere a todos aquellos problemas de decisión para los cuales existe un algoritmo en tiempo polinomial que lo resuelve; la segunda se define como todos aquellos problemas de decisión para los cuales existe un certificador de tiempo polinomial; en el tercero dice que los problemas NP-Completos pueden ser utilizados para resolver todos los demás problemas NP; y por último los problemas NP-Difícil, son aquellos problemas para los cuales no existe un algoritmo certificador en tiempo polinomial, las cuales

se asocian como las *versiones de búsqueda*, la mayoría de los problemas que se ven en la práctica son NP-Difícil [4]. Es frecuente asociar la eficiencia computacional con algoritmos de tiempo polinomial. Dicha asociación es justificada por el hecho que los polinomios son una clase de funciones de crecimiento moderado que está cerca de operaciones que corresponden a la naturaleza con la que los algoritmos se componen. Además, la case de algoritmos en tiempo polinomial es independiente de un modelo específico de cómputo.

Por lo que podemos derivar que la importancia de clasificar los problemas radica en encontrar similitudes entre ellos, para poder desarrollar algoritmos eficientes y verificar la noción de intratabilidad de estos problemas, es decir, que sería posible identificar todos aquellos problemas que no sabemos resolver y/o que no existe algoritmo en tiempo polinomial capaz de resolverlo.

2.3. Coloreado de grafos

A mediados de siglo XIX Francis Guthrie notó que se podía colorear el mapa de Inglaterra con sólo cuatro colores, esto con el objetivo de dar a las regiones vecinas colores diferentes, dando la oportunidad de observar los bordes comunes y tratar de minimizar la distracción visual, utilizando el menor número de colores. Francis se preguntó si esto era posible para cualquier mapa, llegando al problema de coloreado de grafos [4].

El problema de coloreado puede representarse por medio de un grafo no dirigido, en el cual los nodos juegan el papel de las regiones a colorear, mientras que las aristas representan los pares de nodos que son vecinos. Formalmente la literatura lo define de la siguiente manera: sea G=(V,E) un grafo no dirigido con un conjunto de nodos V y un conjunto de aristas E, tal que los nodos adyacentes sea el menor número de aristas monocromáticas posible. Dado la definición anterior podemos concluir que un subconjunto del grafo G puede ser llamado "*independent set*", si no cuenta con dos nodos adyacentes (unidos por una arista) que pertenezcan a él, dicho de otra manera, a partir del grafo ya coloreado existe un independent set. Una definición importante utilizada en el contexto del problema de coloreado de grafos es que si partimos en k conjuntos independientes el grafo G es llamado "k-coloring", por último, podemos concluir que el problema de coloreado de grafos es encontrar el coloreado óptimo dado un grafo G [9].

2.4. Algoritmos genéticos

Los algoritmos genéticos (AG) fueron desarrollados por John Holland, y estos se basan en mecanismos de selección natural, combinando la sobrevivencia del individuo (dependiendo de su aptitud) a través de estructuras de cadenas, estas se intercambian por estructuras aleatorias formando los algoritmos de búsqueda. En cada generación, una nueva criatura artificial (cadena) es creada usando bits y pedazos de aptitudes del anterior. El espacio de búsqueda de los AG se representa en una gráfica continua, es decir es un esquema numérico lineal, el cual se puede ver de muchas formas y tamaños. La idea es que, dentro de un espacio de búsqueda finito o mediante una discretización finita del espacio de búsqueda, los algoritmos comiencen a buscar los valores de la función objetivo en cada punto del

espacio, una a la vez. Los problemas con los que nos enfrentamos en la vida cotidiana son espacios de búsqueda discontinuo y multimodal, mientras que estos métodos dependen de requerimientos de continuidad restrictiva, abarcando un dominio limitado de problemas. [17].

2.4.1. Algoritmos genéticos híbridos

Los algoritmos genéticos híbridos están inspirados por modelos que combinan la adaptación evolutiva de una población con el aprendizaje individual durante su ciclo de vida, mediante el refinamiento local y así mejorar a cada individuo. La combinación de los algoritmos genéticos con búsqueda local también son llamados "algoritmos meméticos". Se comenzaron a aplicar debido a que los algoritmos genéticos no son buenos para problemas de optimización combinatoria, por lo que se buscaron alternativas para dotarlos de otros mecanismos, para ser competitivos en el área y mejorar la eficiencia de la búsqueda [18]. Esto debido a que los algoritmos genéticos son utilizados en un espacio continuo, mientras que los problemas de optimización se encuentran dentro de un espacio discreto.

2.5. Cómputo paralelo

Cuando se menciona el paralelismo y el verdadero uso de cómputo paralelo se habla de problemas complejos o extensos. El ámbito de la ciencia en computación se ha enfrentado a estos problemas lo largo de los años, ya que existen restricciones en las soluciones potenciales, debido al costo computacional o bien al costo espacial, esto brindó la oportunidad de reinventar una nueva forma de uso para otro tipo de hardware capaz de romper estas barreras.

El principal objetivo del cómputo paralelo es mejorar la velocidad de cómputo, encontrando nuevas aplicaciones y ventajas de los multiprocesadores y el hardware heterogéneo, por medio del manejo eficiente de los recursos, a su vez debe ser portable y fácil al momento de escribir un programa, como si fuera escrito en cómputo secuencial [38].

El cómputo paralelo puede definirse como el uso simultaneo de múltiples recursos de cómputo (núcleos o computadoras) para realizar el cálculo concurrente. Por lo cual un problema grande puede ser partido en pedazos pequeños, y cada pedazo es resuelto en diferentes "núcleos", con la restricción de que los pedazos del problema no sean dependientes unos de otros. Hoy en día los diseños de chips es integrar múltiples cores en un solo procesador, usualmente llamado "multicore" y los cuales soportan paralelismo a nivel arquitectura. La programación en paralelo se puede ver como el proceso de mapear la computación de un problema a cores disponibles de los cuales la ejecución paralela es obtenida [30].

2.5.1. Plataforma CUDA

CUDA (Compute Unified Device Architecture) es una plataforma de cálculo paralelo creada por Nvidia en el 2007, con múltiples núcleos (GPU's), para resolver problemas de una manera eficiente. Está caracterizada por paralelizar un rango muy amplio de aplicaciones, algunas de ellas son: matrices dispersas, ordenamiento, búsqueda y modelos físicos. Con la transparencia de cientos núcleos de procesadores de hilos concurrentes.

La plataforma CUDA provee tres puntos clave, una jerarquía de grupo de hilos, memoria compartida, y barreras de sincronización, las cuales garantizan una clara estructura paralela, con diferentes lenguajes de códigos convencionales, entre ellos: C, como uno de los más utilizados. Así como su abstracción que guía al programador a particionar el problema en sub-problemas, que pueden ser resueltos de manera paralela [31]. CUDA intenta aprovechar la potencia del GPU (Unidad de Procesamiento gráfico) para incrementar el rendimiento del sistema, pasando de ser un sistema de procesamiento central a realizar "coprocesamiento" repartido entre CPU (Central Processing Unit) y la GPU [32]. Una de las mayores ventajas de CUDA es que su base es el lenguaje C, por lo que tiene una gran compatibilidad, además está disponible en laptos, PCs, servidores, entre otros.

2.6. Teoría de grafos

2.6.1. Grafos

Para entender mejor el concepto de *grafo*, es necesario primero analizar el concepto de un conjunto. Trudeau [39] lo define como una colección de distintos objetos en el que ninguno de ellos es un conjunto por sí solo. Está implícito que la interpretación de "colección" está compuesta de muchas cosas, solo una cosa, o incluso nada, mientras que el concepto de "objeto" desde un punto de vista matemático es cualquier cosa concebible, incluyendo números. Lo único que queda fuera de lo concebible cuando se habla de figuras, es que contenga propiedades geométricas de un triángulo y además las propiedades de un círculo, todo lo demás es aceptable. Los objetos son llamados elementos del conjunto. Con los conceptos anteriores podemos definir un grafo, el cual es un objeto que consiste en dos conjuntos llamado *conjuntos de vértices* y *conjunto de aristas*. El conjunto de vértices debe cumplir con dos propiedades: no estar vacío y ser finito, mientras que el conjunto de aristas puede estar vacío, de lo contrario sus elementos es un subconjunto de dos elementos del conjunto de vértices. Ejemplo: {P,Q,R,S,T,U} pertenece al conjunto de vértices y {{P,Q},{P,R},{Q,R},{S,U}} al conjunto de aristas y ambos constituyen un grafo, y puede ser llamado "G" de manera más corta. Los elementos del conjunto de vértices son llamados vértices o nodos, y el conjunto de aristas es llamado aristas.

Diestel [7] menciona que el conjunto de vértices de un grafo se denota por V(G), y el conjunto de arista de un grafo como E(G). El número de vértices de un grafo G es su *orden*, escrito como |G|; mientras que el número de aristas se denota por ‖G‖. Los grafos son *finitos, infinitos, y contables* de acuerdo a su orden. Un grafo de orden 0 o 1 es llamado *trivial*. El número de vértices se denota por "V" y el número de aristas por "E". Si {X,Y} es una arista del grafo, decimos que {X,Y} es una unión o conexión de vértices X y Y, y decimos que ambos son adyacentes uno del otro. La arista {X,Y} es incidente a cada X y Y, y cada X y Y es incidente a {X,Y}. Dos aristas incidentes al mismo vértice son llamadas *aristas adyacentes*. Un vértice incidente que no contiene aristas está aislado. Diestel [7] también menciona que un par no adyacente de vértices o aristas es llamado *independiente* es decir, el conjunto de vértices o de aristas es *independiente* si dos de sus elementos no son adyacentes. El conjunto de vértices independientes también son llamados *estables*. En Jensen [40] cita el "teorema 1" de bruijn y Erdös, el cual menciona que si todos los subgrafos finitos de un grafo infinito G es k-coloreable, donde k es finita, entonces G es k-coloreable. Este teorema fue probado por L. Pósa.

2.6.2. Coloreado de grafos

Trudeau [39] define un grafo coloreado como una asignación de colores a cada vértice de tal forma que los vértices adyacentes sean de colores diferentes. Mientras que Diestel [7] lo define como un grafo $G = (V, E)$ donde $c : V \longrightarrow S$ tal que $c(v) \neq c(w)$ cuando v y w son adyacentes. En otras palabras, un grafo ha sido coloreado si para cada arista se tiene dos coloreados distintos en sus puntas (ver Figura 2.1). Todos los elementos del conjunto S es llamado *colores* disponibles. Lo que se busca es encontrar el entero más pequeño k tal que G sea un *k-coloreado*. Como podemos observar un k-coloreado es una

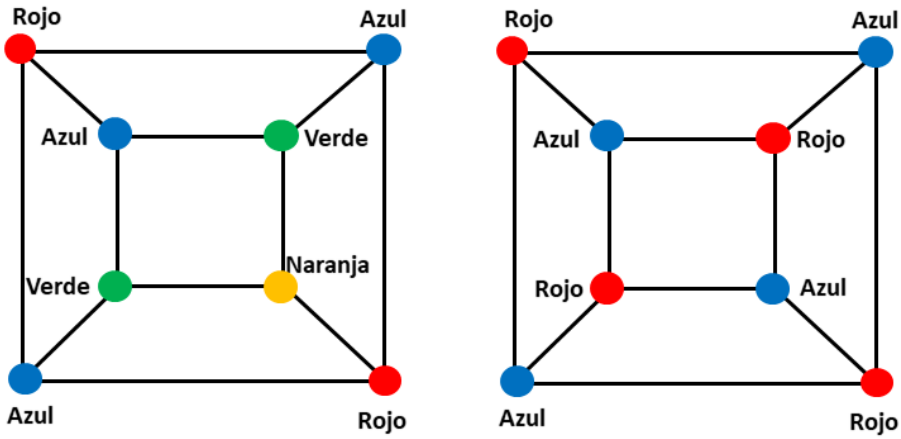

Figura 2.1: Coloreado de grafos

partición de vértices en k conjuntos independientes, también llamadas *clases de colores* [7]. Usualmente los colores se representan por números. Un grafo sin etiquetas que ha sido coloreado se parece a un grafo etiquetado. Una vez que el grafo ha sido coloreado; se tiene que verificar si todos los vértices con el mismo color nunca sean adyacentes.

El número cromático de un grafo es el número más pequeño de colores con los cuales puede ser coloreado y se denota por "χ". Un grafo G con $\chi = k$ es llamado *k-cromático*, mientras que cuando $\chi(G) \leq k$ se le llama *k-coloreable*. Un grafo siempre puede ser coloreado dando colores diferentes a cada vértice, por lo tanto, cualquier grafo $\chi \leq |V|$, tomando en cuenta que χ es siempre mayor a 1, entonces el número cromático de un grafo debe satisfacer la desigualdad $1 \leq \chi \leq |V|$.

El grafo completo son solo aquellos para los cuales χ es igual a $|V|$, a diferencia de los grafos normales, este tiene la característica que todos los vértices son adyacentes, por lo cual todos los vértices deben contener colores distintos; mientras que los grafos vacíos o nulos son los únicos grafos con $\chi = 1$, el cual tiene la posibilidad de contener el mismo color en cada punta, y como se puede observar los grafos completos y los grafos vacíos son complementarios. Con esta excepción, el número cromático de un grafo debe satisfacer la siguiente desigualdad $2 \leq \chi \leq |V| - 1$. Para un grafo arbitrario no hay una regla simple que determine donde caerá el número cromático dentro del rango $2 \leq \chi \leq |V| - 1$. En la mayoría de los casos lo que hay que hacer es colorear el grafo con el menor número de colores posible y estudiar el resultado hasta que haya certeza que no hay otro coloreado posible con menor número

de colores. Para un grafo complicado, puede ser necesario probar que no es posible ser coloreado con menos colores [39].

Lo que respecta a las aplicaciones posibles para el coloreado de grafos, a continuación se mencionan algunas de las más utilizadas, como son: *construir tablas de horarios, asignación de frecuencias, colocación de registros o pruebas de circuitos impresos.* Y Algunos de los métodos utilizados en la literatura para dar solución a este tipo de problemas han sido: *enfoques de construcciones aproximadas "greedy", estrategias híbridas, metaheurísticas de búsqueda local (temple simulado,tabú), redes neuronales* [47], entre otras.

2.6.3. Grados de vértices y coloreado

Diestel [7] define el grado de un grafo como el conjunto de vecinos de un vértice v en G y se denota como $N(v)$, un vértice de grado cero se le llama *aislado.* Por otro lado Brooks en 1941 observó que para cada grafo G puede ser coloreado por Δ(G)+1 colores, donde Δ(G) es el grado máximo de G, y caracterizar el grafo por Δ(G) colores no son suficientes. Así mismo propuso un teorema importante, es el teorema 11 el cual menciona que para todo grafo G se cumple que χ(G)$\leq$ Δ(G)+1. Más aún, se cumple que χ(G)=Δ(G)+1 si y sólo si $\Delta(G) \neq 2$ y G tiene un grafo completo de tamaño Δ(G)+1, o Δ(G)=2 y el grafo G tiene un ciclo de longitud impar. El número del coloreado col(G) es definido por el número más pequeño d para algún orden lineal menor que el conjunto de vértices, en otras palabras, si los vértices de G son $x_1, x_2, ..., x_n$, entonces: $col(G) = 1 + min_p max_i\{d(x_{p(i)}, G_{p(i)})\}$, donde el mínimo se toma sobre todas las permutaciones p de {1,2,...,n} y $G_{p(i)}$ es el subgrafo de G introducido por $x_{p(1)}, x_{p(2)}, ..., x_{p(i)}$, y donde d(x, H) denota el grado de un vértice x en un grafo H. Es claro que col(G)$\leq$ Δ(G)+1, el cual intensifica la desigualdad χ(G)$\leq$ Δ(G)+1. El orden más pequeño $x_{p(1)}, x_{p(2)}, ..., x_{p(i)}$ de los vértices de G son obtenidos por $x_{p(i)}$ sea un vértice del grado más pequeño $\delta(G_i)$ en el subgrafo $G_i = G - \{x_{p(i+1)}, x_{p(i+2)}, ..., x_{p(n)}\}$, para i=n, n-1,...,1, por lo cual: $col(G) \leq 1 + \max\limits_{1\leq i\leq n} \delta(G_i) \leq 1 + \max\limits_{G'\subseteq G} \delta(G')$ El número del coloreado fue definido por Erdös y Hajnal en 1966. Para grafos finitos de tamaño n el número de coloreado es considerado un parámetro difícil debido a que el mínimo es tomado de n! posibles permutaciones. Sin embargo, existe un algoritmo sencillo de tiempo polinomial para computar col(G) para cualquier grafo G. Esto fue descubierto por Matula en 1968, el col(G) es igual a la cota superior para el número cromático dado por Szekeres y Wilf $SW(G) = \max\limits_{H\subseteq G} \min\limits_{x\in V(H)} (d(x, H) + 1)$. El número cromático total $\chi''(G)$ está relacionado con el número cromático de la lista de aristas $\chi'\ell(G)$. Cada arista xy de G es asignada a una lista $\bigwedge(xy)$ de λ colores; entonces un $\bigwedge-$coloreado en la arista, donde se le asigna un color de la lista. El número cromático de la lista de aristas $\chi'\ell(G)$ es el número mínimo λ, en el cual tal coloreado existe dado $\bigwedge$ con λ colores en cada lista por lo tanto, $\chi'\ell(G) \geq \chi'(G)$. El número de la lista cromática de un grafo es a lo más tan grande como el grado máximo, excepto si el grafo es completo o un ciclo impar [7] [40].

2.6.4. Polinomios cromáticos

Decimos que $P(G, k) = k^n + a_1k^{n-1} + a_2k^{n-2} + ... + a_{n-1}k + a_n$ sea el polinomio cromático de G. P(G,k) son los diferentes números de k-coloreados de G con 1,2,...,k colores. Por la definición de polino-

mios cromáticos, el polinomio cromático de la unión de dos grafos disjuntos G y H es $P(G,k)P(H,k)$. Donde $P(G,k)$ y $P(H,k)$ son, respectivamente, los polinomios cromáticos de G y H. En 1968 Read se preguntó si era posible encontrar un conjunto de condiciones algebraicas suficientes y necesarias para que un polinomio sea polinomio cromático para algún grafo, y demostró que el polinomio cromático P(G,k) de un grafo G en n vértices cumple con las siguientes condiciones necesarias: i) El grado de P(G,k) es n, ii) el coeficiente de k^n es 1, iii) el coeficiente de k^{n-1} es $-|E(G)|$, iv) el término constante es 0, v) los términos se alternan en signo, y vi) P(G,λ)$\leq \lambda(\lambda-1)^{n-1}$ para cualquier entero positivo λ, si G está conectado. Sin embargo, Read notó que todas estas condiciones no eran suficientes para caracterizar polinomios cromáticos entre todos los polinomios. En 1988 Read y Tutte en 1988 confirmaron que para cualquier polinomio cromático dado, el número de vertices y aristas de un grafo debe cumplir con i) e iii). Lo siguiente sería generar todos los grafos y calcular su polinomio cromático, el cual es un proceso finito, pero muy impráctico [39].

2.7. Optimización combinatoria

2.7.1. Problemas de optimización combinatoria

Los problemas de optimización combinatorios, se representan por medio de espacios de soluciones discretas, entre los más conocidos están: camino más corto o planificación de diferentes trabajos de un conjunto de máquinas. El objetivo de estos problemas es minimizar o maximizar una función objetivo dado un conjunto de restricciones. Una instancia del problema es dado por el problema junto con una configuración específica de los parámetros. Formalmente un problema de optimización puede ser definido como una tripleta (S, f, Ω), donde S es el espacio, f es la función objetivo que se quiere maximizar o minimizar, y Ω es el conjunto de restricciones que se quieren cumplir para obtener una solución factible. El objetivo es encontrar la solución óptima global, la cual en el caso de un problema de maximización, es aquella solución s* con el valor más alto que satisfaga las restricciones.

El espacio de búsqueda de la mayoría, es tiempo exponencial de acuerdo a la dimensión del problema. La entrada de los problemas de optimización combinatoria a menudo es un grafo o un conjunto de enteros. Esta entrada tiene que ser representada como una secuencia de símbolos de un alfabeto finito. cuando estos problemas se consideran en grafos, pueden ser representados mediante matrices de adyacencia, esta matriz tiene n^2 entradas, el número de entradas es cuadrático con respecto al número de vértices, cuando la matriz de adyacencia representa un grafo no dirigido se dice que es *simétrica*. Un grafo no dirigido puede tener $\binom{n}{2}$=$\Theta(n^2)$ aristas [45].

Un problema importante con optimización combinatoria es el clasificar problemas difíciles. Para poder distinguir los problemas que son sencillos y aquellos que no lo son, se considera la clase de problemas que pueden ser resueltos por una maquina de Turing determinista en tiempo polinomial y los problemas que son resueltos por una maquina de Turing no determinista en tiempo polinomial (clases P y NP). Los problemas en P puede ser resueltos en tiempo polinomial con el algoritmo apropiado, Algunos ejemplos de problemas que pertenecen a esta clase son: problema de camino más corto con una sola fuente y el problema de árbol de expansión mínima. Mientras que los problemas difíciles son llamados NP, también llamados problemas de decisión, cuya salida es si o no. Un problema es NP si y

sólo si dada una solución cualquiera del problema, puede ser verificado en tiempo polinomial [45]. El enfoque clásico ideal con los problemas NP-Difíciles es buscar buenos *algoritmos de aproximación* [46]. Estos son algoritmos que se ejecutan en tiempo polinomial garantizando que el resultado se encuentra muy cercano al óptimo. Estos algoritmos pueden dar buenos resultados para algunos problemas NP-Difíciles como: el problema de la mochila.

Otro de los algoritmos conocidos en la literatura que ha sido capaz de dar buenos resultados para este tipo de problemas son las heurísticas "*greedy*", sin embargo para algunos problemas más difíciles como los problemas de planificación utilizan métodos de *programación lineal* [45].

2.7.2. Algoritmos de búsqueda estocásticos

Estos algoritmos utilizan el cómputo bio-inspirado, los cuales son métodos que tratan los algoritmos en base a decisiones aleatorias y están inspirados en el proceso natural de sobrevivencia del más apto. Un gran número de estos algoritmos son independientes del problema para resolverlo, se basan es espacios de búsqueda que pueden ser representados por módulos ajustables al problema o bien combinados al algoritmo dependiente del mismo. Este tipo de algoritmos también se conocen por ser de propósitos generales, a diferencia de los enfoques de los algoritmos clásicos. El único componente dependiente del problema es su *función de aptitud* que guía la búsqueda y que tiene que ser ajustada al problema.

La definición formal de Neumann [45] menciona que dado un espacio de búsqueda S, el objetivo es optimizar una función dada $f : S \longrightarrow R$, donde R es el conjunto de todos los posibles valores de la función objetivo. El algoritmo de búsqueda estocástica trabaja en un esquema que brinda el espacio de soluciones, regido bajo ciertas consideraciones elegidas en el primer punto de la función objetivo, estas consideraciones se basan en una distribución de probabilidad en el espacio de búsqueda S y que puede ser determinada bajo una heurística. Es decir, que el punto de búsqueda se elige de acuerdo a una distribución de probabilidad, y estas pueden ser dependientes a los puntos de búsqueda anteriores y sus valores de la función. Este proceso se realiza hasta que el criterio se satisfaga. Algunos de los algoritmos conocidos que caen dentro de esta clasificación son: los algoritmos evolutivos, optimización con colonia de hormigas, búsqueda local aleatoria, algoritmo metrópolis y recocido simulado.

2.7.2.1. Algoritmos genéticos

El algoritmo evolutivo más conocido mundialmente, es el algoritmo genético (AG) [41]. Fue propuesto por Holland en 1960s como una forma de estudiar el comportamiento adaptativo. Los AG son considerados como metodologías de funciones de optimización.

Goldberg en su libro de seminario junto con la tesis de Jong ayudó a definir lo que ahora se conoce como los algoritmos genéticos clásicos, los cuales también son llamados *"Canónicos"* o *"AG simples" (AGS)*. Golberg [17] define a los algoritmos genéticos como un mecanismo de selección natural y genética, mientras que los algoritmos aleatorios, no es más que una caminata aleatoria, en el cual toma ventaja de los datos históricos explorando nuevos puntos, esperando una mejora en el rendimiento. A su vez menciona que los métodos de búsqueda, pueden ser divididos en tres categorías: basados en *cálculo, enumerativos, y aleatorios*, el primero está relacionado con las ecuaciones no lineales donde la función objetivo es igual a cero; los esquemas enumerativos se basan en algoritmos de búsqueda

que empiezan buscando en los valores de la función objetivo en cada punto del espacio, y por último, los esquemas aleatorios son conocidos por ser caminos cortos de los métodos basados en cálculo y enumerativos, es decir, son pasos aleatorios en la búsqueda, y utiliza elecciones aleatorias para dirigir el proceso de búsqueda.

Según A.E. Eiben [41] los algoritmos genéticos tienen una representación binaria (que consta de dos partes: *genotipo*, el cual es la representación codificada de las variables, y *fenotipo*, que es el conjunto de variables), una selección proporcional (fitness), una baja probabilidad de mutación y un énfasis en la inspiración de recombinación genética, la cual genera un nuevo candidato como solución (véase Algoritmo 1).

Debido a que tiene su propuesta fue realizada desde los años 60's, esta ha sido muy utilizada durante muchos años brindando muy buenos resultados para múltiples problemas, pero al algoritmo genético simple le hace falta una integración de múltiples características para ser competente en los problemas discretos.

Los AG tradicionales trabajan dada una población de μ individuos, la selección de los padres llena una población intermedia de μ, permitiendo duplicados. La población intermedia genera permutaciones aleatorias para crear parejas y cruzarlas, esto se aplica para aquellos con probabilidad "pc" y los hijos reemplazan a los padres de manera inmediata. La nueva población intermedia sufre una mutación de individuo por individuo, donde cada l bits en un individuo es generada una mutación de acuerdo a una probabilidad "pm". La población intermedia resultante forma la siguiente generación, reemplazando completamente la anterior. Note que las nuevas generaciones pueden ser pedazos o individuos completos de la anterior, puede haber generaciones en las cuales sobrevivan a la cruza y mutación sin modificaciones, pero esto depende de los parámetros μ, "pc", y "pm".

La literatura menciona que en base a pruebas realizadas, se recomienda que los rangos de mutación estén entre $1/l$ y $1/\mu$, mientras que las probabilidades de cruza se recomiendan alrededor de 0.6 - 0.8, y el tamaño de población entre los 50 y bajo los 100, sin embargo la extensión de estos valores se ven reflejados en el poder de cómputo disponible en esa época. Por otro lado, las recomendaciones en Coley [42] sobre la probabilidad de cruza "pc", y el número aleatorio "rc" es generado en un rango entre 0 - 1, el par de individuos seleccionados para la cruza se cumple sí y sólo si $rc \leq pc$, de caso contrario la pareja continua sin cruzarse. Los valores típicos de "pc" eran 0.4 - 0.9, sin la cruza. La función de evaluación promedio de la población puede ir escalando hasta alcanzar la función de evaluación máxima, después de este punto sólo podrá mejorarse a través de la mutación. En la cruza es frecuente que se utilice la cruza de un solo punto, en el cual se "corta" o asigna un punto de manera aleatoria, para tomar la cadena hasta el punto seleccionado e intercambiar las cadenas y crear dos hijos.

Uno de los problemas de AGS son factores como el *elitismo*, y los modelos no generacionales que fueron añadidos para que exista una convergencia más rápida. Una de estas metodologías es el proceso de *selección*, la cual es un paso fundamental para el proceso de selección de candidatos a reproducirse. En el algoritmo genético este proceso es realizado de forma probabilista, esto implica que aún existe una probabilidad de que los individuos menos aptos sobrevivan. Pueden ser clasificadas en tres grupos: *selección proporcional, selección mediante torneo, y selección de estado uniforme* [42]. En la proporcional se eligen los individuos de acuerdo a su contribución de aptitud con respecto al total

de la población, y esta a su vez puede ser clasificada en tres grupos: *La ruleta, selección basada en rango, y la selección por torneo.*

Una vez seleccionado los individuos, sigue el proceso de cruza, que es un proceso complejo que ocurre entre parejas de cromosomas, estos se alinean, ajustan y se fraccionan en ciertas partes para posteriormente cambiar fragmentos entre sí. A pesar de que la técnica de cruza básica es aplicada en representaciones binarias, pueden ser extendidas bajo ciertas modificaciones. Se dividen en tres técnicas: *Cruza de un punto, cruza de dos puntos, cruza uniforme.* Como en el proceso de reproducción existe un factor proabilístico de que al cruzarse un individuo exista una mutación. En los algoritmos genéticos esto es implementado para que haya diversidad en la población y se mueva un poco entre el espacio de búsqueda, con el fin de que no quedar estancado en óptimos locales.

Algoritmo 1: Plantilla de un algoritmo genético

```
1  Elige una población inicial de cromosomas;
2  while la condición de parada no se cumpla do
3      do
4          if cumpla con la condición de cruza then
5              Selecciona a los padres;
6              Elige los parámetros de cruza;
7              Realiza la cruza;
8          else
9          end
10         if cumpla con la condición de mutación then
11             Elige los puntos de mutación;
12             Realiza la mutación;
13         else
14         end
15         Evalúa la aptitud de la generación;
16     while las suficientes generaciones son creadas;
17     Selecciona a la nueva población;
18 end
```

En la práctica, se suele recomendar porcentajes de mutación entre 0.001 y 0.01 para la representación binaria. Sin embargo investigadores han sugerido el usar porcentajes altos de mutación al inicio de la búsqueda, y luego decrementarlos exponencialmente, favoreciendo el desempeño de los AG [44]. Mientras que otros recomiendan que $pm = \frac{1}{L}$ (donde L es la longitud del cromosoma) [43].

Otra sugerencia es el utilizar auto adaptación, eligiendo una mutación fija que se vaya ajustando a lo largo de la ejecución, donde los rangos son codificados como genes extra en la representación del individuo, y generar la posibilidad de que puedan evolucionar.

Pese a la simplicidad el AGS aún es ampliamente utilizado, no sólo para propósitos educativos, y generar nuevos algoritmos para benchmark, también para problemas relativamente sencillos en los

cuales la representación binaria es posible. Al mismo tiempo ha sido extensamente modelado por investigadores, debido a que ha sido inspiración en el comportamiento de los procesos de evolución en espacios de búsqueda combinatoria [42], una metaheurística utilizada para atacar los problemas combinatorios son la búsqueda de vecindario [47].

2.7.2.2. Optimización con colonias de hormigas

Una de las estrategias bio-inspiradas que han demostrado ser eficientes con problemas combinatorios y de optimización, es la optimización con colonia de hormigas o en inglés *Ant Colony Optimization* (ACO). Fue introducido a la literatura por Colorni, Doringo y Maniezzo en 1992. Estos algoritmos están inspirados en las búsquedas que realizan las colonias de hormigas hasta la fuente de comida, ya que notaron la rapidez con las que las hormigas encuentran el camino más corto.

La información de que camino deben tomar para llegar a la comida se distribuye entre las hormigas dejando pedazos de información en el camino, llamada feromona [47]. Yu, Xinjie, & Gen, Mitsuo [48] definen feromona como la comunicación indirecta entre hormigas, la cual es una sustancia química que dispara la respuesta natural de otro miembro de la misma especie. Ellas escogen los caminos con 50 % de probabilidad debido a que ambos caminos contienen un rastro de feromona, cuyo valor es igual al inicio del algoritmo. Pero a medida que pasa el tiempo, una mayor cantidad de feromonas serán depositadas en el camino más corto ya que las hormigas al tomar ese camino llegan más rápido al nido. Tomando en cuenta que la feromona es una sustancia química que se evapora, esto provoca que el camino más corto a medida que pasa el tiempo genera más feromona, mientras que los otros caminos al no ser elegidos cada vez con más frecuencia, la sustancia se va evaporando hasta ser nula (véase Algoritmo 2).

La primera propuesta del trabajo de Dorigo et al. fue la publicación de ACO en 1996, el enfoque de su trabajo fue llamado *ant system* (AS), la cual es diferente a la mencionada con anterioridad la cual utiliza la actualización de una feromona local, así como la actualización de una feromona global, la metodología propuesta en el trabajo AS, solamente funciona con la actualización de una feromona local.

Rao [33] explica ACO como un proceso que puede ser representado por un problema de optimización como un grafo multicapa, donde el número de capas es igual al número de variables y el número de nodos en esa capa en particular es igual al número de valores discretos para la variable correspondiente. Este proceso puede ser explicado; una colonia consiste en N hormigas. Las hormigas empiezan en el nido, el camino a través de las distintas capas desde la primera hasta la última capa. Cada hormiga puede seleccionar solamente un nodo de cada capa de acuerdo a las reglas del estado de transición dadas. Los nodos seleccionados a través del camino visitado por una hormiga representan una solución posible. Una vez que el camino esté completo, las hormigas depositan feromona en el camino basado en la regla de actualización local. Cuando todas las hormigas completen su camino, la feromona en el camino mejor global es actualizado usando la regla de actualización global.

Al principio del proceso de optimización, todas las aristas o arreglos son inicializados con una cantidad de feromona igual. Todas las hormigas comienzan desde el nido y terminan en el nodo destino seleccionando cada nodo intermedio de manera aleatoria. El proceso de optimización termina cuando

el número máximo de iteraciones es alcanzado o ninguna solución mejor sea encontrada en un número sucesivo de ciclos o iteraciones. Podemos afirmar que el camino que contenga la cantidad más grande de feromona es considerado como el componente de la solución óptima [10].

Una vez inicializados los parámetros y los caminos de feromonas, el ciclo principal consta de tres pasos. Primero, m hormigas construyen la solución a la instancia del problema, bajo algún criterio, basada en la información de la feromona y la información heurística disponible. Puede ser mejorada en la fase de búsqueda local, y finalmente antes de comenzar la siguiente generación, los caminos de feromona se adaptan para reflejar la experiencia de búsqueda de las hormigas.

Algoritmo 2: Algoritmo ACO para problemas de optimización combinatoria

```
1 Inicializa individuos;
2 while i=1 a número de hormigas do
3 |   Construye la solución de la hormiga;
4 |   Aplica búsqueda local (opcional);
5 |   Actualiza feromona;
6 end
```

La elección de la solución agrega una probabilidad de ser realizada en cada paso de construcción. La regla más utilizada es la de *Ant System* (AS)

$$p(c_i^j \mid S_p) = \frac{t_{ij}^{\alpha} \cdot [\eta(c_i^j)]^{\beta}}{\sum_{c_i^l \in N(sp)} t_{il}^{\alpha} \cdot [\eta(c_i^l)]^{\beta}} \qquad \forall c_i^j \in N(s_p) \tag{2.1}$$

Donde $\eta(\cdot)$ es una función que asigna a cada solución $c_i^j \in N(s_p)$ un valor heurístico. El parámetro α y β determinan la influencia del comportamiento del algoritmo. El actualizar la feromona intenta crear soluciones que pertenezcan a buenas soluciones, dos mecanismos utilizados para esto es; lo primero es depositar la feromona, la cual incrementa el nivel de esta a la solución que está asociada el conjunto S_{upd} elegida de las soluciones buenas. La segunda es la evaporación del camino, en la cual decrementa la feromona depositada por las hormigas previas a medida que el tiempo avanza. Tratando de evitar una convergencia prematura hacia la región subóptima, favoreciendo la exploración de nuevas áreas en el espacio de soluciones. La actualización de la feromona es implementada como:

$$T_{ij} = (1-\rho)T_{ij} + \sum_{s \in S_{upd} | c_i^j \in s} g(s) \tag{2.2}$$

Donde S_{upd} es el conjunto de soluciones utilizadas para depositar feromona, $\rho \in (0, 1]$ es un parámetro llamado rango de evaporación, $g(\cdot) : S \longmapsto \Re^+$ es una función la cual f(s)¡f(s')$\Longrightarrow$g(s)$\geq$g(s'). Este determina la calidad de una solución y comúnmente es llamada *función de evaluación* [10].

2.7.2.3. Búsquedas locales aleatorias

Las heurísticas de técnicas con soluciones aproximadas, han sido ampliamente utilizadas para atacar los problemas combinatorios. En los años 70's se hizo claro que estos problemas pertenecían a la clasificación NP-Difícil. Uno de los más populares era la búsqueda local, estas comienzan de una solución inicial factible, y progresivamente la mejora a una solución factible que difiere muy poco de la actual, la búsqueda termina cuando encuentra el óptimo local (véase Algoritmo 3 y 4). La calidad de la solución obtenida así como el tiempo de cómputo depende de la riqueza de el conjunto de transformaciones (movimientos) en cada iteración [47]. En los algoritmos de búsqueda local aleatorio, se trabaja cada iteración con una solución s. Una nueva solución $s^{'}$ se construye a partir de s, eligiendo una solución del vecindario de esta. s es reemplazada por $s^{'}$ si esta no es inferior a s. La definición de vecindario es un parámetro crucial. Si es muy pequeño puede quedar estancado en óptimos locales, de lo contrario los individuos elegidos cercanos a la solución actual podrían tener una probabilidad baja de ser elegidos. Este tipo de problemas tienen un espacio de búsqueda $\{0,1\}^n$, y usualmente su vecindario se define como todos los puntos de búsqueda a una distancia hammiltoniana de 1 o 2 de la solución actual s [45].

Algoritmo 3: Búsqueda local aleatoria con un individuo

```
1 Escoge s ∈ {0,1}^n aleatoriamente (uniforme);
2 while condición de parada do
3 |   Escoge i ∈ {1,...,n} aleatoria y uniforme, ademas de cambiar el bit i de s;
4 |   if f(s' ≤ s) then
5 |   |   Reemplaza s con s';
6 |   else
7 |   end
8 end
```

Algoritmo 4: Búsqueda local aleatoria con dos individuos

```
Escoge s ∈ {0,1}^n aleatoriamente (uniforme);
Escoge b ∈ {0,1} aleatoriamente (uniforme);
while condición de parada do
    if b=0 then
        Escoge i ∈ {1,...,n} aleatoria y uniforme, ademas definimos s' cambiando el bit i de s;
    else
    end
    if b=1 then
        Escoge (i,j) ∈ {(k,l)|1 ≤ k < l ≤ n} aleatoria y uniforme, ademas definimos s'
         cambiando el bit i y j de s;
    else
    end
    if f(s' ≤ s) then
        Reemplaza s con s';
    else
    end
end
```

2.7.2.4. Búsqueda tabú

La búsqueda tabú originalmente fue propuesta por Fred Glover en 1986, y provee soluciones muy cercadas al óptimo local. Éstas estrategias incluyen una memoria a corto plazo para prevenir regresar a los movimientos recientes, declarándolo como tabú o prohibido, durante un número de iteraciones llamado *tenor del tabú* y una memoria de largo plazo para reforzar los componentes atractivos (véase Algoritmo 5). Estas memorias son llamadas *listas tabú*, son utilizadas para recordar algunas de las últimas transformaciones realizadas en la solución actual y prohibiendo regresar a ellas. Un elemento básico en la búsqueda tabú es la definición de su *espacio de búsqueda* y su *estructura de vecindario*. El espacio de búsqueda de las heurísticas de búsqueda local o búsqueda tabú es el espacio de todas las posibles soluciones consideradas (visitadas) durante la búsqueda. Es importante mencionar que no siempre es buena idea restringir el espacio de búsqueda sólo en soluciones factibles; en muchos casos, permitir a la búsqueda moverse a soluciones no factibles es deseable, y en algunos casos necesario. La estructura del vecindario son transformaciones locales que pueden ser aplicadas a la solución actual S, y define el conjunto de soluciones vecinas en el espacio de búsqueda N(S). Formalmente, N(S) es un subconjunto del espacio de búsqueda definido como:

$$\text{N(S)=\{Soluciones obtenidas aplicando una transformación local a S\}}$$

Para un problema en específico existen múltiples posibles estructuras de vecindarios de acuerdo a la definición del espacio de búsqueda. Las estructuras de vecindario utilizan movimientos "*Agrega/quita*" e "*intercambia*", ya sea abriendo un estado cerrado o cerrando uno abierto y moverse a través de los

estados abiertos. Las listas tabú usualmente son implementadas en listas circulares de tamaño fijo. Suponiendo que tenemos una función f(S) tratando de minimizar bajo algún dominio y le aplicamos la mejor versión de la búsqueda tabú, donde en cada iteración escoge al mejor movimiento disponible [47].

Algoritmo 5: Plantilla búsqueda tabú

```
1  Elige (construye) una solución inicial S_0 ;
2  S := S_0, f* := f(S_0), S* := S_0, T := φ ;
3  while la condición de parada no se cumpla do
4  |  Selecciona S en min[f(S')]; S'ε Ñ(S);
5  |  if f(S)¡f* then
6  |  |  f* := f(S), S* := S ;
7  |  |  Guarda en tabú para el movimiento actual en T (elimina la entrada más antigüa si es
   |  |   necesario);
8  |  else
9  |  end
10 end
```

Donde S es la solución actual, S^* es la solución mejor conocida, f^* es el valor de S^*, N(S) es el vecindario de S y $\tilde{\text{N}}$(S) es el subconjunto admisible de N(S) (no tabú). Los criterios de terminación comunes en la búsqueda tabú son:

- Después de un número fijo de iteraciones (o tiempo del CPU).
- Después de un número de iteraciones sin una mejora en el valor de la función objetivo.
- Cuando se alcanza el objetivo (un valor de umbral previo).

En esquemas complejos de tabú, la búsqueda usualmente termina después de completar una secuencia de *fases*, la duración de cada fase depende de alguno de los criterios anteriores. Una búsqueda tabú "regular"debe evaluar el objetivo para cada elemento del vecindario N(S) de la solución actual. Esto puede ser extremadamente costoso desde un punto de vista computacional. una alternativa es agregar aleatoriedad la cual puede actuar como un mecanismo anti-ciclo; esto permite una lista tabú más corta, que si se hiciera la exploración completa del vecindario.

Uno de los mayores problemas de los enfoques se búsqueda local es que tienden a pasar la mayoría del tiempo en un área restringida del espacio de búsqueda. Pese a que se pueden obtener buenas soluciones, puede fallar en la exploración de partes interesantes del espacio de búsqueda y terminar con soluciones muy alejadas del óptimo. La *diversificación* es un mecanismo que trata de relajar este problema forzándola realizar búsquedas en áreas inexploradas del espacio de búsqueda. Existen dos técnicas de diversificación: *reiniciar diversificación*, forzando a los componentes que han sido utilizados muy poco en la solución actual (o la mejor solución conocida) y reiniciar la búsqueda desde ese punto; y la *diversificación continua*, que integra ciertas diversificaciones en el proceso de búsqueda regular.

Una gran variedad de investigaciones recientes con búsqueda tabú se enfrentan con varias técnicas para realizar la búsqueda de una forma eficiente. Estos métodos incluyen explotar mejor la información, y que esté disponible durante la búsqueda y crear mejores puntos de inicio, así como operadores

de vecindario más eficientes y estrategias de búsquedas paralelas. Otra tendencia importante en la búsqueda tabú es la *hibridización* que son búsquedas tabú usada en conjunto con otros enfoques de solución como algoritmos genéticos [47].

2.7.2.5. Temple simulado

Temple simulado es una búsqueda local (metaheurística) con un fácil manejo para implementaciones, tienes propiedades convergentes y usa movimientos de ascenso de colina capaces de escapar de óptimos locales. Es llamado así debido a la analogía del proceso físico de enfriamiento de sólidos, el cual permite a un sólido cristalizarse si se calienta y después pasa por un proceso de enfriamiento lento hasta que alcanza su máxima configuración cristalina (estado mínimo de energía). Si el proceso de enfriamiento es lo suficiente lento, la configuración del sólido resultante es una estructura estable. Temple simulado establece la conexión entre el comportamiento de la termodinámica y la búsqueda por el mínimo global, para problemas de optimización discretos. En cada iteración el algoritmo genera valores para dos soluciones (solución actual y nueva solución) que son comparadas. Las soluciones que den mejores resultados son siempre aceptadas, mientras que las soluciones que no mejoren son aceptadas con la esperanza de escapar de óptimos locales, y la probabilidad de aceptarlas depende del parámetro de temperatura [47] [45](véase Algoritmo 6). Yu, Xinjie, & Gen, Mitsuo [48] mencionan la fórmula 2.3 que calcula la probabilidad.

$$p_t(i \longrightarrow j) = \begin{cases} 1, f(j) \leq f(i) \\ exp\dfrac{f(i) - f(j)}{t}, \quad \text{de lo contrario} \end{cases} \tag{2.3}$$

Donde t es el parámetro de control llamado temperatura. Si j no es peor que i, j toma el lugar de i con probabilidad de 1. De lo contrario, la probabilidad depende de que tan mala sea j y cual sea el valor de la temperatura. La temperatura del RS empieza alta y de acuerdo con algunas reglas de recocido se va decrementando.

Algoritmo 6: Plantilla de temple simulado

```
Elige una población inicial;
Seleccione el cambio de temperatura;
Seleccione el enfriamiento de la temperatura;
Seleccione una temperatura inicial;
Seleccione el número de iteraciones ejecutadas en cada temperatura M_k;
while la condición de parada no se cumpla do
    contador de repeticiones m=0;
    do
        Genera solución nueva que pertenece al vecindario;
        Calcula △ = f(w') − f(w);
        if △ ≤ 0 then
            w⟵w'
        else
        end
        if △ > 0 then
            w⟵w' con probabilidad exp(-△/t_k)
        else
        end
        m⟵m+1;
    while Hasta que m=M_k;
    k⟵k+1;
end
```

2.8. Cómputo paralelo

2.8.1. CUDA (Compute Unified Device Architecture)

Al encontrarse con una barrera de poder, el área arquitectónica se vio forzada a buscar nuevos paradigmas para mejorar el desempeño, por ello, la industria decidió que la forma de realizarlo era reemplazando los procesadores únicos por múltiples procesadores eficientes en el mismo chip [34]. Debido a esta necesidad en la ciencia e ingeniería, cada vez más GPUs son incluidos a CPUs para acelerar una amplia gama de computadoras, el cual es llamado "computo heterogéneo" y cada vez más interfaces gráficas multi-hilo son utilizadas.

Los desarrolladores de software necesitan competir con las diversas arquitecturas y plataformas de computo paralelo. Uno de los principales aspectos que han intentado mejorar es el desempeño de los dispositivos de computo utilizados por los usuarios, incrementando la velocidad con el que el reloj del procesador opera. Aunque el aumento de procesamiento no es el único método utilizado que ha sido mejorado, es una fuente confiable para mejorar el rendimiento. En años recientes la industria se ha visto obligada a buscar alternativas para el poder de procesamiento debido a las limitaciones en la

manufactura de circuitos integrados (temperatura y tamaño de los transistores). A comparación del análisis de información en los procesadores normales, el computo de propósito general en una unidad de procesamiento gráfico (GPU) es un nuevo concepto. Pero la idea de computo en un procesador gráfico ya ha sido implementado a principios de los años 90's, la compañía que popularizo la programación con GPU, además de liberar una interfaz de programación para este hardware, fue *Silicon Graphics*, liberando las bibliotecas de *OpenGL*. La cual es utilizada como una plataforma independiente estandarizada, para crear aplicaciones gráficas en 3D. Esto causó que por primera vez, los desarrolladores tienen algún control sobre el proceso de computo en el GPU [35]. El uso de estas tecnologías interactúa como co-procesadores para datos paralelos aritméticos.

La revolución del paralelismo hace más fácil escribir programas que son eficientes, portables y correctos. Tener múltiples procesadores más simples en un chip, simplifica el diseño del hardware y su verificación [34]. El poder de la unidad de procesamiento gráfico (GPU) de tu computadora, puede ser utilizado para escribir software de alto rendimiento para una amplia gama de aplicaciones [30].

Debido a que la única manera en que se podía interactuar con el GPU era por medio de APIs como OpenGL y DirectX. NVIDIA revelo el primer GPU construido con la arquitectura CUDA, con la que se liberaban de diversas limitaciones y restricciones de diseño vistas en los GPUs previos y enfocándose en cómputo de uso general. NVIDIA tomo el estándar C, agregando un pequeño numero de palabras reservadas llamado CUDA C, lo que hace que sea muy accesible para los programadores, además de proveer de un hardware especializado el cual permite explotar el poder de computo, demostrando que las aplicaciones utilizadas por los procesadores gráficos de NVIDIA tienen un mejor desempeño monetario y energético que otras implementaciones tradicionales.

Una de las aplicaciones hechas por la plataforma CUDA fue el estudio realizado por la universidad Temple junto con la industria Leader Procter & Gamble, el cual consistió en el uso de simulaciones moleculares con distintos materiales, permitiendo acelerar las pruebas tradicionales de los laboratorios, y extendiendo las pruebas con un gran número de variables respecto a las condiciones ambientales [35].

2.8.2. Arquitectura CUDA

A lo largo de los años, investigaciones han brindado grandes aportaciones en el área de computación, pero a medida que pasan los años es necesario aumentar cada vez más el poder temporal y espacial, por lo que distintos investigadores han propuesto que la solución no es aumentar los ciclos de reloj de los procesadores [35] [38], si no utilizar otras herramientas que permita extender estas barreras. Una propuesta muy aceptada en la actualidad es el uso de tarjetas gráficas (GPU), las cuales contienen miles de procesadores por lo que permite el cómputo paralelo con grandes volúmenes de datos. En generaciones pasadas de GPUs y sus APIs, tenían restricciones en los accesos a memoria, usualmente sólo permitían escrituras secuenciales en una matriz lineal. Esto debido a los límites en las APIs gráficas y los límites correspondientes en los procesadores, es decir, el acelerador no permitía el acceso a un elemento individual en matrices paralelas [36].

Por lo que NVIDIA decidió proponer la arquitectura CUDA con el cual construyó GPUs que pueden ser usados para tareas de gráficos tradicionales como cómputo para uso general, atacando dos paradigmas; computo heterogéneo y el paralelismo, por medio de una extensión de lenguaje C. NVIDIA

decidió unificar todas las formas del paralelismo en algo que llamó CUDA hilo (CUDA *thread*). Utilizando el nivel más bajo de paralelismo combinado con una programación primitiva. El compilador y el hardware puede manejar cientos de CUDA hilos juntos y utilizar varios estilos de paralelismo dentro del GPU, como multihilo (*multithread*), múltiples instrucciones múltiples datos (MIMD), una instrucción múltiples datos (SIMD) [19]. NVIDIA clasifica su modelo de programación como una instrucción múltiples hilos (SIMT). con los cuales le permite al programador enfocarse en explotar el potencial de las tarjetas [30].

La Arquitectura de CUDA incluye tuberías unificadas, permitiendo que cada unidad lógica aritmética (ALU) en el chip sea calculada por un programa de propósito general. Estas ALU fueron construidas para cumplir los requerimientos para la precisión flotante aritmética. Por otro lado las unidades de GPU se les permite leer y escribir accesos de memoria arbitrarios así como el acceso de software manejador de caché conocido como memoria compartida. Además NVIDIA proporcionó un controlador de hardware especializado para explotar el poder computacional de su arquitectura [35].

Las plataformas de cómputo GPU NVIDIA cuenta con una familia de productos: *Tegra, Geforce, Quadro, y Tesla*. El prodcto Tegra fue diseñado para dispositivos móviles y embebidos como tabletas y teléfonos, Geforce es para uso gráfico, Quadro es para visualización profesional, y tesla para el uso de computo paralelo (Figura 2.2).

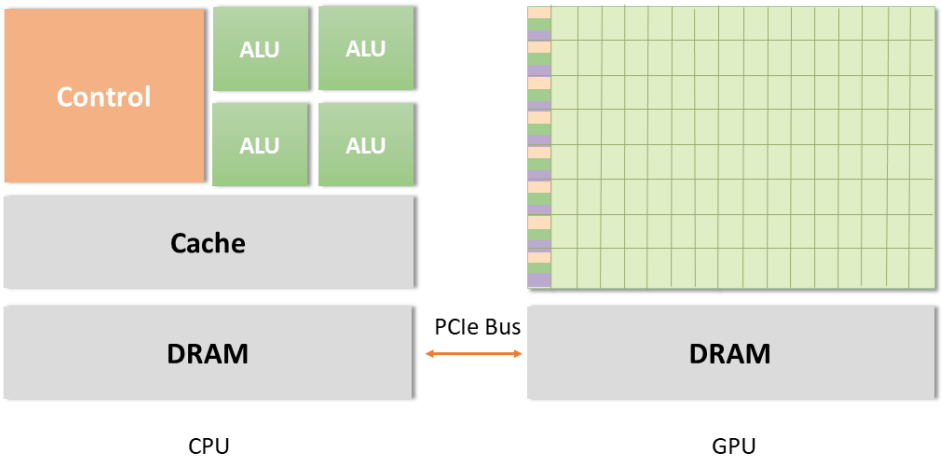

Figura 2.2: Arquitectura Tesla

CUDA implementa una arquitectura heterogénea, la cual consiste en dos multi-núcleos CPU y dos o más multi-núcleos GPUs, donde el GPU (*device*) es una plataforma que co-procesa junto con el CPU (*host*) a través de un bus, como se muestra en la figura 2.3. El código del CPU es el responsable del manejo del entorno, código, y datos para el dispositivo antes de cargar los datos en el mismo. Mientras que el GPU es usado para acelerar la ejecución de una porción de los datos de manera paralela [30]. Un aspecto único de esta arquitectura es su flexibilidad de accesos a los recursos locales como los registros o memoria local hacia los hilos. Cada "*streaming multiprocessors* (SM)" es capaz de ejecutar un número de hilos y los recursos locales se dividen entre los hilos.

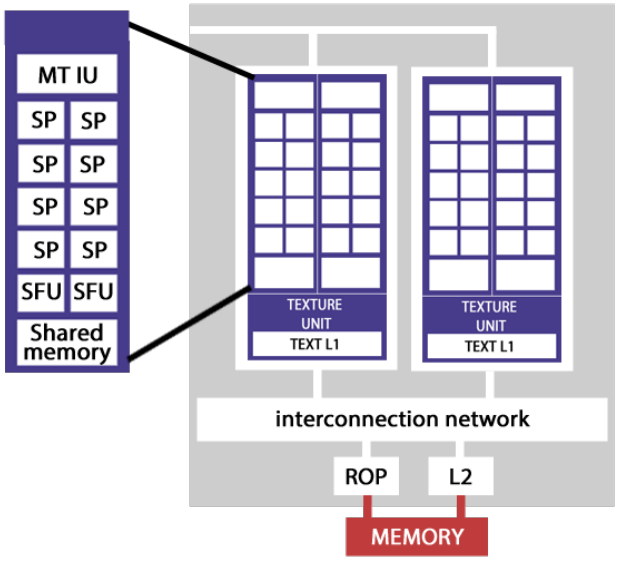

Figura 2.3: Arquitectura heterogénea [30]

El cómputo en tarjetas gráficas no pretende reemplazar el cómputo en CPU. El cómputo en CPU es utilizado para tareas con un mayor control, debido a su habilidad en el manejo de lógica compleja y su paralelismo en su nivel de instrucciones, mientras que el cómputo en GPU es preferido para tareas con datos paralelos y extensos, gracias a su gran número de núcleos programables, soportando multihilo, y a que tiene un extenso ancho de banda en la precisión de punto flotante (*peak*), comparado con el CPU. El combinar CPU con GPU permite atributos complementarios para un mejor rendimiento de distintas aplicaciones, usando dos tipos de procesadores. Cuando se busca la calidad en el rendimiento es necesario utilizar ambos, donde el CPU ejecuta las partes secuenciales, mientras que en el GPU ejecuta los datos extensos de forma paralela [30].

2.8.3. Modelo CUDA

La plataforma CUDA puede ser accedida a través de bibliotecas CUDA, directivas de compiladores, interfaces de programación de aplicaciones (API), y extensiones de lenguajes de programación como: *C, C++, Fortran, y Python* (Figura 2.4). CUDA C es una extensión de estándar ANSI C, combinándolo con la programación heterogénea, así como el manejo sencillo de APIs, memoria, entre otras.

CUDA provee dos niveles de APIs para el manejo de los dispositivos GPU y organización de hilos: CUDA *driver* API, la cual es de bajo nivel y difícil de programar, pero brinda un mayor control, y CUDA *runtime* API la cual es de alto nivel que se implementa encima del API *driver* [30].

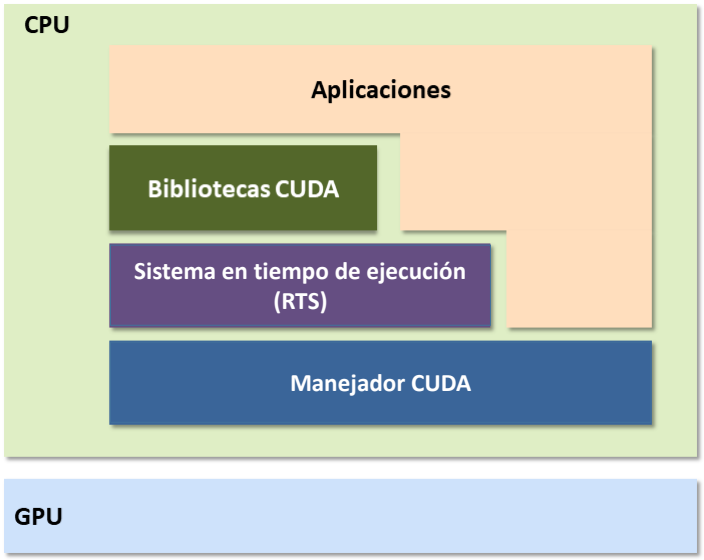

Figura 2.4: API CUDA [30]

Un programa en CUDA consiste en una mezcla de las siguientes partes: el código *host* el cual se ejecuta en CPU y el código del dispositivo, que se ejecuta en el GPU. NVIDIA utiliza su propio compilador *nvcc* [30], este separa el código del dispositivo del código *host* durante el proceso de compilación. Por otro lado, el código *host* es código en estándar C y utiliza los compiladores de C. Mientras que el código del dispositivo está escrito por una extensión de lenguaje C con palabras reservadas para funciones de datos paralelos, llamados *kernels*.

El objetivo de los compiladores NVIDIA es proporcionar una abstracción de la arquitectura de las tarjetas aceleradoras gráficas (SMX, cuda cores, memoria, entre otras.) y hacer un mapeo de instrucciones de bajo nivel PTX (*Parallel Thread Execution*), incluyendo un conjunto de instrucciones estables para los compiladores, además de ser compatible a través de las generaciones de GPUs. PTX utiliza registros virtuales, para que el compilador los utilice como múltiples vectores de registros físicos, que un hilo SIMD (*single instruction multiple data*) necesita, mientras que se optimiza la disponibilidad de los registros equitativamente entre los hilos SIMD [19] (Es posible observarla en la Figura 2.5).

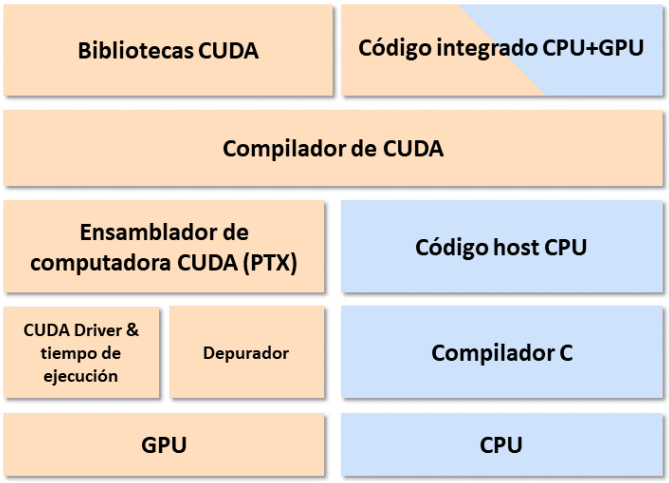

Figura 2.5: Programa en CUDA [30]

Además de lo anterior, NVIDIA introdujo a su modelo de programación una mejora que llamó *memoria unificada*, la cual actúa como un puente entre los espacios de memoria del *host* y el dispositivo. Permitiendo el acceso de apuntadores de la memoria de CPU y GPU, mientras que el sistema realiza la migración de datos automatizada entre el *host* y el dispositivo. Este manejo por parte del programador de la memoria y los datos permite optimizar las aplicaciones así como maximizar el potencial del hardware. El modelo de programación de CUDA es asíncrona para que el trabajo realizado por el GPU pueda ser traslapada junto con las comunicaciones entre *host*-dispositivo [30].

2.8.4. Manejo de memoria CUDA

CUDA ofrece una jerarquía de memoria en la arquitectura del GPU. Puede ser dividida en dos: *memoria global*, la cual es análoga a la memoria del sistema del CPU; y *memoria compartida*, que es similar a la caché del CPU. Además, cada dispositivo GPU tiene un tipo de *memoria particular*, usada para diferentes propósitos.

Esta jerarquía de memoria consiste en múltiples niveles de memoria con diferentes latencias, anchos de banda y capacidades [30]. La figura 2.6 muestra la jerarquía de los espacios de memoria. cada una con diferente alcance, tiempo de vida, y comportamiento caché. Un hilo en un *kernel* tiene su propia memoria local. Un bloque de hilos tiene su propia memoria compartida, visible para todos los hilos dentro del bloque, y se mantiene durante el tiempo de vida del bloque. Todos los hilos pueden acceder a la memoria global.

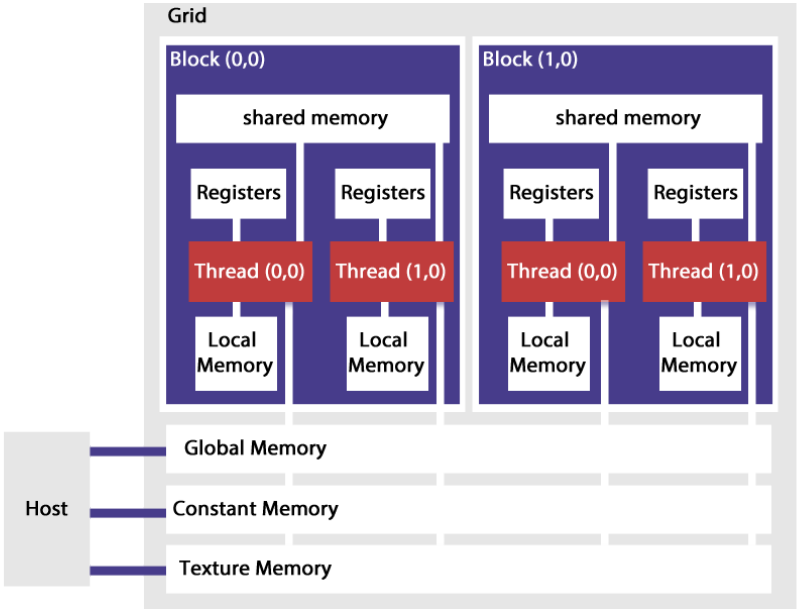

Figura 2.6: Jerarquía de memoria [30]

Los registros son los espacios de memoria que pueden ser accedidos con mayor rapidez dentro del GPU. Dentro de los registros se guardan las variables y los arreglos declarados dentro del *kernel*, estas variables son privadas por cada hilo. Los registros comparten el tiempo de vida con el *kernel*, una vez terminada su ejecución, el registro no puede ser accedido de nuevo. Los registros son recursos repartidos en *warps* listos en un SM.

La memoria compartida, debido a que se encuentra dentro del chip, es mucho menor su latencia que el de la memoria global y local. Es similar a L1 caché, además de ser programable. Cada SM tiene una cantidad limitada de memoria compartida, debido a que se reparte para todos los hilos dentro del bloque. La memoria compartida es declarada dentro de la función *kernel*, pero una vez que el bloque de hilos termina su ejecución, su asignación dentro de la memoria compartida será liberada y asignada a otro bloque de hilo. Los hilos dentro del bloque pueden ser cooperativos gracias a que utilizan datos compartidos. El acceso a la memoria compartida debe ser sincronizada, por medio de la variable *void _syncthreads()*. Esta función crea una barrera y sólo permite continuar la ejecución hasta que todos los hilos del bloque lleguen a ella, evitando múltiples accesos de escritura en el mismo espacio de memoria por diferentes hilos.

La memoria constante se encuentra dentro del dispositivo, puede ser llamada mediante la siguiente variable: *__constant__*, puede ser accedida por todos los hilos. Esta memoria da el mejor rendimiento, cuando los cálculos a realizar por diferentes datos son los mismos dentro de un *warp*, es decir cuando cada hilo dentro del warp lee un sólo registro.

La memoria de textura también se encuentra dentro del dispositivo, es de sólo lectura. Este tipo de memoria es muy parecida a la memoria global, debido a que puede ser accedida por una cache de lectura. La memoria de textura está optimizada para la localización de espacios de memoria en 2D, y que los hilos dentro del *warp* que utilizan accesos de datos 2D den el mejor rendimiento.

La memoria global es la más utilizada por el GPU y con mayor latencia. La variable en memoria global se puede declarar de manera estática y dinámica. Para declararla estática en el dispositivo se utiliza la siguiente palabra reservada: *__device__*, cuando es utilizada en el *host* se utiliza la palabra reservada *cudaMalloc* y se libera usando *cudaFree*. Punteros a la memoria global se pasan a las funciones kernel como parámetros. Se debe tener cuidado al utilizar múltiples hilos dentro de la memoria global, debido a que no es posible sincronizar los bloques, lo que causaría que accedan y modifiquen la información. Para optimizar las transacciones de memoria deben ser múltiples de 32, 64 y 128 bytes [30].

2.8.5. Otras tecnologías para cómputo paralelo

La idea del usar un ambiente multiGPU fue tomando como una vertiente una vez propuesto la programación con propósitos generales, trayendo consigo un aumento considerable en la rapidez. La evolución del cómputo en GPUs ha ido en aumento los últimos años permitiendo acelerar el código en diferentes dominios, como acoplamiento molecular hasta análisis médicos y criptografía. Hoy en día el proceso multi-núcleo abarca más y más el ámbito de mercado, enfrentando con coaliciones generadas por su llegada, como los límites de escalamiento de semiconductores, el análisis de poder asociada, el reto respecto a temperatura y el pobre rendimiento al incrementar los microprocesadores dentro de un núcleo. Por lo cual, los programadores vieron la oportunidad de resolverlo por medio de el cómputo paralelo, mejorando el rendimiento del los chip multi-núcleo. Entre algunos de los métodos más conocidos se encuentra OpenMP, MPI y CUDA [37].

En la siguiente tabla (2.1) muestra las tecnologías utilizadas para paralelizar, como lo son: MPI, OpenMP y CUDA. Generalizando las principales diferencias que existen entre ellas para realizar una comparación visual y de ser posible generar los argumentos necesarios sobre por que surge la necesidad de utilizar estas nuevas tecnologías, así como la tendencia que existe hacia CUDA, debido al amplio desarrollo y disponibilidad a nivel mundial.

CUDA	MPI	OpenMP
Se caracteriza por su tecnología de GPUs de propósito general, así como por APIs para diferentes lenguajes de programación.	Se caracteriza por su modelo de programación basado en paso de mensajes.	Se caracteriza por su modelo de programación basado en memoria compartida.
Múltiples jerarquías de memoria.	Uso de memoria distribuida.	Uso de memoria compartida.
Buen rendimiento y fácil de programar.	Buen rendimiento cuando se cuenta con redes de alto desempeño.	Bajo rendimiento derivado de implementar la abstracción de memoria compartida.
Especializada para el modelo SIMT (Single Instruction Multiple Thread).	Especializada para el modelo MIMD (Multiple Instructions Multiple Data).	Especializada para el modelo SPMD (Single Program, Multiple Data Stream).
Soporta lenguaje de nivel alto y bajo.	Uiliza lenguaje de nivel alto.	Uiliza lenguaje de nivel alto.
Especializada en la arquitectura NVDIA.	Independiente de la plataforma.	Independiente de la plataforma.
Especializada para arquitecturas multi-núcleo.	No se especializa en arquitecturas multi-núcleo.	No se especializa en arquitecturas multi-núcleo.

Tabla 2.1: Tabla comparativa respecto a las distintas tecnologías paralelas.

Capítulo 3

Trabajos Relacionados

3.1. Algoritmos exactos

Como se menciona en capítulos anteriores, el problema de coloreado de grafos se encuentra dentro de la clase NP-Completa [1], el problema de determinar el número cromático de un grafo arbitrario, dentro del radio el peor escenario es menor a $O(n^{\epsilon})$, $\epsilon > 0$ (también es NP-Completo). Una de las metodologías más populares y eficientes para resolver el problema de coloreado de grafos, es a través de enumeración implícita (*implicit enumeration*), el primer algoritmo utilizando este enfoque fue el propuesto por Brown [49], este algoritmo funciona coloreando los nodos de forma secuencial de acuerdo a un pre-orden específico, para encontrar alternativas de coloreados factibles por medio de *backtraking* y obtener el punto de partida hacia la siguiente fase.

Si tomamos un grafo G=(V,E), en el primer paso obtenemos un coloreado factible utilizando *ub* colores obtenidos, después de que los nodos del grafo G han sido ordenados por un criterio. Una vez que los nodos han sido ordenados, a cada nodo se le asigna el color menos factible, utilizando *ub* como una cota superior, el algoritmo realiza el *backtraking* en orden inverso de los nodos, hasta encontrar el nodo v_j, que es re-coloreable por un color alternativo factible ($< ub$) que no ha sido utilizado por v_j. En el último caso un coloreado mejor es encontrado y *ub* es actualizado. El algoritmo termina cuando el *backtraking* alcanza al nodo v_1.

Una versión modificada del algoritmo de Brown, es el utilizado por Brelaz & Korman, en el cual para mejorar la eficiencia utiliza el concepto de re-ordenamiento dinámico (*dynamic reordering*), la idea principal de este es escoger el siguiente nodo (a un color) con el menor número de asignación de colores factibles. Cuando el nodo es coloreado, el algoritmo escoge el color que es inadmisible para el gran número de nodos no colorados. Lo anterior permite reducir considerablemente el tamaño del árbol de búsqueda [50].

3.2. Algoritmos heurísticos

el objetivo de las heurísticas es encontrar un buen coloreado en un tiempo razonable. Dentro de esta categoría existen dos enfoques relacionados a resolver el problema de coloreado de grafos utilizando

grandes instancias de grafos, la primera es el algoritmo de aumento sucesivo (*Successive Augmentation Algorithms*) y algoritmos de mejora iterativa (*Iterative Improvement Algorithms*) [50].

La razón por la cual es necesario incrementar el número de heurísticas para el problema de coloreado de grafos, es debido a que el implementar los algoritmos exactos solo alcanza a lidiar con instancias pequeñas del problema, mientras que los grafos que se basan en escenarios reales contienen miles o incluso millones de nodos y aristas, para los cuales es necesario soluciones factibles en tiempo razonable.

3.2.1. Coloreado heurístico greedy secuencial

El algoritmo colorado heurístico greedy secuencial SGCH por sus siglas en inglés, consiste en extender un coloreado parcial (puede estar vacío) aumentando sucesivamente el número de nodos coloreados (color a un nodo no coloreado), una vez que un color ha sido asignado a un nodo, ya no es posible cambiarlo. Regularmente este algoritmo primero realiza un ordenamiento a los nodos bajo cierto criterio. De tal manera que cada nodo es coloreado en un orden específico, con el color más pequeño posible. La calidad del coloreado del algoritmo depende del ordenamiento previo de los nodos [50].

Múltiples esquemas de ordenamiento han sido propuestos y probados. Por ejemplo el Welsh & Powell [52], en el que ordena los nodos de forma decreciente de acuerdo a su grado, llamando a esta heurística *Largest First* (LF). Mientras que *Largest First* ordenado por Matula [53] propone el ordenamiento de los nodos en base a los siguientes pasos:

1. v_n sea un vértice de grado mínimo en G=G_n.
2. Para $j = n - 1$ a 1, v_j sea un vértice de grado mínimo en G_j.

Otra versión del algoritmo SGCH es aquella que no utiliza un pre-orden en los nodos, este algoritmo es llamado heurística Dsatur de Brelaz [51], el siguiente nodo a colorear es el que tenga el grado de saturación máxima. El nodo cuyo color del vecino contiene el mayor número con colores diferentes. Dsatur colorea un nodo con el mínimo número de colores factibles. Existe otro algoritmo que no necesita un pre-orden de los nodos propuesto por Leighton llamado *Recursive Largest First* (RLF), este genera clases de colores sucesivas. Funciona de manera similar a Dsatur, debido a que el coloreado del nodo siguiente es definido durante el proceso de coloreado. Para generar una clase de color, el algoritmo primero introduce dos conjuntos V_R y U, los cuales forman una partición de los nodos no coloreados. Inicialmente $U = \emptyset$, V_R contiene todos los nodos no coloreados, el conjunto U se compone de todos los nodos no coloreados que no pueden ser miembros de la clase de color en construcción [50]. Los siguientes dos pasos son utilizados por RFL para generar las clases de colores:

Paso 1: Escoge el nodo v con el grado máximo en el subgrafo G(V_R) realizada por V_R. Coloca v en S_j y mueve todas las u $\in V_R$ con (u,v) $\in E$, de V_R a U.

Paso 2: Mientras $V_R \neq \emptyset$:
Escoge un nodo $v \in V_R$ que contenga el mayor número de vecinos en U y agrega v al conjunto S_j.
Mueve todos los vecinos de v del conjunto V_R a U.

Utilizando la idea de Johri & Matula, Johnson [54] propuso un algoritmo de aumento sucesivo llamado XRLF, el cual es una generalización de la versión heurística RLF con algunos parámetros de control. El primer paso del algoritmo es construir múltiples conjuntos independientes que serán utilizados como candidatos potenciales de las clases de colores. Elige (de forma iterativa) un conjunto independiente como la siguiente clase de color que al remover minimiza la densidad de las aristas del subgrafo. Cuando el número de nodos sin color restantes es relativamente pequeño, el algoritmo cambia hacia una búsqueda exhaustiva. Este algoritmo ha sido probado en grafos aleatorios $G_{n,p}$ con $n \geq 1000$ y $p = 0.5$. Los resultados mostraron que XRLF consume un tiempo considerable del CPU y genera resultados prometedores [50].

3.2.2. Mejora iterativa heurística

La heurística *greedy* iterativa de coloreado, IG por sus siglas en inglés, fue propuesta por Culberson [55] y esta aplica el algoritmo SGCH de forma iterativa para nodos ordenados. La idea de IG se origina al observar la construcción los colores realizado por el algoritmo SGCH a cada nodo ordenado, no será deteriorado por el anterior. Múltiples métodos de ordenamiento han sido propuestas y probadas. Pero se ha demostrado que en general el combinar diferentes esquemas de ordenamiento es mucho más efectivo que el ordenamiento puro. Lo anterior debido a que al combinar diferentes esquemas permite al IG encontrar vecindarios más grandes y como consecuencia la calidad de las soluciones tienden a ser mejores [50].

3.3. Algoritmos metaheurísticos

3.3.1. Búsqueda de vecindario variable

La búsqueda de vecindario variable o *variable neighborhood search* (VNS), fue propuesto por Mladenovic y Hansen en 1997. VNS se basa en la idea de un cambio dinámico del vecindario dentro de la búsqueda local. Procede por una metodología descendente que a través del vecindario hacia la exploración del óptimo local. Este método se mueve de la solución actual hacia una nueva, si es que implica una mejora en ella. De esta manera, la variable óptima se mantiene obteniendo vecindarios prometedores [58].

Avanthay [56] en 2003 introdujo una adaptación del método VNS para el problema de coloreado de grafos. Proponiendo una técnica que probó ser efectiva en determinar soluciones de alta calidad.

3.3.2. Búsqueda tabú

Este algoritmo fue propuesto por Glover en 1986, la búsqueda tabú o *tabu search* (TS) es una metaheurística que guía la búsqueda local, permitiendo escapar de óptimos locales además de implementar un esquema de exploración. TS simple aplica una mejora dentro de la búsqueda local donde en cada iteración la mejor solución del vecindario es seleccionada como la nueva solución actual. Una memoria a corto plazo junto con la lista tabú cuyos atributos de las soluciones son guardadas para evitar ciclos en términos cortos [58].

Gonzalez-Velarde [57] en el 2002 aplicó búsqueda tabú para tratar de resolver el problema de coloreado de grafos. En su artículo se menciona que el método propuesto fue mediante un método de búsqueda tabú que aplica una versión simple de cadenas de expulsión para colorear grafos.

3.3.3. Recocido simulado y búsqueda tabú

Estas se componen de métodos de búsqueda tradicionales que son diseñados para mejorar la solución actual, realizando el moviento desde una solución factible hacia una mejor dentro del espacio de soluciones hasta que ninguna otra solución sea encontrada dentro del vecindario. Una de las trampas más comunes dentro de esta metodología es cuando el espacio de soluciones está compuesto por múltiples puntos de óptimos locales, un punto óptimo x^* en un vecindario pequeño de x^*. Una vez que la búsqueda se mueve a través del vecindario se queda ahí hasta que falla al encontrar una solución óptima [50].

Para evitar este tipo de trampas, muchas metodologías recientes de búsquedas usan un mecanismo de "*escape*" o "*saltos*" fuera del vecindario. Estos saltos pueden deteriorar el valor objetivo de forma temporal. Sin embargo, guían la búsqueda hacia diferentes áreas dentro del espacio de soluciones.

Recocido simulado y la búsqueda tabú son dos técnicas heurísticas que son diseñadas para escapar de óptimos locales y evitar ciclos. Recocido simulado fue propuesto por Metropolis para simular el proceso de enfriamiento de materias sólidas después de haber sido calentadas al punto que son derretidas. Las propiedades estructurales de enfriamiento en los materiales sólidos puede ser controlado mediente una variable de temperatura durante el proceso de enfriamiento.

Múltiples heurísticas de recocido simulado han sido propuestas para resolver el problema de coloreado de grafos, por ejemplo, Chams [59] presentó una heurística de recocido simulado puro y una heurística que combina recocido simulado con RLF llamado SA-RLF (por sus siglas en inglés), primero utilizando RLF para construir las clases de colores hasta que un número de nodos sin colorear alcanza un nivel específico. Una vez alcanzado el nivel, SA-RLF cambia a colorear el grafo restante mediante la heurística de recocido simulado. Los resultados computaciones de SA-RFL se comparan con los resultados de Dsatur de Brélaz y RFL de Brown, de acuerdo a los resultados de Chams SA-RFL parece brindar mejores resultados en términos de tiempo del CPU, y soluciones de buena calidad [50].

3.3.4. Algoritmos híbridos

Este tipo de algoritmos o técnicas son conocidos por combinar múltiples estrategias ya sean del mismo tipo o diferentes para resolver un problema, también son llamados algoritmos meméticos.

Galinier [9] en 1999 realiza una propuesta utilizando un algoritmo híbrido para el problema de coloreado de grafos, combinando nuevas clases de operadores de cruza especializadas. Los resultados experimentales demostraron que el algoritmo de coloreado híbrido garantiza una mejora en los resultados dentro de su operador de búsqueda tabú, demostrando la importancia de la cruza dentro del proceso de la búsqueda. Además de realizar experimentos acerca del comportamiento del algoritmo referente a la evolución de la función de evaluación y la diversidad de la población. Al principio es evidente la necesidad de preservar la diversidad de la población para que la búsqueda sea eficiente.

Concluyendo con la afirmación del gran potencial que existe al utilizar este tipo de algoritmos para problemas combinatorios.

Ehsan [60] en 2008 utiliza una metaheurística muy conocida llamada optimización de colonia de hormigas ajustado a la estructura del sistema de hormigas Max-Min, explotando la búsqueda local para mejorar el rendimiento. A la vez utiliza una función de evaluación para distinguir entre las soluciones coloreadas. Los pasos generales del algoritmo es generar la fase de construcción mediante el uso de colonia de hormigas produciendo coloreados parciales no factibles y posteriormente aplica una búsqueda local adecuada transformándolos en soluciones factibles y mejorarlas.

Zhipeng [61] en 2010 propone un algoritmo de metaheurística híbrida la cual integra búsqueda tabú dentro de un algoritmo evolutivo para resolver el problema de coloreado de grafos. Destacando la importancia de la diversidad de los individuos, para ello utiliza una técnica greedy para cruzar a los individuos, en inglés es llamada *greedy partition crossover* (GPX) proponiendo que el operador de cruza sea multi-padre adaptativo el cual demuestra obtener mejores resultados que con un operador de cruza de dos padres además de proponer una nueva función de "calificación de bondad"que considera tanto la calidad como la diversidad de los individuos.

3.3.5. Algoritmos paralelos

El procesamiento paralelo es definido como ejecución concurrente o simultánea de instrucciones en una computadora. La motivación obvia de la paralelización es incrementar la eficiencia del procesamiento. Existen múltiples aplicaciones que requieren de grandes cantidades de procesamiento y análisis.

Lewandowski [62] en 1994 introdujo un algoritmo híbrido para el coloreado de grafos el cual combina una versión paralela del algoritmo Morgenstern *S-Impasse* con una búsqueda exhaustiva *Branch and bound*. Dicho algoritmo híbrido fue implementado en una máquina de conexión (*connection machine* CM-5), mientras que los datos fueron analizados por la aplicación, así como el uso de grafos generados de forma aleatoria. Lewandowski menciona que los resultados obtenidos pueden ser comparados con dos heurísticas secuenciales simples: el algoritmo de saturación de Brélaz (Dsatur) y el algoritmo de Leighton RFL (*Recursive Largest First*), además de mejorar el trabajo reportado previamente por Morgenstern y Johnson. En general el algoritmo propone que un gestionador de procesos empiece realizando la búsqueda exhaustiva paralelizada dentro de un grupo de procesadores, mientras que en otro grupo en paralelo se realice la heurística *S-Impasse*.

Por otro lado, Szymon [63] en 2007 propuso otro algoritmo paralelo utilizando recocido simulado para el problema de coloreado de grafos. Para ello utilizó múltiples procesadores trabajando de forma concurrente en cadenas individuales con el objetivo de minimizar la función de costo y guardar la mejor solución encontrada. La coordinación del algoritmo es por medio de un modelo esclavo-maestro en el cual un procesador es es responsable de recolectar las soluciones, elegir la solución actual y distribuir la cantidad de unidades esclavas. Los experimentos revelaron que el rendimiento depende mucho en escoger adecuadamente el proceso de enfriamiento, así a la generación inicial con un coloreado que se ajuste al problema. Además de demuestra que el modelo utilizado dentro del algoritmo alcanza su punto óptimo cuando el número de esclavos es relativamente pequeño.

Capítulo 4

Propuesta de solución al problema

4.1. Introducción

Como se menciona en el capítulo 2, en la actualidad existen múltiples problemas y variaciones de los mismos, que al resolverlos generan una respuesta que puede ser utilizada en la vida cotidiana. Hace años hubiera sido imposible el pensar que una computadora fuera capaz de resolver estos problemas, debido a que el espacio de búsqueda real es discontinuo y multi-modal, además de que el espacio computacional era mucho más limitado. Hoy en día existe un incremento considerable en este aspecto, además de haber nuevas tecnologías, así como refinamiento y optimización en los algoritmos. Los problemas de la vida díaria se encuentran clasificados en problemas NP-Difíciles. El área de computación es uno de los fundadores en tratar de clasificar los problemas, con el objetivo de tener claro cuales son aquellos que son capaces de resolver y aquellos que no. En la clasificación de problemas difíciles (NP-Difícil) dominan aquellos que hasta la fecha no existe una forma de verificar si una solución es correcta (a menos que P=NP). Los problemas combinatorios se encuentran dentro de esta categoría, pero debido a lo anterior, existen diferentes propuestas de investigación que brindan resultados satisfactorios en tiempo polinomial, entre los más conocidos se encuentran los algoritmos *greedy*, estos ofrecen soluciones rápidas, y al aplicarlas con instancias pequeñas brindan incluso soluciones óptimas. La literatura menciona que además de esa, existen otras técnicas que se han puesto a prueba para resolver problemas combinatorios, como lo son *las búsquedas locales*, *construcción de clase de colores*, *algoritmos evolutivos*, *heurísticas*, *redes neuronales* y *estrategias híbridas*. Las búsquedas locales es una de las que más se han puesto a prueba para tratar de dar buenas soluciones a este tipo de problemas, el desempeño de estas técnicas dependen mucho de su función de vecindario, por lo que dicha función debe caracterizar el problema. Una propuesta que ha brindado muy buenos resultados en el área, son las estrategias híbridas, las cuales combinan una o más estrategias para tratar de dar una solución. Debido a los buenos resultados y a la gran aceptación de este tipo de técnicas, surgieron los *algoritmos evolutivos híbridos* o *meméticos*, los cuales combinan la adaptación evolutiva con el aprendizaje individual mediante el refinamiento local.

El problema de coloreado de grafos es uno de los problemas NP-Difícil clasificado por Karp [1], este se define por medio de un grafo no dirigido en el que cada par de nodo vecino o adyacente (que se encuentre unido por una arista) no se clasifique en el mismo color. Al obtener una solución buena

y factible en este problema, da lugar a la posibilidad de generar una respuesta a problemas cotidianos como: *construcción de tablas de horarios*, *asignación de frecuencias*, *colocación de registros*, entre otras, en un tiempo polinomial.

En la actualidad para tratar de extender la barrera de instancias que pueden ser resueltas en este problema, los científicos han jugado con la capacidad de procesamiento del *hardware*, así como el considerar el utilizar *hardware* cuya finalidad no consiste en la computación de propósito general. Una de las opciones más utilizadas y aceptadas en la actualidad son las tarjetas gráficas, considerando la paralelización como una opción viable en la cual se divide el problema en diferentes GPUs realizando diferentes tareas simultáneamente, con esto se logra reducir el tiempo y por lo tanto aumentar la posibilidad de instancias analizadas dentro del espacio de búsqueda. Las tarjetas más utilizadas y estudiadas son las de $NVIDIA$ llamadas $CUDA$ cuya ventaja también es el proporcionar una jerarquía de memorias disponibles en la arquitectura, así como la gran variedad de compatibilidad dentro de los lenguajes de programación.

A continuación se muestra la propuesta de este trabajo de tesis, la cual utiliza técnicas híbridas, como lo es el uso de estrategias greedy dentro del algoritmo genético híbrido en conjunto con el refinamiento individual mediante dos estrategias de búsqueda local: *metrópolis* y *ascenso de colina*, cada una realizada de forma asíncrona mediante GPUs de la tarjeta CUDA. Además de proponer el estudiar las distintas jerarquías de memoria que brinda CUDA (*memoria global*, *memoria compartida*, *memoria local*), así como distintos bloques, hilos y tamaños de población, con el propósito de observar el comportamiento de los algoritmos genéticos híbridos, secuenciales, paralelos (con cada configuración CUDA), utilizando ambas técnicas locales.

4.2. Propuesta

Como se menciona anteriormente, los problemas combinatorios difíciles se clasifican como problemas NP-Difícil, en particular el problema combinatorio de coloreado de grafos [13], el cual consiste en colorear los vértices de un grafo con un mínimo número de colores y cuya restricción es que para cada par de nodos vecinos o adyacentes, sus colores sean diferentes. Debido que es un problema conocido e importante en múltiples aspectos cotidianos, se han propuesto diferentes alternativas para resolverlo incluyendo *algoritmos greedy, algoritmos de búsqueda local, algoritmos genéticos, y algoritmos híbridos*. Los algoritmos genéticos por lo regular son búsquedas aleatorias con operadores de cruza y mutación, capaces de resolver problemas cuyos espacios de búsqueda son codificados como cadenas de bits aplicados dentro de espacios continuos, pero una vez utilizado dentro de un espacio discreto los investigadores se dieron cuenta que no daba buenos resultados.

Una de las alternativas dentro del análisis de metaheurísticas son los algoritmos híbridos, los cuales combinan técnicas evolutivas con refinamiento local, estas forman parte de los más utilizados y estudiados dentro del ámbito de algoritmos evolutivos. Además de demostrar que generan soluciones de buena calidad para el problema de coloreado de grafos. Por esto Galinier [9] propuso utilizar estas estrategias mediante una representación del individuo basada en un enfoque de *partición*, donde un conjunto de vértices se considera que pertenecen a la misma clase y una configuración es considerada como una

partición de vértices en clases de colores, es decir, que la representación del individuo es mediante k número de bolsas de colores y en cada bolsa contiene los nodos que pertenecen a ese color. De esta forma, en la cruza se requiere transmitir clases de colores o subconjuntos de clases de colores, todo esto en conjunto con los algoritmos evolutivos híbridos, en los cuales combina el algoritmo evolutivo con una estrategia de búsqueda guiada (Tabú). Su investigación obtuvo muy buenos resultados con grandes instancias de grafos para este problema.

En la cruza, utiliza una estrategia *greedy* para la construcción de cada partición parcial V_i. Este tipo de cruza es llamada "*Greedy Partition Crossover (GPX)*", la cual realiza la elección de acuerdo a un criterio de optimización en particular.

Lo que concierne a la parte de mutación del algoritmo evolutivo, Galinier menciona que en su artículo propone cambiar la mutación por una búsqueda guiada, Utilizando la búsqueda local (tabú), esto da oportunidad a generar movimientos hacia óptimos locales, y de ellos partir hacia nuevas generaciones en la que una vez que se tiene la cruza del nuevo individuo, esa nueva configuración es utilizada para moverse dentro del espacio de soluciones (moviendo los nodos en diferentes clases de colores). Lo anterior, mediante las restricciones de la búsqueda tabú, las cuales no permiten regresar a estados pasados, generando diversidad en la población, así como obtener las mejores configuraciones cada generación.

En este trabajo se propone paralelizar en la plataforma CUDA variantes del algoritmo implementado por Galinier, por ello se concluyó que debido a que la memoria disponible dentro de la tarjeta CUDA es limitada, el implementar la técnica tabú no es una opción viable debido al consumo excesivo que esta implica. Por lo tanto se propone el utilizar dos técnicas distintas para implementar la búsqueda estocástica y caracterizar el impacto que puede llegar a tener el tipo de búsqueda dentro del algoritmo evolutivo. Las técnicas propuestas son: *ascenso de colina* y *metrópolis*.

La parte que se desea paralelizar dentro del algoritmo genético híbrido es la búsqueda local, ya que en el artículo de Galinier realiza la búsqueda de cada uno de los individuos dentro de un número fijo de generaciones, causando un retardo considerable en el tiempo. Mientras que en este trabajo se realiza la búsqueda local de cada individuo de forma paralela (asíncrona) mediante el uso de los hilos CUDA, cada hilo toma un individuo realizando su búsqueda independiente. Para evitar que el algoritmo avance, se utilizó una barrera CUDA, la cual garantiza que el algoritmo continúe una vez que todos los hilos hayan terminado de ejecutarse y avanzar a la siguiente generación. La población resultante será compuesta por óptimos locales que conformarán la nueva generación de individuos. Lo que respecta a la condición de parada, se utilizan dos aspectos: un número fijo de generaciones o hasta que tres generaciones den los mismos resultados. A continuación se puede observar en la Figura 4.1 la metodología propuesta del algoritmo genético híbrido paralelo, para resolver el problema de coloreado de grafos.

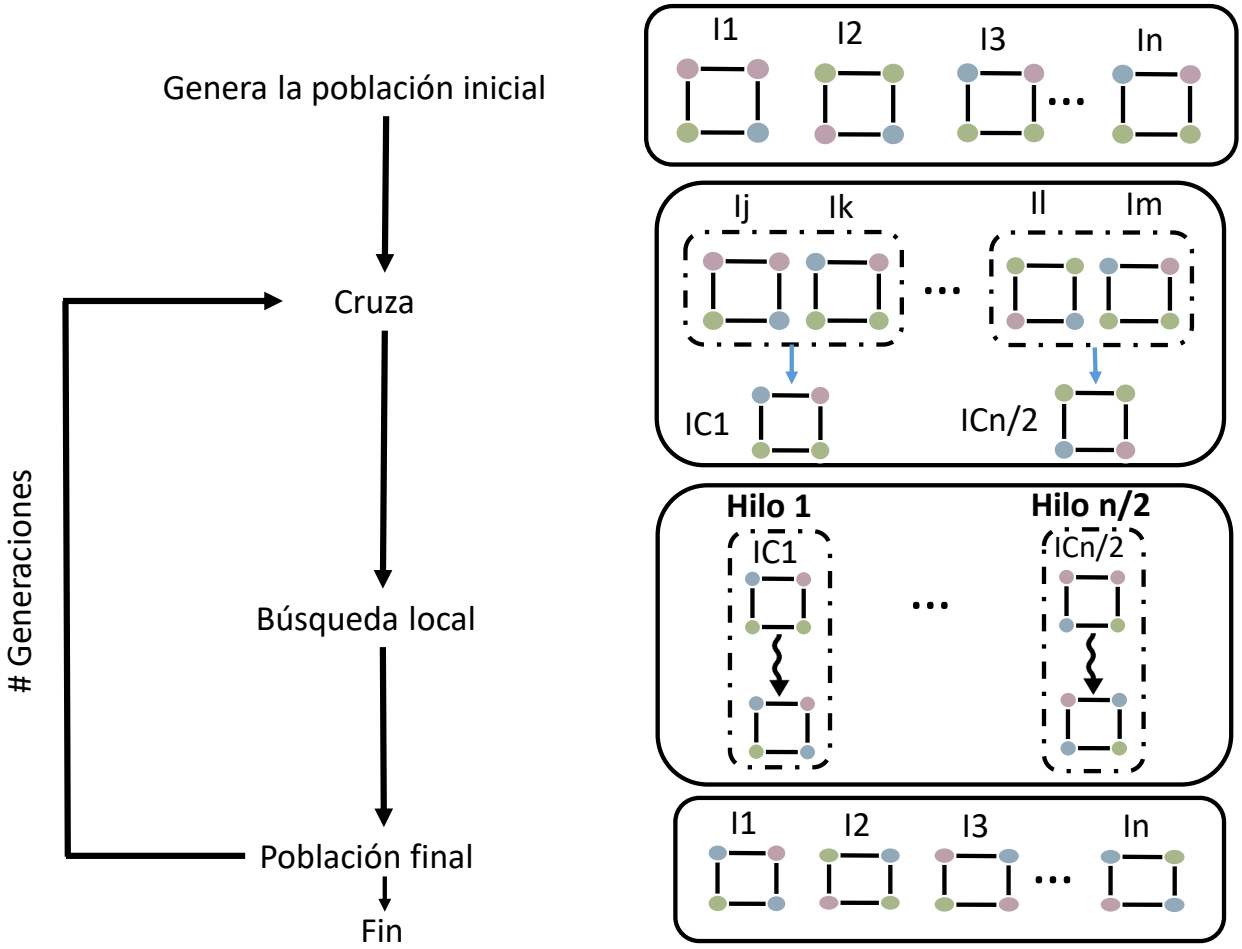

Figura 4.1: Propuesta paralela para solucionar el problema de Coloreado de Grafos.

Además de lo anterior, se propone analizar las distintas jerarquías de memoria de la plataforma CUDA (global,compartida y local) en conjunto con variaciones en el tamaño de la población, número de bloques, e hilos, para ambas estrategias de búsqueda local, mediante experimentos exhaustivos para caracterizar el desempeño de cada algoritmo en términos del número cromático, y el tiempo total de ejecución. Para esto se utilizarán tanto experimentos estandarizados (*benchmarks*) como instancias generadas de manera aleatoria.

4.3. Algoritmo evolutivo híbrido

Como hemos estado mencionando, este trabajo se basa en el algoritmo híbrido propuesto por Galinier, con pequeñas modificaciones, además de la aportación de la paralelización dentro de las búsquedas locales, las cuales permiten llevar a las pruebas a un nivel comparativo respecto a las configuraciones CUDA. Pero antes de llegar a las configuraciones de CUDA, el primer paso para generar

el algoritmo genético es la *función objetivo*, la cual juega un papel importante dentro del mismo. La utilizada por Galinier y también utilizada en este trabajo es la siguiente: dado un grafo G=(V,E) y un número de colores (k) considera todas las posibles particiones de V en k clases, incluyendo aquellas que no son un coloreado válido. A cada partición se le asigna una penalización igual al número de aristas que se encuentren dentro de la misma clase, por lo que si una partición tiene una penalización de cero se dice que es un coloreado válido (una solución), es decir, se van contando el número de aristas monocromáticas de cada uno de los nodos del grafo que se encuentran dentro del conjunto de particiones (el grafo coloreado). Muchas veces no es posible obtener un coloreado válido y se busca obtener particiones con una penalización mínima, que puede ser representada por los siguientes puntos:

- Una configuración $c \in C$ es cualquier partición $c = \{V_1, ..., V_k\}$ de V en k subconjuntos.
- $\forall c \in C, f(c) = |\{e \in E : \text{las puntas de } e \text{ se encuentran en la misma } V_i \in c\}|$

Al utilizar esta técnica de penalización como la función objetivo dentro del algoritmo evolutivo, da lugar a que al minimizar la función, se obtiene el grafo coloreado con el menor número de aristas monocromáticas. Para este trabajo de tesis se realizó una propuesta de implementación secuencial basada en el artículo de Galinier, así como una propuesta de algoritmo paralelo con cada configuración CUDA, cada una de las versiones del algoritmo fue realizado con metrópolis y ascenso de colina de colina. Una vez obtenido los resultados fue posible realizar un análisis comparativo en función del tiempo y el número de aristas monocromáticas de cada uno de ellos. A continuación se describen las propuestas de los algoritmos híbridos secuencial y paralelo:

Algoritmo 7: Coloreado Híbrido secuencial

Data: Grafo G=(V,E),número de colores k, número de configuraciones p
Result: La mejor configuración encontrada dada k

```
1  P=Inicializa_población (G,k,p);
2  while número fijo de generaciones do
3      while 1 ≤ p/2 do
4          (c1, c2)=Escoge_padres(P);
5          C=GPX(c1, c2);
6      end
7      while 1 ≤ p/2 do
8          P=Búsqueda_Local(G,C);
9          Actualiza_Población(P);
10     end
11 end
```

Algoritmo 8: Coloreado Híbrido paralelo

```
Data: Grafo G=(V,E),número de colores k, número de configuraciones p
Result: La mejor configuración encontrada dada k
P=Inicializa_población (G,k,p);
while número fijo de generaciones do
    while 1 ≤ p/2 do
        (c1, c2)=Escoge_padres(P);
        C=GPX(c1, c2);
    end
    P=Búsqueda_Local<<< bloques, hilos >>>(G,C);
    Pone barrera CUDA;
    while 1 ≤ p/2 do
        Actualiza_Población(P);
    end
end
```

Los datos de entrada de ambos algoritmos son el grafo, el número de colores y el tamaño de la población. Tal como en el algoritmo original utilizado por Galinier, el primer paso es inicializar la población, implementada mediante una estrategia *greedy* llamada *algoritmo de saturación*, la cual se basa en el artículo de Brélaz (1979) [51] (ver detalles en Sección 4.3.2). Una vez hecho esto, se pasa a realizar la búsqueda local sobre la población inicial, Galinier menciona la importancia de esto, ya que se desea que el conjunto que conforma a los individuos iniciales contenga diversidad entre ellos, ya que "una población homogénea no se desarrolla de manera eficiente", es decir se desea que la distribución dentro espacio de soluciones sea dispersa. Además ayuda a la cruza, ya que la nueva generación estará compuesta de posibles soluciones potenciales.

Una vez que se cuenta con las configuraciones iniciales sigue la *cruza*, para esta se utiliza otra estrategia *greedy* llamada *greedy partition crossover*, la cual forma un hijo a partir de dos padres, equilibrando la información que se pasa de cada padre hacia el hijo. El siguiente paso en el algoritmo, es realizar nuevamente una búsqueda local, es aquí donde se marca la diferencia entre el algoritmo secuencial y el paralelo. En el algoritmo secuencial se realiza la búsqueda local para cada individuo dentro de un ciclo, y una vez terminada la búsqueda se pasa a reemplazar a los nuevos hijos con el peor de los padres (veáse algoritmo 7).

Mientras que el paralelo, realiza una sola vez para el total de hijos (las búsquedas se realizan de forma simultánea) utilizando cada una de las jerarquías de memoria CUDA (global, compartida, local) y variaciones en el número de bloques, hilos y tamaño de la población, agregando una barrera $CUDA$ que no permite avanzar hasta que cada uno de los hilos (individuos) haya terminado su búsqueda, por último, se actualiza la población reemplazando con la peor configuración de los padres, asegurando que el total de la población nuevamente sea de tamaño p (veáse algoritmo 8). Lo anterior por un número fijo de generaciones, y el resultado final de los algoritmos en base a su función objetivo, es la configuración con el menor número de aristas monocromáticas encontrada.

4.3.1. Configuración de un individuo

En los algoritmos evolutivos existen tres aspectos importantes, la primera y una de las más importantes es la configuración del individuo, ya que refleja y brinda información del problema. Sin una representación apropiada, puede dar resultados no satisfactorios o carentes de sentido. Una de las recomendaciones es que la codificación debe ser capaz de representar todas las posibles soluciones, ajustarse a los operadores, y ser sencillas. Galinier [9] resalta en su artículo que para el problema de coloreado de grafos, existe una tendencia a definir al individuo mediante un *enfoque de asignación*, repartiendo colores a los vértices, pero debido a que este enfoque de asignación se convierte en una tupleta (color,vértice), esta representación no brinda información necesaria para resolver el problema de coloreado de grafos, ya que cada vértice se le asigna un color, pero no se tiene claro la relación entre los colores.

Por lo cual, para el problema de coloreado de grafos, es más significativo el enfoque de *partición* propuesto por Galinier, que clasifica a los conjuntos de vértices o nodos que pertenezcan a la misma clase o color, es decir, que la configuración del individuo se parte en clases de colores. Al mismo tiempo, gracias a que la configuración es fácil, permite diferentes formas de cruzar al individuo.

Para utilizar el enfoque de partición, lo primero es construir una configuración parcial de tamaño máximo de las subclases de colores de ambos padres, obteniendo una configuración completa. Es decir, que dado dos padres (configuración coloreada) $c_1 = \{V_1^1, ..., V_k^1\}$ y $c_2 = \{V_1^2, ..., V_k^2\}$ se forma una nueva configuración o individuo $c = \{V_1, ..., V_k\}$ de acuerdo a las siguientes características:

- Cada subconjunto V_i es incluido en una clase de uno de los dos padres: $\forall i(1 \leq i \leq k)\exists j : V_i \subseteq V_j^1$ o $\exists j : Vi \subseteq V_j^2$, por lo tanto V_i es con conjunto independiente.
- La unión de V_i tiene un tamaño máximo: $|\cup_{1 \leq i \leq k} V_i|$.
- La mitad de V_i es incluida en una clase del padre 1 y la otra mitad en la clase del padre 2, ya que se desea equilibrar la influencia de ambos padres.

La ventaja de este método, es que puede ser utilizado mediante alguna estrategia *greedy*, tomando a ambos padres sucesivamente y creando clases V_i. El algoritmo utilizado en el artículo de Galinier y en este trabajo es llamado *"Greedy partition crossover (GPX)"*. Para determinar si este enfoque era pertinente, con grafos aleatorios y una k fija obtuvieron la frecuencia con la que los nodos adyacentes quedan en la misma clase o color, y los resultados demostraron que con rareza estos nodos quedan en la misma clase, apoyando la idea de utilizar este enfoque. Una particularidad de estas estrategias es que permiten elegir el criterio de optimización más adecuado al problema. En este trabajo el criterio utilizado fueron dos aspectos: *grado del nodo y clase*. Para implementar el enfoque dentro del trabajo, fue necesario definir las estructuras de datos adecuadas para su codificación, debido a que el lenguaje que se utilizó fue *python*, se optó por utilizar un diccionario por diferentes razones:

- Funcionan como una lista enlazada.
- Contienen una llave y un valor.
- La búsqueda es diferente a la de una lista ya que utilizan un algoritmo hash, realizándolo en menos tiempo.

El diccionario contiene una estructura de dos datos: una *llave* y los *valores*, y la búsqueda de alguno de sus valores se ejecuta por medio de un algoritmo hash, realizando la búsqueda casi instantánea. Para este trabajo, los colores fueron representados por la llave, asignando una etiqueta de número a cada color, mientras que para el subconjunto de nodos pertenecientes a ese color, fue representado por los valores de dicha llave, es decir, que cada llave es capaz de guardar una lista enlazada que contiene todos los nodos que pertenecen a ese color.

Esto permite la mejor representación del individuo para el problema de coloreado de grafos con el lenguaje utilizado. Además de la representación del individuo, existen dos datos importantes que están ligados a el, como lo es: el *número de aristas monocromáticas*, la cual como su nombre lo indica, guarda el total de número de aristas monocromáticas de una configuración (grafo coloreado), y un *vector de probabilidades*, el cual guarda la probabilidad de acuerdo al número de nodos que contenga cada bolsa de color, ambos datos serán utilizados a lo largo del algoritmo genético híbrido, en particular en la búsqueda local.

4.3.2. Inicialización de la población

Siguiendo la metodología utilizada en al artículo de Galinier, se crea la población inicial utilizando el enfoque de partición (configuración) y se ajusta al lenguaje python mediante una estructura "diccionario"sobre cada individuo (mencionada en 4.3.1). El primer paso es utilizar una estrategia *greedy*, Galinier propone el utilizar el algoritmo "Greedy saturation"de Brélaz [51] con algunas adaptaciones para que produzca particiones de k clases.

La metodología en este trabajo de tesis fue la siguiente, como paso inicial y una de las contribuciones de este trabajo hasta este punto, fue que el primer individuo fuera tomado de una estrategia *greedy* para coloreado de grafos, llamada *smallest last* proporcionada por la biblioteca *networkx* de python (la cual trabaja con grafos), el principal problema de esta estrategia es que va agregando clase de colores a medida que crea necesitarlas, lo que significa que la mayoría de las veces da como resultado un coloreado con una k (número de colores) mayor a la ingresada en el algoritmo genético híbrido.

Para resolver dicho problema todos los nodos cuyo índice fuera mayor que k, son ingresados aleatoriamente dentro de cualquiera de las bolsas existentes en el algoritmo genético híbrido, para ello, simplemente se van tomando uno a uno los nodos y se pregunta si la case de color a la cual pertenece es menor o igual al número de colores ingresado en el algoritmo genético híbrido, si es así, se ingresa en la clase de color de la estructura "diccionario" correspondiente, si no es así, es asignada aleatoriamente a una clase de color menor o igual que k. El realizar este paso garantiza que cuando son instancias pequeñas, el primer individuo la mayoría de las veces sea una solución potencial, así como agregar diversidad en la población y utilizar distintas estrategias en la inicialización de la población.

Una vez obtenido el primer individuo, el siguiente paso fue continuar con la metodología del artículo de Galinier utilizando la estrategia "Greedy saturation", esta comienza con los subconjuntos de clases de colores vacías $V_1 = ...V_k = \emptyset$, y en cada paso se selecciona un nodo v del conjunto de nodos V, tal que v sea el que contenga el menor número de clases permitidas, es decir, aquellas clases que no contenga nodos adyacentes a v. Dicho nodo v es agregado en la clase de color que contenga índice i más pequeño, es decir la primera V_i que encuentre. Para implementar esta estrategia utilizando python fue necesario

una estructura de datos llamada "*Heap*", la cual genera un vector ordenado, y cuya complejidad para mantener el *max heap* o *min heap* es de $\ominus$(lg n), y cuya complejidad de extracción es igual a $\ominus$(1). La propiedad de *max heap* o *min heap* es utilizada para mantener los conjuntos del *heap* ordenados, el objetivo es que su padre siempre es mayor que sus hijos (cuando hablamos de maximizar), y este puede ser ordenado respecto algún criterio en particular.

A diferencia del artículo de Galinier el cual utilizó el número de clases permitidas como único criterio de ordenamiento, en este trabajo se realizó en base a dos criterios: el *número de clases permitidas*, y el *grado del nodo*, el objetivo de utilizar ambos criterios, es que el nodo que se encuentra al tope del *heap*, siempre sea aquel que tenga el menor número de clases y el mayor grado, generando que el primer nodo en salir será aquel que causa el mayor conflicto.

Lo primero que se realizó fue que al crear el *heap*, a cada nodo se le asignó un valor inicial de número de clases permitidas, como sabemos al principio los nodos no se encuentran dentro de ninguna clase de color, es decir, que inicialmente cualquier nodo tiene todas las clases de colores disponibles, por lo tanto, el valor inicial de todos los nodos es igual al número de colores (k).

Como mencionamos anteriormente, queremos que los nodos se ordenen con el menor número de clases y el mayor grado, para lograr esto, fue indispensable utilizar la propiedad de *max heap*, pero como queremos que el primer criterio sea el menor, para ello el valor inicial (k) de cada nodo se multiplico por -1, asegurando que tome el menor, mientras que el grado se ordena el mayor. Desafortunadamente este proceso no garantiza que todos los nodos queden dentro de una clase, por lo que hay ocasiones en las que puede haber nodos sin asignar a un color, esto debido a que dichos nodos entran en conflicto en todas las clases de colores (nodos con número de clases permitidas igual a cero), por lo que los nodos que sobran se asignan a una clase de manera aleatoria.

En el algoritmo 9 explica a detalle los pasos que fueron usados para crear a la población. Lo primero fue crear la estructura *heap* mediante las características mencionadas anteriormente para cada nodo, estos contienen cuatro características individuales: *número de clases permitidas*, *grado del nodo*, *id del nodo*, y un *conjunto de colores* (al principio se encuentra vacío), este último se utiliza con el objetivo de optimizar cuando pregunta si la clase de color se encuentra dentro del conjunto de v (paso seis del algoritmo), es decir, que algún nodo adyacente a el, ya se encuentre en esa clase de color. Debido a que *python* utiliza los conjuntos mediante tablas hash la complejidad de la búsqueda es $\ominus$(1), realizando la búsqueda de manera eficiente.

Como se explica anteriormente fue necesario utilizar un *heap híbrido*, el cual ordena de acuerdo a dos criterios: el menor número de clases permitidas y el mayor grado del nodo v. Una vez que el *heap* se encuentra ordenado, se pasa a inicializar las clases de colores vacías (al principio no contienen ningún nodo), el siguiente paso clave es el ciclo que se encarga de tomar el nodo del tope del *heap* hasta que este quede vacío, para asignar el nodo a una clase/bolsa de color se toma el nodo que se encuentra al tope del *heap* y se recorren uno a uno los colores, en cada color se pregunta si el color i se encuentra en el conjunto de clases hasta encontrar un color válido, al encontrarla asigna el nodo (es ahí donde rompe el ciclo), pero si i es mayor al número de colores, indica que el algoritmo no encontró una clase cuyo nodo o sus adyacentes no hayan sido asignados anteriormente, este es el caso que se mencionaba anteriormente en el cual el algoritmo no garantiza la asignación de todos los nodos dentro

de alguna clase de color, por lo que realiza la asignación aleatoria dentro de alguna de las clases de colores, el último paso es disminuir el número de las clases del nodo, como sabemos al inicio este dato fue multiplicado por -1 por lo que se disminuye sumando 1, además de lo anterior es indispensable actualizar el conjunto de clases de cada uno de los nodos adyacentes al nodo v, es decir que a todos los nodos adyacentes en su conjunto de clases se les asigna el color i, agregando esa información en dichos nodos, garantizamos que ya no puedan ser asignados a ciertas clases de colores (en las que tienen nodos adyacentes).

Algoritmo 9: Crea Individuo

```
Data: Grafo G=(V,E), número de colores
Result: Un individuo en particiones de número de colores
Heap = Inicializa_Heap(G, numero_colores);
Inicializa las bolsas a vacías;
while el Heap esté vacío do
    toma el elemento tope del heap;
    while i ≤ numero_colores do
        if i se encuentra en el conjunto_clases then
            i=i+1;
        else
            break;
        end
    end
    if i ≤ numero_colores then
        Ingresa v en bolsa_i;
    else
        Ingresa v en bolsa aleatoria;
    end
    Disminuye_Clases(G,Heap,v,i);
    i=0;
end
Retorna individuo;
```

Existe una contribución importante dentro del algoritmo, debido a que será utilizado a lo largo del algoritmo genético híbrido, en el cual se calcula la probabilidad de cada una de sus bolsas de colores, y se agrega al vector de probabilidades relacionado a esa nueva configuración, por medio de la fórmula: *números de nodos contenidos en la bolsa/Total de nodos del grafo*, dando como resultado que dicho vector contenga una probabilidad de cada bolsa equivalente al número de nodos que contenga, y que la suma de las probabilidades de todas bolsas sea igual a 1 (por cada individuo).

Galinier menciona que una de las diferencias fundamentales de éste algoritmo genético, es que el operador de mutación es reemplazado por una búsqueda local, además menciona la importancia de la diversidad de los individuos de la población. Como sabemos el algoritmo no garantiza el obtener

individuos dispersos que representen el espacio total de la búsqueda, por lo que Galinier recomienda implementar una búsqueda local en seguida de la creación de los individuos. Otro investigador que soporta esta teoría es Glover y en su líbro [47] menciona que las opciones más comunes son inicializar la población de forma aleatoria, o crear soluciones en base a una construcción heurística *greedy*, además menciona dos ventajas importantes de realizarlo con técnicas *greedy* sobre inicialización aleatoria, debido que al combinar las técnicas *greedy* con búsquedas locales generan mucho mejores soluciones, por lo que en promedio toma menos pasos de mejora, y por lo tanto, requiere menos tiempo del CPU, por lo cual en te trabajo se realizó una búsqueda local en la población inicial después de ser creados (Véase búsqueda local 2.7.2.3).

4.4. Operador de cruza

Para cruzar a los individuos Galinier [9] propone en su artículo el construir una configuración parcial del individuo, de tamaño máximo de las subclases de colores tomado de dos padres, para después ajustarla y obtener una configuración completa. Para ser más precisos, que mediante dos padres $c_1 = \{V_1, ..., V_k\}$ de conjuntos disjuntos de vértices, cumplan con las siguientes propiedades:

- Cada subconjunto V_i es incluido en una clase de uno de los dos padres: $\forall i(1 \leq i \leq k)\exists j : V_i \subseteq V_j^1$ o $\exists j : Vi \subseteq V_j^2$, por lo que V_i es con conjunto independiente.
- La unión de V_i tiene un tamaño máximo: $|\cup_{1\leq i\leq k} V_i|$ es máximo.
- Cerca de la mitad de V_i es incluida en una clase del padre 1 y la otra mitad en la clase del padre 2, ya que se desea equilibrar la influencia de ambos padres.

En los algoritmos genéticos el operador de cruza juega un papel muy importante, ya que este se encarga de pasar aspectos del individuo a futuras generaciones de acuerdo a algún criterio. Tomando en cuenta que la búsqueda local realizada después de la creación de los individuos, permite que la población inicial se componga de óptimos locales, y la probabilidad de que se encuentren dispersos dentro del espacio de búsqueda sea mayor, por lo que el trabajar con posibles soluciones potenciales inicialmente, ayuda a que la cruza pase sus características al nuevo individuo.

La técnica utilizada en el artículo de Galinier debido a los buenos resultados obtenidos y la utilizada en este trabajo de investigación fue: *greedy partition crossover* (GPX), su metodología consiste en que dos individuos (configuraciones) $c_1 = \{V_1^1, ..., V_k^1\}$ y $c_2 = \{V_1^2, ..., V_k^2\}$ los cuales son elegidos aleatoriamente de la población por medio de la función *Escoge_padres*, asegurando elegir dos individuos distintos (padres) y crear una nueva configuración $c = \{V_1, ..., V_k\}$ (hijo). Para lograrlo, en el algoritmo GPX (Algoritmo 10) observamos que el primer paso es mediante $1 \leq l \leq k$, el cual recorre uno a uno a los padres equitativamente, y permite obtener información de ambos.

Una vez que se elige el padre con el que se trabajará, comienza una búsqueda dentro de sus clases de colores, aquella que tenga la cardinalidad máxima será una de las clases del nuevo individuo, para ello se pasa dicho subconjunto V_l como alguna de las clases del nuevo individuo, es importante mencionar que no queremos que un nodo se encuentre clasificado en diferentes colores, para ello, es necesario que el subconjunto de nodos elegido sea eliminado de ambos padres, con el objetivo de que no sea elegido nuevamente. Si al terminar de recorrer las bolsas de colores (V_k) existen nodos sin ser asignados a

alguna clase de color, estos son ingresados aleatoriamente en las clases del nuevo individuo, debido a el porcentaje de aleatoriedad que existe dentro del algoritmo, genera incertidumbre en el movimiento de los puntos dentro del espacio de búsqueda, por lo cual, se requiere aplicar nuevamente búsquedas locales en los nuevos individuos, con el objetivo de tratar de mejorar las instancias si es posible (llevarlas nuevamente a su optimo local).

Algoritmo 10: GPX

Data: configuración $c_1 = v_1^1, ..., v_k^1$ y $c_2 = v_1^2, ..., v_k^2$
Result: configuración $c = v_1, ..., v_k$

```
for 1 ≤ l ≤ k do
    if l es impar then
        A=1;
        Escoge i tal que v_i^A tenga la máxima cardinalidad;
    else
        A=2;
        Escoge i tal que v_i^A tenga la máxima cardinalidad;
    end
    v_l = v_i^A ;
    Quita las aristas de V_l de c_1 y c_2;
end
Asigna vértices aleatorias a V-(v_1 ∪ ...v_k);
```

4.5. Búsqueda local

La literatura menciona que las búsquedas locales son técnicas que han sido implementadas muchas veces por sí solas, brindando muy buenos resultados incluso en problemas combinatorios, entre las más utilizadas se encuentra *asenso de colina*, este tipo de técnicas evalúa la función con el nuevo punto (candidato), tomando la solución siempre y cuando esta sea una mejora con respecto al valor de la función en el punto anterior, esta migración a través de mínimos locales continua hasta encontrar el mínimo global o hasta alcanzar el criterio de terminación.

Además del ascenso de colina, existe otra búsqueda local llamada *metrópolis*, esta además de siempre aceptar soluciones progresivas, no desecha las que no representan una mejora, si no que para ellas utiliza una cierta probabilidad de que sea aceptada o rechazada, en la mayoría de los casos en este tipo de algoritmos las peores soluciones tienen probabilidad baja de ser aceptadas. Un aspecto importante de metrópolis es la capacidad que brinda al tener la posibilidad de salir de óptimos locales, así como explorar otros puntos dentro del espacio de búsqueda que a partir de soluciones no tan buenas, abra camino a encontrar mejores. Debido a que en los problemas combinatorios no existe una solución única, si no un conjunto finito de soluciones válidas que satisfacen el criterio de búsqueda, para este trabajo de tesis se plantea utilizar ambos mecanismos de búsqueda y comparar cual de ellas pueda brindar mejores resultados para el problema de coloreado de grafos.

Como se menciona anteriormente, la función objetivo propuesta para el problema de coloreado de grafos requiere que la función mín$\{\{u, v\} : \{u, v\} \in E \mid f(u) = f(v)\}$, es decir, que minimice el número de aristas monocromáticas. Un coloreado válido para este problema sería que dado un número de colores (k) el número de aristas monocromáticas sea igual a cero, dado que con ese número de colores es posible colorear el grafo sin que existan nodos adyacentes con el mismo color (aristas monocromáticas), sin embargo sabemos que el tener la opción de poner un número fijo de colores, implica que no exista solución, por esas instancias es que la función busca minimizar el resultado obteniendo la mejor configuración encontrada dado ese número de colores k.

4.5.1. Detalles de implementación

4.5.2. Implementación de búsqueda local paralela

La principal aportación de este trabajo de tesis es la implementación de la búsqueda local de forma paralela, permitiendo que cada individuo realice su búsqueda asíncrona, eliminando la dependencia que existe entre los individuos de la población, para lograrlo a partir del lenguaje *python* en conjunto con la distribución gratuita *anaconda*, se propone utilizar el compilador *numba*, el cual maneja una máquina virtual de bajo nivel para compilar código *python* a código nativo, el cual puede ser ejecutado en el CPU o GPU en tiempo real. Esta cuenta con una versión para CUDA, la cual soporta programación en GPU CUDA, compilada por un subconjunto restringido de código *python* a kernels CUDA, así como las funciones del dispositivo y el modelo de ejecución CUDA [64]. Para utilizarla fue necesario agregar la biblioteca *numba* e importar CUDA para hacer uso de los GPUs.

Los autores de *numba* mencionan en su página [64] que la versión CUDA contiene un conjunto de restricciones de los tipos de datos que pueden ser utilizados dentro de ella, esto dejó limitaciones en la implementación que tuvieron que ser resueltas a lo largo del trabajo, para ello lo primero que se realizó fue un cambio de la estructura diccionario de *python* hacia un vector *numpy* (aceptado por *numba* CUDA), el cual funciona como un vector multidimensional o matriz. Las dimensiones del vector numpy fueron compuestas de tres, esto debido a que la primera dimensión contiene al *total de los individuos*, la segunda representa a las *clases de colores*, y la tercera el *total de nodos del grafo por cada bolsa de color* (ver Figura 4.2). En la última se realiza una representación binaria del nodo, de tal manera que los nodos que se encuentren dentro de la clase de color del individuo, el índice corresponde al *id del nodo* y contendrá un 1. En la figura 4.2 se observa un ejemplo de la nueva representación de los individuos para la instancia en la que el tamaño de la población sea de dos individuos, cada individuo contiene tres bolsas de colores y el total de nodos del grafo es igual a cuatro.

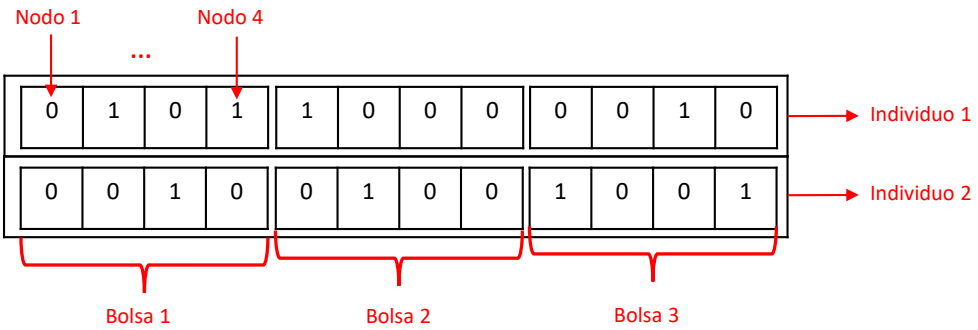

Figura 4.2: Nueva representación del individuo para la plataforma CUDA

Gracias a la nueva representación de los individuos, fue posible continuar con la implementación de los algoritmos de búsqueda local paralelos. Entre las limitaciones de numba se encuentra el uso de listas, diccionarios, o incluso la biblioteca de generador de números aleatorios utilizada por el algoritmo secuencial, por lo cual se optó por utilizar los vectores *numpy* así como una biblioteca especial para números aleatorios en CUDA llamada *cuda.random*.

Los cambios realizados respecto al algoritmo secuencial, fueron tres: el uso de generador de números aleatorios, elección del la clase y el nodo dentro de la clase, esto debido a que con la nueva configuración del individuo, no se tiene certeza de cuales son los nodos que se encuentran dentro de las clases de colores, y por último la forma en que realiza el cálculo de número de aristas monocromáticas de un nodo en una clase de color específica, la cual utiliza el nodo que se quiere saber si tiene alguna adyacencia dentro de la clase, la clase y la matriz de adyacencia, la metodología es recorrer todos los nodos V, preguntando si se encuentran en la clase, si es así entonces se verifica en la matriz de adyacencia simétrica (ya que es un grafo no dirigido) si dicho nodo tiene arista con el nodo inicial, si es así se suma el número de aristas monocromáticas.

En el algoritmo 11 muestra a detalle la metodología utilizada con asenso de colina para resolver el problema, los datos de entrada en el algoritmo son el grafo, el número de colores k, la nueva configuración del individuo, el total de número de aristas monocromáticas del individuo antes de la búsqueda, y por último el vector de probabilidades de las clases de colores de ese individuo.

El primer ciclo es el que elige una clase aleatoria de acuerdo a su vector de probabilidades (bolsa probabilística), es utilizada por medio de la nueva biblioteca de números aleatorios (cuda.random) tomando en cuenta que la clase no se encuentre vacía, una vez elegida la clase de color, el siguiente paso es tomar un nodo aleatorio, debido a que no sabemos que nodos se encuentran dentro de la clase, es necesario verificar que el nodo aleatorio se encuentre en ella, este paso es realizado hasta encontrar alguno de los nodos que se encuentren dentro de la clase elegida. El siguiente paso del algoritmo, es calcular el número de aristas monocromáticas del nodo elegido dentro de la clase probabilística, seguido de la elección de una nueva clase diferente a la anterior, y se procede a calcular nuevamente el número de aristas monocromáticas del nodo, pero esta vez en la nueva clase, lo cual genera el impacto que tiene el movimiento del nodo de la clase probabilística hacia la nueva clase.

Si el resultado de la nueva clase es menor, significa que es un progreso, realizando el movimiento en el vecindario y por último restando el resultado al número de aristas monocromáticas inicial del individuo, si no, se descarta la opción y vuelve a empezar. Esto se realiza por 1000 iteraciones o hasta que el número de aristas monocromáticas del individuo sea igual a cero, dando como resultado final el individuo con el menor número de aristas monocromáticas encontrado.

Algoritmo 11: Búsqueda ascenso de colina paralelo

```
Data: Grafo G=(V,E), número de colores, individuo,AristasMono,probabilidades
Result: configuración del individuo con menor número de aristas monocromáticas
1  while numero iteraciones do
2      while BolsaVacia = 0 do
3          bolsaProbabilistica = bolsaProbabilidad(probabilidades,numeroColores);
4          BolsaVacia();
5      end
6      while el nodo no se encuentre en la bolsa do
7          nodo = random(0,número total de nodos);
8          verifica si el nodo existe dentro de la bolsaProbabilistica;
9      end
10     monoAct = Calcula el número de aristas monocromáticas del nodo en la
        bolsaProbabilistica;
11     bolsaNueva = BolsaAleatoria(bolsaProbabilistica,numeroColores);
12     monoPost = Calcula el número de aristas monocromáticas del nodo en la bolsaNueva;
13     if monoPost ≠ 0 AND monoPost < monoAct then
14         Pasa el nodo a la bolsa nueva;
15     end
16     if Numero aristas monocromaticas = 0 then
17         Termina la búsqueda local;
18     end
19 end
```

Por otro lado, el algoritmo de *metrópolis* funciona de forma similar al algoritmo de ascenso de colina, con la excepción de que este les brinda una oportunidad a aquellas soluciones que no generan un progreso al momento de ser evaluado, esto se realiza a través de cualquier fórmula de distribución probabilista, entre las más comunes y la utilizada en este trabajo de tesis, es la fórmula de distribución de Boltzmann (Fórmula 4.1), la cual es utilizada para la simulación de distribución de partículas (átomos o moléculas) sobre estados de energía utilizando un equilibrio térmico. Algunas de las aplicaciones conocidas de ésta, es en la distribución de prueba en *deeplearning*.

$$P(\triangle E) = exp(- \triangle E/kbT) \tag{4.1}$$

Esta ecuación a menudo es utilizada para entender como evolucionan cantidades físicas, como la energía, la temperatura, el movimiento de fluidos y la conductividad eléctrica. Boltzmann demostró que

el número de moléculas y pequeños pedazos de energía tienden a infinito, donde P(E) es la probabilidad de la energía de E, y k = 1.38065×10^{-23} J/K llamada constante de Boltzmann, por lo tanto el radio entre dos estados de energía es dada por $e^{-\triangle E/kbT}$.

Es posible observar que el algoritmo metrópolis al principio, contiene los mismos pasos que los utilizados por el de ascenso de colina, así como el utilizar la biblioteca especializada para números aleatorios de CUDA. Cuando enviamos el individuo a realizar la búsqueda local, como es mencionado anteriormente, cada individuo contiene el cálculo de aristas monocromáticas (respecto a su coloreado) antes de entrar a la búsqueda local, esto es muy importante debido a que el evaluar continuamente la función es muy costoso.

Tal como fue realizado en la búsqueda de ascenso de colina, en la búsqueda local metrópolis se elige una clase probabilistica y un nodo contenido de dicha clase, procediendo al calculo del número de aristas monocromáticas que tiene ese nodo en esa clase, ya que el calcular la evaluación del nodo por clases, es mucho menos costosa que la evaluación completa del grafo.

Una vez obtenido el cálculo, se toma otra bolsa, y se evalúa el número de conflictos que agregaría a la nueva bolsa si el nodo estuviera en ella, a continuación se restan ambos valores (*monoPost-monoAct*), si el resultado es menor que cero, quiere decir que existe una mejora al mover el nodo, si es mayor que cero, se pasa a utilizar la fórmula de distribución (4.1), de tal manera que se genere una probabilidad en la cual el movimiento sea aceptado o rechazado. Una vez obtenido el resultado de la probabilidad, se genera un número aleatorio uniforme de 0-1, si el número es menor que la probabilidad, entonces acepta el cambio del nodo, si no, rechaza el movimiento. Los parámetros utilizados en la fórmula fueron T=2.7, k=1.38064852, ya que estos fueron los que brindaron un movimiento sutil en la elección de puntos del espacio de búsqueda. Todo esto se realiza durante 1000 iteraciones o hasta que el número de aristas monocromáticas del individuo sea cero.

Además de paralelizar ambas búsquedas locales, cada una se llevó a cabo mediante las diferentes jerarquías de memoria que permite CUDA, las cuales son: memoria global, memoria compartida y memoria local, asimismo cada jerarquía de memoria se realizó con diferentes bloques, e hilos. El objetivo de utilizar las distintas jerarquías así como el uso de diferentes bloques e hilos, las cuales llamamos configuraciones CUDA, permitió observar el comportamiento del algoritmo con cada una de estas características, por lo cual, fue posible obtener una configuración que brinda los mejores resultados de acuerdo al problema de coloreado de grafos y una vez obtenido dicho representante realizar la comparación de las distintas técnicas.

Respecto a la metodología utilizada para realizar el uso de las jerarquías de memoria, para todas las jerarquías se utilizó las llamadas a *kernel* CUDA, para la llamada en numba es necesario utilizar la palabra reservada *cuda.jit*, y en todas, el primer paso es mandar el individuo, matriz de adyacencia, vector de probabilidades, así como el número de aristas monocromáticas del individuo, desde el CPU (*host*) hacia el dispositivo CUDA (*device*), para ello se llama a la palabra reservada *cuda.to_device*, seguido de la definición del *kernel* (*Búsqueda_Local*$<<<$ *bloques*, *hilos* $>>>$*(G,C)*), en esta parte es donde se especifica el números de bloques e hilos que desea que el *kernel* utilice, en este trabajo se propuso utilizar dos tamaños dentro de la población 128 y 256, para cada tamaño se utilizaron variaciones de bloques e hilos, para una población de 128 se utilizaron 2 bloques con 32 hilos y 1 bloque

con 64 hilos, mientras que para la población de 256 se utilizaron 2 bloques con 64 hilos y 1 bloque con 128 hilos. Durante las ejecuciones siempre se puso el tamaño de la población de acuerdo al número de bloques e hilos utilizado, debido a que en los experimentos se realizaron pruebas para observar la eficiencia del mismo de acuerdo al movimiento de las configuraciones relacionadas con CUDA (véase pruebas en 5.1). La principal diferencia entre las jerarquías de memoria existe dentro del *kernel*, dicha diferencia es la declaración de las variables dentro del mismo, ya que con la memoria global basta con inicializar mediante las plabras reservadas *cuda.blockId.x*, *cuda.threadIdx.x*, *cuda.blockDim.x*, mientras que en la memoria compartida además de realizar lo anterior, es necesario convertir los parámetros del kernel a arreglos CUDA, mediante la palabra reservada *cuda.shared.array* y para memoria local es mediante el uso de *cuda.local.array*, por último, se ingresa una barrera CUDA por medio de la palabra reservada *cuda.syncthreads*(). Una vez terminado de ejecutar el *kernel*, es necesario recibir los resultados del dispositivo al CPU, esta parte pasa los resultados de los arreglos CUDA, hacia arreglos dentro del dispositivo del CPU, para esto se utiliza la palabra reservada *to_host*().

Algoritmo 12: Búsqueda metrópolis paralelo

```
Data: Grafo G=(V,E), número de colores, individuo,AristasMono,probabilidades
Result: configuración del individuo con menor número de aristas monocromáticas
1  while numero iteraciones do
2      while BolsaVacia = 0 do
3          bolsaProbabilistica = bolsaProbabilidad(probabilidades,numeroColores);
4          BolsaVacia();
5      end
6      while el nodo no se encuentre en la bolsa do
7          nodo = random(0,número total de nodos);
8          verifica si el nodo existe dentro de la bolsaProbabilistica;
9      end
10     monoAct = Calcula el número de aristas monocromáticas del nodo en la
        bolsaProbabilistica;
11     bolsaNueva = BolsaAleatoria(bolsaProbabilistica,numeroColores);
12     monoPost = Calcula el número de aristas monocromáticas del nodo en la bolsaNueva;
13     if ProbabilidadAceptar() = True then
14         Pasa el nodo a la bolsa nueva;
15     end
16     if Numero aristas monocromaticas = 0 then
17         Termina la búsqueda local;
18     end
19 end
```

4.6. Actualiza población

El último paso en el algoritmo genético es actualizar a la población, en esta existen tres opciones para ejecutar el reemplazo, decidir si se reemplazará siempre al individuo original, si será en cierto número (aleatorio) de veces, o si será cuando tiene mejor aptitud que el original. La utilizada en este trabajo es una variación de la primera, reemplazando el peor de los padres por el nuevo hijo, así se garantiza siempre tener el mismo número de individuos (tamaño de la población), además de asegurar que exista un movimiento constante en los puntos dentro del espacio de búsqueda.

Capítulo 5

Resultados experimentales

5.1. Descripción de los experimentos

Para la implementación de los experimentos se utilizó una computadora ASUS, con un procesador intel *core* i7 de 7_{ma} generación, la cual cuenta con una tarjeta gráfica $GEFORCE$ GTX de NVIDIA, esta tarjeta contiene 768 núcleos CUDA, indispensables para la implementación en paralelo del problema de coloreado de grafos. Por otro lado, el algoritmo genético híbrido se implementó en la plataforma pycharm, codificado con *python* 3.6 junto con la extensión para CUDA en *python* llamada *numba*, con ella fue posible realizar la paralelización de *python* bajo ciertas restricciones, como lo fueron las estructuras aceptadas por numba dentro del *kernel*. Esto generó un cambio en la implementación del algoritmo paralelo respecto al secuencial, los detalles de esta se describen el el capítulo 4.3. Lo que corresponde a las pruebas implementadas para el problema de coloreado de grafos, en este trabajo de tesis se realizó mediante el uso de dos tipos de experimentación: el primero fueron los *grafos* más conocidos para el problema de coloreado de grafos (*benchmarks*), y el segundo fueron *grafos aleatorios* generados con el modelo Gilbert G(n,p), en el que cada arista tiene una probabilidad independiente $0 < P < 1$ de aparecer en el grafo. Estos dos tipos de grafos fueron los utilizados en las pruebas bajo diferentes esquemas de configuraciones CUDA. El primer paso fue la investigación de los grafos *benchmarks*, en la cual gracias a los resultados obtenidos, se realizó una comparación de las distintos técnicas y algoritmos genéticos, así como las configuraciones CUDA. Mientras que las pruebas de los grafos aleatorios, se realizó en dos partes, la primera consistió en una comparación de las distintas configuraciones CUDA con ambas búsquedas locales, debido a los resultados obtenidos fue posible obtener un representante para cada técnica de búsqueda local. Una vez que se obtuvieron dichos representantes, la segunda parte consistió en ver el comportamiento del problema dentro de diferentes esquemas de resolución, realizando una comparación de los algoritmos greedy, el algoritmo genético híbrido secuencial y los representantes de las configuraciones CUDA, en base a los resultados generados en las pruebas con grafos aleatorios. Lo podemos ver a detalle en las secciones 5.1.1 y 5.1.2.

5.1.1. Experimentos con grafos benchamark

Como se menciona en la sección anterior, la primera parte de los experimentos fue con los grafos más conocidos para el problema de coloreado de grafos (*benckmarks*). A continuación en la Tabla 5.1

muestra los grafos utilizados, así como los datos correspondientes de cada uno de ellos como: el nombre del grafo, el número de nodos y el número cromático (mejor coloreado encontrado hasta la fecha).

Nombre	Num. Nodos	χ
myciel3	11	4
myciel4	23	5
$2 - Fullins\_3$	52	5
Jean	80	10
Anna	138	11
DSJC250.5	250	28
DSJC500.5	500	48
$le450\_15c$	450	15
$le450\_25c$	450	25
$flat300\_28\_300$	300	31

Tabla 5.1: Grafos utilizados en pruebas.

Los grafos anteriores, así como muchos otras instancias de grafos conocidos para el problema de coloreado de grafos se pueden encontrar en las siguientes páginas web:

- https://sites.google.com/site/graphcoloring/vertex-coloring.
- https://cse.unl.edu/ tnguyen/npbenchmarks/graphcoloring.html.
- https://turing.cs.hbg.psu.edu/txn131/graphcoloring.html.

También es importante destacar que algunos de los grafos mencionados en la tabla 5.1 fueron utilizados en el artículo de Galinier [9] para sus experimentos. Mientras que para este trabajo además de experimentar con los grafos del artículo, se utilizaron instancias de grafos diferentes. Galinier [9] menciona en su artículo que existen múltiples heurísticas propuestas para el problema de coloreado de grafos, entre ellas se encuentra los algoritmos *greedy*, y los algoritmos genéticos híbridos. Estas dos fueron caracterizadas mediante los grafos *benchmarks* para la parte experimental de este trabajo de tesis. Los algoritmos *greedy* que fueron utilizados, son los brindados por la biblioteca *NetworkX* de *python*, la cual cuenta con diferentes estrategias para resolver el coloreado de grafos, las elegidas fueron: *smallest last* e *independent set*, mientras que para el algoritmo híbrido secuencial la implementación del algoritmo de búsqueda local fue mediante el uso de dos metodologías: *ascenso de colina* y *metrópolis*. Además de utilizar estas dos heurísticas, una aportación importante en este trabajo fue agregar el comportamiento con el algoritmo híbrido paralelo, para este también se utilizó *ascenso de colina* y *metrópolis* para la búsqueda local, además de diferentes combinaciones de configuraciones CUDA, como fueron el *tamaño de la población*, *tipo de memoria*, y el *número de bloques* e *hilos*.

A continuación, en la Tabla 5.2 muestra los resultados de cada uno de los grafos mostrados en la Tabla 5.1, esta primera parte muestra aquellos que fueron resueltos con la estrategia *greedy* y *smallest last*, además de contener los resultados obtenidos al resolver el problema de coloreado de grafos en base a dos aspectos: *tiempo y número de aristas monocromáticas*:

	myciel3		myciel4		2-Fullins-3		Jean		Anna	
	Tiempo	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**
1	0	0	0.00098944	0	0.00199103	0	0.00882077	0	0.01357746	0
2	0.001080751	0	0.00204134	0	0.00503421	0	0.00698447	0	0.01296377	0
3	0.00103569	0	0.00076389	0	0.00390935	0	0.00603056	0	0.01305127	0
4	0	0	0.00204539	0	0.00532198	0	0.00868392	0	0.01491714	0
5	0.000997066	0	0.00103712	0	0.00199556	0	0.00292397	0	0.00589991	0
6	0.001035929	0	0.00103045	0	0.00586653	0	0.00797796	0	0.01206112	0
7	0.00118041	0	0.00098968	0	0.00196218	0	0.00906634	0	0.01495409	0
8	0.000914097	0	0.00204158	0	0.00598526	0	0.00837183	0	0.01498723	0
9	0.001001596	0	0.00216293	0	0.00499821	0	0.00832582	0	0.01008821	0
10	0.001031876	0	0.00103068	0	0.0020299	0	0.00802374	0	0.01456666	0
Promedio	0.000827742	0	0.00141325	0	0.00390942	0	0.00752094	0	0.01270669	0
Desviación estándar	0.0.000441375	0	0.000574067	0	0.001740139	0	0.001853076	0	0.002850843	0

	DSJC250.5		DSJC500.5		le450_15c		le450_25c		flat300_28_300	
	Tiempo	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**
1	0.18002892	5526	0.56450772	25477	0.19518161	9706	0.178092	3828	0.18968272	8147
2	0.13081598	6232	0.57553506	25681	0.1395359	9212	0.14919329	3369	0.20750785	8693
3	0.13644075	6141	0.58836222	27240	0.13700843	9881	0.15642786	3235	0.17689562	8356
4	0.13940096	6366	0.58314371	25261	0.15064168	8930	0.15404987	2641	0.19444466	7992
5	0.13326335	5990	0.55653024	24517	0.14017057	9235	0.18060517	3134	0.18354893	8255
6	0.12493348	5844	0.56497264	25905	0.15996933	9018	0.15149283	3574	0.18300486	8700
7	0.1547401	6234	0.53995395	25148	0.15297413	9131	0.14375377	3175	0.18957567	9472
8	0.13731337	6484	0.568223	25555	0.14015603	9153	0.1982615	3544	0.22163653	7931
9	0.14039087	5825	0.55367136	25933	0.14648628	9367	0.19048333	3353	0.17822242	8062
10	0.16453218	5786	0.67362976	26505	0.14463878	9459	0.15519142	3576	0.26199269	8334
Promedio	0.144186	6042.8	0.57685297	25722.2	0.15067627	9309.2	0.16575511	3342.9	0.19865119	8394.2
Desviación estándar	0.017072765	298.8659306	0.036847255	751.2241565	0.017186357	299.9784437	0.01925917	326.6709013	0.026168112	461.4821051

Tabla 5.2: Resultados del algoritmo Greedy smallest last

Igual que en la tabla anterior, en la siguiente Tabla (5.3) se observan los resultados del algoritmo *greedy* con los grafos *benchmarks* 5.1, pero está vez utilizando la estrategia *independent set*.

	myciel3		myciel4		2-Fullins-3		Jean		Anna	
	Tiempo	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**
1	0.001042604	0	0.00207376	0	0.02420115	0	0.03201699	6	0.08351636	6
2	0.001060963	0	0.00593352	0	0.02396965	0	0.03618264	0	0.09010482	1
3	0.000999928	0	0.003124	0	0.01196957	7	0.06904364	2	0.12993979	1
4	0	0	0.00274706	0	0.01067495	0	0.02993107	0	0.08819222	1
5	0.000902653	0	0.00240445	0	0.02458811	7	0.02945971	0	0.10948062	6
6	0.006980181	0	0.00992918	0	0.03403783	0	0.06814098	0	0.12305546	2
7	0.001147032	0	0.00199485	0	0.01400447	0	0.06122756	18	0.10574389	3
8	0.000916481	0	0.00202847	0	0.01110244	7	0.02806282	0	0.09796309	2
9	0.00099802	0	0.0020299	0	0.01376152	0	0.03003287	6	0.09116292	6
10	0.001134157	0	0.00299168	0	0.0098896	4	0.02895141	0	0.08353281	0
Promedio	0.001518202	0	0.00352569	0	0.01781993	2.5	0.04130497	3.2	0.1002692	2.8
Desviación estándar	0.001947615	0	0.00253953	0	0.008244111	3.341656276	0.017396426	5.750362307	0.016372589	2.347575582

	DSJC250.5		DSJC500.5		le450_15c		le450_25c		flat300_28	
	Tiempo	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**
1	0.71219349	3081	4.08032513	14528	1.65504456	7288	1.96129394	3056	1.00315809	5009
2	0.63602591	3316	4.21059871	13399	1.73037601	7039	2.0486989	3104	1.02753139	5273
3	0.68116379	3275	4.18206382	13769	1.70519519	7585	2.99763012	3232	2.16179967	3852
4	0.64661813	3167	4.14987826	13455	1.61309004	7254	1.92279816	3157	1.13168716	5135
5	0.665452	3393	4.28239417	12908	1.79854155	7342	2.04688311	3455	1.07471061	4877
6	0.6592083	3301	4.31834388	13926	1.70776725	7170	2.08314037	3541	1.13150764	4164
7	0.7134254	3160	4.43626428	13899	1.7456255	7202	1.96387196	3265	1.09823585	4761
8	0.6460228	3472	4.19519901	14735	1.64642072	7357	2.00889182	3374	1.06710649	4531
9	0.64968204	3063	4.23165846	14325	1.64952087	7435	1.95107675	3130	1.01384592	4883
10	0.73180985	3415	4.40513253	12394	1.70805693	7531	3.7826345	3456	2.15524006	4873
Promedio	0.67416017	3264.3	4.24918582	13733.8	1.69596386	7320.3	2.27669196	3277	1.28648229	4735.8
Desviación estándar	0.033770294	141.8191728	0.111950085	723.7996653	0.05547509	166.8625848	0.617358638	169.8947387	0.461756035	438.8935077

Tabla 5.3: Resultados del algoritmo Greedy independent set

Como es posible observar en las tablas 5.2 y 5.3, los algoritmos *greedy* presentan buenos resultados con respecto al tiempo, pero malos resultados en el total de número de aristas monocromáticas, esto quiere decir que colorea todo el grafo muy rápido, pero que a muchos nodos vecinos les asigna el mismo color. Por otro lado, también es posible observar que el algoritmo *greedy* con la estrategia *independent set* no da buenos resultados en el número de aristas monocromáticas con grafos pequeños, pero si con grafos grandes, pero si solo tomamos en cuenta el tiempo, es posible observar que el algoritmo *greedy* con estrategias *smallest last* es la que brinda los mejores resultados.

Para los resultados del experimento respecto a la columna de NAM (número de aristas monocromáticas), lo que se realizó fue que se contaron como aristas monocromáticas, todas aquellas que el algoritmo *greedy* les asignó una etiqueta (color), mayor al número de colores ingresado en el algoritmo genético híbrido. Ya que como sabemos el algoritmo *greedy*, utiliza las técnicas *smallest last* e *independent set* para resolver el problema de coloreado dado un grafo, una diferencia fundamental entre estas estrategias y el algoritmo genético, es que este último permite elegir el número de colores (k) con el cual se colorea todo el grafo, mientras que el algoritmo *greedy* da el resultado y el número de colores de a cuerdo a como fue construyendo la solución. Por lo anterior, la k elegida dentro del algoritmo genético para las pruebas con grafos *benchmarks* fue su número cromático (χ), es decir, el coloreado mínimo de acuerdo a la literatura, con el que ha sido posible colorear el grafo *benchmark*. Como sabemos para cada grafo es un número cromático distinto, para identificarlos es posible encontrarlos en las páginas web o libros que analizan los grafos para el problema de coloreado.

Como se menciona anteriormente, además de caracterizar los algoritmos *greedy* también se puso a prueba el desarrollo del algoritmo genético híbrido secuencial, en el cual debemos recordar que dentro del desarrollo de este, además fueron utilizadas algunas estrategias *greedy* basadas en el articulo de Galinier [9] en la inicialización de la población y la cruza, pero a diferencia del artículo en el cual utiliza una estrategia de búsqueda local tabú, es que en este trabajo se utilizaron dos estrategias de búsqueda diferentes: *ascenso de colina* y *metrópolis*. Además del uso de estas estrategias, también se realizó un movimiento de los parámetros de la población para las pruebas, en las cuales se realizó una variación de 128 y 256, esto con el fin de brindarle la oportunidad al algoritmo de tener una cantidad mayor de puntos dentro del espacio de soluciones, aumentando la probabilidad de encontrar alguna instancia que minimice el mayor número de aristas monocromáticas. Los resultados del algoritmo genético híbrido

secuencial con cada una de las estrategias y tamaños de la población, son los que se muestran en las siguientes tablas. En la Tabla 5.4 muestra los resultados con metrópolis y una población de 128, mientras que la Tabla 5.5 muestra los resultados con metrópolis y una población de 256:

	myciel3		**myciel4**		**2-Fullins-3**		**Jean**		**Anna**	
	Tiempo	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**
1	0.01199341	0	0.03589511	0	0.1236577	0	0.24137998	0	0.74102044	0
2	0.01196766	0	0.0359292	0	0.12864637	0	0.24234486	0	0.76000023	0
3	0.01298308	0	0.03690004	0	0.12864637	0	0.23739123	0	0.74201536	0
4	0.011971	0	0.04092908	0	0.12566352	0	0.2313807	0	0.75298977	0
5	0.01097059	0	0.03590345	0	0.12366915	0	0.23535776	0	0.7320435	0
6	0.0119679	0	0.03594136	0	0.12466645	0	0.23836231	0	0.74700212	0
7	0.01097035	0	0.03693175	0	0.12865233	0	0.24333048	0	0.75797558	0
8	0.01196766	0	0.03692675	0	0.12965226	0	0.22838902	0	0.74897718	0
9	0.01097035	0	0.03490543	0	0.12366939	0	0.23836112	0	0.73802757	0
10	0.01295424	0	0.03589773	0	0.14162087	0	0.24235225	0	0.72805238	0
Promedio	0.01187162	0	0.03661599	0	0.12785444	0	0.23786497	0	0.74481041	0
Desviación estándar	0.000737584	0	0.001642349	0	0.005397994	0	0.004958445	0	0.01056792	0

	DSJC250.5		**DSJC500.5**		**le450_15c**		**le450_25c**		**flat300_28**	
	Tiempo	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**
1	34.4339173	124	220.954092	371	74.6766167	431	72.3697755	86	50.3881962	138
2	33.5382807	128	210.217115	370	73.823859	426	71.7563999	85	51.2509372	141
3	33.3797312	129	210.934348	365	75.2016697	431	71.1939857	82	51.2569222	148
4	33.4814603	131	209.716462	369	74.1120951	439	71.4312761	82	51.0035985	145
5	34.1416974	128	210.321828	356	73.9667981	436	71.2816436	84	50.9148679	146
6	33.7896411	131	211.013944	372	73.5575523	450	72.178252	84	51.1412635	141
7	34.383019	122	211.182548	371	72.938226	442	71.6885824	85	50.7263722	146
8	35.5189779	131	209.721447	363	73.2145226	439	70.6603301	86	51.2589483	144
9	34.0399714	125	212.868005	364	73.4199181	431	71.9787707	85	51.675802	140
10	33.3697598	127	211.649294	375	73.8128757	440	70.6593909	83	50.9925952	134
Promedio	34.0076456	128	211.857908	368	73.8724133	437	71.5198407	84.2	51.0609503	142
Desviación estándar	0.66429813	3.134042473	3.333795444	5.581716184	0.674564817	6.98013053	0.586973473	1.475729575	0.348335461	4.295992965

Tabla 5.4: Resultados del AGH con metrópolis y población de 128

	myciel3		myciel4		2-Fullins-3		Jean		Anna	
	Tiempo	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**
1	0.02194118	0	0.07283092	0	0.25032377	0	0.47273564	0	1.48798871	0
2	0.02293754	0	0.07083535	0	0.25633907	0	0.46475625	0	1.52093339	0
3	0.02894998	0	0.07180858	0	0.25633287	0	0.47871947	0	1.52691579	0
4	0.02293777	0	0.07580853	0	0.26234794	0	0.4787457	0	1.52289486	0
5	0.02296662	0	0.07783961	0	0.26035094	0	0.47871399	0	1.52492094	0
6	0.02292514	0	0.07178903	0	0.26424408	0	0.49567437	0	1.52193594	0
7	0.02396894	0	0.0758214	0	0.26034164	0	0.48469782	0	1.52292371	0
8	0.02598262	0	0.07684398	0	0.26130319	0	0.47871923	0	1.51993537	0
9	0.02592969	0	0.07583761	0	0.25829911	0	0.47672486	0	1.52093315	0
10	0.02494669	0	0.07775307	0	0.28420877	0	0.48273587	0	1.51694322	0
Promedio	0.02434862	0	0.07471681	0	0.26140914	0	0.47922232	0	1.51863251	0
Desviación estándar	0.002122832	0	0.00264188	0	0.008913434	0	0.007969171	0	0.011104716	0

	DSJC250.5		DSJC500.5		le450_15c		le450_25c		flat300_28	
	Tiempo	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**
1	68.7292261	122	426.529261	358	144.684978	439	139.248773	79	97.8519464	141
2	68.71224	124	431.752662	369	145.426843	421	139.686529	83	98.0863228	137
3	68.7950163	122	420.425332	369	145.684744	432	139.201893	80	97.7582502	134
4	68.5028117	115	420.560595	365	145.029028	441	139.263215	82	97.8199213	135
5	68.9525898	120	419.488497	363	144.687906	420	139.389055	83	97.5864158	136
6	68.2015743	125	424.154968	368	144.622495	439	139.280027	84	97.3677421	139
7	68.3182971	129	419.596524	363	144.809958	415	139.154729	81	97.2739573	145
8	67.4735565	126	421.244795	368	144.37256	437	139.326871	82	97.523931	140
9	67.7887104	127	422.957178	366	144.185096	434	139.216652	84	97.477066	138
10	66.9030452	124	433.415206	364	144.34134	438	139.072661	80	97.4301987	141
Promedio	68.2377067	123	424.012502	365	144.784495	432	139.28404	81.8	97.6175751	139
Desviación estándar	0.661089216	3.949683532	5.035431528	3.465704995	0.477765284	9.406853294	0.166414901	1.751190072	0.253962014	3.306559138

Tabla 5.5: Resultados del AGH con metrópolis y población de 256

Los resultados obtenidos en las tablas 5.4 y 5.5 del algoritmo genético híbrido con metrópolis como búsqueda local, se observa notablemente una ventaja de la mitad del tiempo con una población de 128, a diferencia de la población de 256, pero lo que respecta a la característica de NAM los resultados muestran una disminución significativa en el número, es decir que con una población del doble si fue capaz de encontrar instancias en las cuales los individuos minimizan el NAM, pero en el doble de tiempo.

A continuación se observa las tablas que contienen los resultados del algoritmo genético híbrido secuencial utilizando la estrategia de ascenso de colina, con cada una de las variaciones en el tamaño de la población (128 y 256), una vez más tomando como referencia los resultados respecto al tiempo y número de aristas monocromáticas.

	myciel3		myciel4		2-Fullins-3		Jean		Anna	
	Tiempo	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**
1	0.01200557	0	0.03490591	0	0.12465429	0	0.23038363	0	0.74700809	0
2	0.01198578	0	0.03490639	0	0.12466645	0	0.23737168	0	0.75296474	0
3	0.01194739	0	0.03490686	0	0.13070512	0	0.23438287	0	0.75498247	0
4	0.01199889	0	0.03490615	0	0.1276567	0	0.24335241	0	0.75797296	0
5	0.01200032	0	0.03490591	0	0.1336422	0	0.24132395	0	0.75398302	0
6	0.01196718	0	0.03490639	0	0.1296792	0	0.23935962	0	0.75396466	0
7	0.01201415	0	0.03991747	0	0.13361645	0	0.23736453	0	0.76691628	0
8	0.01199841	0	0.03991866	0	0.13566875	0	0.24433279	0	0.7709384	0
9	0.01199889	0	0.03685117	0	0.13165998	0	0.24435163	0	0.75996852	0
10	0.01096845	0	0.03693223	0	0.12964344	0	0.26526451	0	0.76097083	0
Promedio	0.0118885	0	0.03630571	0	0.13015926	0	0.24174876	0	0.757967	0
Desviación estándar	0.000323876	0	0.002069323	0	0.003715174	0	0.009408658	0	0.007049779	0

	DSJC250.5		DSJC500.5		le450_15c		le450_25c		flat300_28	
	Tiempo	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**
1	34.0878696	108	207.860431	342	74.1909549	409	73.7062657	66	53.4609864	128
2	33.9991004	112	208.57447	352	73.9624765	406	73.5247037	65	53.6884115	123
3	33.913336	110	208.396972	350	74.1496665	420	74.1140952	65	53.0471277	125
4	33.4665	109	207.903316	352	73.2573633	406	73.4937546	68	53.5388188	123
5	33.4994431	109	207.958386	349	73.9698389	419	74.189955	63	53.0730648	122
6	33.7009046	107	207.775604	347	72.8205531	415	73.8505254	67	52.1275933	125
7	34.0220447	112	207.992037	337	73.4525609	418	73.6682682	66	52.9693365	126
8	33.8175294	111	207.383683	346	74.334475	409	73.8923864	67	53.4430747	129
9	33.8634367	101	206.581817	347	72.5423317	415	73.1087642	65	52.9413812	124
10	34.1397007	111	207.685867	345	73.6613183	404	73.9239511	65	53.5906794	123
Promedio	33.8509865	109	207.811258	347	73.6341539	412	73.747267	65.7	53.1880474	125
Desviación estándar	0.233277893	3.265986324	0.547936885	4.62000481	0.607034453	5.971227307	0.3202231	1.418136492	0.464443859	2.299758441

Tabla 5.6: Resultados del AGH con ascenso de colina y población de 128

	myciel3		myciel4		2-Fullins-3		Jean		Anna	
	Tiempo	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**
1	0.0239284	0	0.07083583	0	0.25828624	0	0.47176909	0	1.48605728	0
2	0.02396083	0	0.07282281	0	0.25432062	0	0.47373343	0	1.46508193	0
3	0.02293634	0	0.07084322	0	0.25832653	0	0.47971725	0	1.5079639	0
4	0.02493382	0	0.0757916	0	0.25928211	0	0.47970963	0	1.52293277	0
5	0.02389765	0	0.08079314	0	0.27525806	0	0.47472882	0	1.52691078	0
6	0.02293825	0	0.07183933	0	0.26029229	0	0.50365281	0	1.51195717	0
7	0.02293277	0	0.07480121	0	0.26227999	0	0.46874642	0	1.50697303	0
8	0.02493382	0	0.07577181	0	0.25726438	0	0.46974373	0	1.51294851	0
9	0.02492976	0	0.07081056	0	0.25829029	0	0.4707408	0	1.50697017	0
10	0.02494049	0	0.07385325	0	0.28124762	0	0.47669339	0	1.48300052	0
Promedio	0.02403321	0	0.07381628	0	0.26248481	0	0.47692354	0	1.50307961	0
Desviación estándar	0.000875391	0	0.003145117	0	0.008671316	0	0.01015588	0	0.019233026	0

	DSJC250.5		DSJC500.5		le450_15c		le450_25c		flat300_28	
	Tiempo	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**
1	70.5293796	106	441.117275	341	153.084399	407	146.447914	66	107.111513	123
2	70.9492526	105	439.359968	344	153.662491	411	145.717695	64	106.995825	124
3	71.848856	107	439.836769	345	152.642048	418	145.262793	64	108.305322	127
4	70.9263165	102	439.630596	345	151.28704	414	144.304349	66	107.334925	118
5	70.9941368	107	442.634522	350	152.627177	411	145.396436	65	107.636144	124
6	70.9602275	107	441.527525	347	152.854654	416	145.541073	65	108.483907	124
7	70.4884889	109	454.25481	348	158.835179	400	144.468942	64	107.496485	127
8	69.8621318	110	445.137683	341	152.745842	413	144.135834	65	106.813313	125
9	70.2869971	108	437.545121	346	153.235503	416	144.345296	64	107.549411	124
10	70.5044787	109	456.500491	337	151.255808	409	144.914815	62	107.402766	121
Promedio	70.7350266	107	443.754476	344	153.223014	412	145.053515	64.5	107.512961	124
Desviación estándar	0.533318094	2.201009869	6.480931135	3.83550663	2.11824682	5.275730597	0.749111629	1.178511302	0.532215852	2.668749187

Tabla 5.7: Resultados del AGH con ascenso de colina y población de 256

Como en el caso anterior (metrópolis), los resultados del algoritmo genético híbrido secuencial con ascenso de colina y poblaciones de 128 y 256 (tablas 5.6 y 5.7) muestran que el tiempo de la población de 256 con respecto a la de 128 individuos es muchas veces el doble o más, pero una vez más muestra una disminución en el número de aristas, reafirmando la conclusión anterior.

Una vez realizado las pruebas del algoritmo genético híbrido de forma secuencial con cada una de las estrategias de búsqueda local, fue posible continuar con las pruebas del algoritmo genético híbrido paralelo en la plataforma CUDA y la extención numba. Con el fin de obtener una investigación más completa, además de las configuraciones anteriores del algoritmo genético secuencial se agregó distintas configuraciones de memoria que permite CUDA, las cuales son: *memoria global, memoria compartida y memoria local*. En cada configuración de memoria se transfirió la *matriz de adyacencia del grafo, la instancia del individuo, el vector de probabilidades y el vector de número de aristas*. El objetivo de esto era ver la diferencia entre los tipos de memoria, así como el ver que tanto afectaba la condición de competencia por los recursos, dependiendo la distancia de esta. Para las pruebas realizadas en CUDA, además del aspecto de la memoria, también consideró el aspecto del número de hilos y el número de bloques que iban a ser utilizados en el *kernel* CUDA, cada una dependiendo del tamaño de la población (128 y 256), dichas configuraciones fueron las siguientes:

Población 128	Población 256
2 bloques con 32 hilos	2 bloques con 64 hilos
1 bloque con 64 hilos	1 bloque con 128 hilos

Tabla 5.8: configuraciones del AGH-CUDA con respecto al tamaño de la población.

El motivo de utilizar diferentes tamaños de número de bloques y número de hilos, fue que en la literatura menciona que la paralelización en CUDA tiende a dar mejores resultados si los hilos son múltiples de 32 en cada bloque, experimentar con esto permitió analizar los diferentes resultados y el comportamiento del algoritmo. Por otro lado, lo que respecta al número de colores utilizado en el algoritmo paralelo, fue el mismo utilizado con los anteriores (número cromático χ), dando como resultado, la mejor configuración que fue capaz de encontrar, con el mínimo número de aristas mono-

cromáticas. En la tabla muestra los resultados del algoritmo genético híbrido paralelo con cada una de las configuraciones anteriores, permitiendo hacer una comparación a simple vista. Los resultados que se muestran en las tablas son el promedio de 10 corridas del algoritmo con el mismo grafo y distintas semillas. A continuación se dividen dos tablas, la primera tabla (5.9) muestra los resultados del algoritmo genético híbrido paralelo con las diferentes configuraciones y tamaños de población con búsqueda local metrópolis, mientras que la segunda tabla (5.10) muestra lo mismo pero, con la búsqueda local ascenso de colina.

	myciel3		**myciel4**		**2-Fullin-3**		**Jean**		**Anna**	
	Tiempo	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**
Global,128,2B,32H	0.0124676	0	0.03668928	0	0.1294549	0	0.24205575	0	0.76824338	0
Global,128,1B,64H	0.01208477	0	0.03680036	0	0.13353307	0	0.24884129	0	0.75777636	0
Global,256,2B,64H	0.02383668	0	0.0747021	0	0.25820274	0	0.47802968	0	1.49112074	0
Global,256,1B,128H	0.02343116	0	0.06561277	0	0.2483758	0	0.5311223	0	1.44964225	0
Compartido,128,2B,32H	0.01245883	0	0.03904974	0	0.12496932	0	0.23276341	0	0.72482629	0
Compartido,128,1B,64H	0.01217918	0	0.03811331	0	0.13223476	0	0.24504087	0	0.76275992	0
Compartido,256,2B,64H	0.0241441	0	0.07400806	0	0.26549172	0	0.47712281	0	1.51675532	0
Compartido,256,1B,128H	0.02424293	0	0.07539182	0	0.25961161	0	0.47422667	0	1.44497166	0
Local,128,2B,32H	0.01562073	0	0.03428657	0	0.12965605	0	0.23118894	0	0.72170324	0
Local,128,1B,64H	0.01249373	0	0.03593302	0	0.12805967	0	0.23275688	0	0.72170436	0
Local,256,2B,64H	0.02342687	0	0.06873183	0	0.25306246	0	0.45769961	0	1.43717144	0
Local,256,1B,128H	0.02414665	0	0.07439873	0	0.27027705	0	0.47712581	0	1.5648118	0

	DSJC250.5		**DSJC500.5**		**le450_15c**		**le450_25c**		**flat300_28**	
	Tiempo	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**
Global,128,2B,32H	33.7784859	126.2	211.159676	366.7	71.5544534	434.9	72.8595934	82.3	51.8439337	141.2
Global,128,1B,64H	33.7664801	123.7	214.284681	365.6	70.9570144	437.9	71.6842514	82.8	52.8520553	140.2
Global,256,2B,64H	66.5663851	123.5	421.985079	363.5	145.053013	437.5	149.404148	82.5	101.116071	140
Global,256,1B,128H	65.5345762	123.7	411.801365	366	138.462901	433.1	139.13256	83.6	102.435103	138.6
Compartido,128,2B,32H	32.2797088	125.5	205.553325	367.1	69.78218	434.8	69.8541036	81.8	54.7158808	139.6
Compartido,128,1B,64H	33.6483535	125.1	213.496964	366.8	71.8928846	435.7	71.8097772	83.2	51.7925991	141
Compartido,256,2B,64H	66.5423517	122	430.630632	360.9	141.605158	432.6	142.871373	82.6	100.438336	138.9
Compartido,256,1B,128H	63.3256643	125.3	421.142926	362.3	142.19439	437	141.475995	80	100.910794	140.2
Local,128,2B,32H	32.3939242	127	206.529582	363.1	70.2711495	434.9	69.1058066	82.8	48.9696568	139.9
Local,128,1B,64H	32.3908168	127	207.107711	364.2	68.9168411	439.2	69.597785	82.9	48.9541056	141.6
Local,256,2B,64H	64.4675533	123.4	415.066547	362	142.391495	432.7	136.831269	80.7	99.8263281	138.3
Local,256,1B,128H	67.2900504	122.6	444.546514	361.6	144.741161	436.1	141.839243	82.2	99.9947874	139

Tabla 5.9: comparación de resultados del AGH-CUDA metrópolis con los distintos tipos de memoria, bloques e hilos.

La tabla 5.9 resalta en gris aquellas configuraciones que tuvieron los mejores resultados respecto al tiempo y al número de aristas monocromáticas por grafo *benchmark*. Es posible ver en los resultados que existe una tendencia hacia la memoria local, es decir que a grandes rasgos, la mayoría de grafos *benchmarks* utilizando memoria local son los que obtienen el menor tiempo, así como el menor número de aristas monocromáticas. Una vez que se analiza a detalle, se observa que en específico la memoria local con población de 128, 2 bloques y 32 hilos, es aquella que brindó los mejores resultados del tiempo. Estos reafirman la teoría de que gracias a que los datos del individuo se encuentran en la memoria local

exista una disminución en el tiempo para obtener los datos, así como una minimización en la condición de competencia.

	myciel3		**myciel4**		**2-Fullin-3**		**Jean**		**Anna**	
	Tiempo	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**
Global,128,2B,32H	0.01186512	0	0.03740296	0	0.13064239	0	0.24175212	0	0.75179014	0
Global,128,1B,64H	0.01216662	0	0.03750648	0	0.12895861	0	0.23986328	0	0.75629468	0
Global,256,2B,64H	0.02403724	0	0.07341001	0	0.2646903	0	0.48102677	0	1.50167754	0
Global,256,1B,128H	0.02499306	0	0.07186005	0	0.24993515	0	0.45770319	0	1.4683805	0
Compartido,128,2B,32H	0.01167412	0	0.03760276	0	0.13293967	0	0.24412897	0	0.75360692	0
Compartido,128,1B,64H	0.01166618	0	0.03751254	0	0.13184466	0	0.24494262	0	0.74899721	0
Compartido,256,2B,64H	0.02343147	0	0.07029381	0	0.25618696	0	0.45926509	0	1.44184403	0
Compartido,256,1B,128H	0.02343309	0	0.06873243	0	0.24993711	0	0.48649487	0	1.4972954	0
Local,128,2B,32H	0.01405582	0	0.03124785	0	0.12654028	0	0.22807615	0	0.7185735	0
Local,128,1B,64H	0.01249394	0	0.03593137	0	0.13434041	0	0.23275118	0	0.72014003	0
Local,256,2B,64H	0.02394104	0	0.07450573	0	0.25841138	0	0.47673178	0	1.51734428	0
Local,256,1B,128H	0.02343206	0	0.07030921	0	0.25619063	0	0.46083097	0	1.52664618	0

	DSJC250.5		**DSJC500.5**		**le450_15c**		**le450_25c**		**flat300_28**	
	Tiempo	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**
Global,128,2B,32H	34.1717166	126.5	213.940944	367.6	72.9817449	433.4	73.1647449	83.3	51.7450323	139.3
Global,128,1B,64H	34.3502736	126.5	214.641463	365.1	73.133162	437.3	70.9769254	83.7	51.8578098	140.4
Global,256,2B,64H	64.92534	123.9	413.151073	362.5	139.837441	437.2	139.158007	81.4	99.3718972	138.8
Global,256,1B,128H	64.9549603	124	411.359283	364.6	139.243994	431.9	138.284741	81.6	97.7830749	137.3
Compartido,128,2B,32H	34.288105	126	216.863896	363.2	74.5060439	438	72.5581412	82.9	56.5781646	138.3
Compartido,128,1B,64H	34.5847636	124.2	213.416294	365.5	73.0850838	436.6	72.7328963	83.4	52.5269308	140.1
Compartido,256,2B,64H	68.1067633	124	427.984928	365	144.212103	433	138.417273	82.6	98.2642722	138.1
Compartido,256,1B,128H	67.4484771	121.7	429.068003	361.4	139.036223	434.1	142.060453	82.1	100.116905	136.3
Local,128,2B,32H	32.9921419	125	207.696505	364.8	70.207198	434.6	70.5585664	82.7	50.3506656	142.5
Local,128,1B,64H	33.0202579	125.2	204.747229	365.8	70.6045974	436.5	72.2004636	82	49.8178983	141.3
Local,256,2B,64H	67.0653446	123.4	430.594769	362.3	145.049938	431.2	148.267676	82.8	101.732809	138.4
Local,256,1B,128H	67.0821071	122.8	412.174914	363.9	141.39629	431.6	137.739474	82	98.5904665	138

Tabla 5.10: comparación de resultados del AGH-CUDA ascenso de colina con los distintos tipos de memoria, bloques e hilos.

En la tabla 5.10 se analizan los resultados del algoritmo genético paralelo, pero esta vez con la búsqueda local ascenso de colina, nuevamente se puede observar como en la tabla 5.9, que la tendencia de la mejor configuración CUDA se encuentra en la memoria local, en concreto con una población de 128, 2 bloques y 32 hilos es la que brinda los mejores resultados. Si realizamos una comparación de las dos estrategias de búsqueda local, se observa una variación de los resultados, es decir, que al tratar de comparar los mecanismos de búsqueda e identificar cual de ellas es la mejor no resulta tan fácil verlo, esto debido a que muchas veces estas estrategias se comportan de manera estocástica y existe un grado de aleatoriedad, por lo que es posible observar una oscilación de buenos resultados con cada una de las estrategias locales dependiendo del grafo.

Como última instancia se reunieron todos los resultados anteriores con el objetivo de poder realizar una comparativa a mayor profundidad sobre ellos, como fueron los algoritmos *greedy*, el algoritmo genético híbrido secuencial, y los resultados del algoritmo genético híbrido paralelo con cada una de las diferentes configuraciones. A continuación se muestra la tabla 5.11 donde se puntualiza cada uno.

	myciel3		myciel4		2-Fullin-3		Jean		Anna	
	Tiempo	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**
smallest last	0.00082774	0	0.00141325	0	0.00390942	0	0.00752094	0	.01270669	0
Independent set	0.0015182	0	0.00352569	0	0.01781993	2.5	0.04130497	3.2	0.1002692	2.8
Secuencial-Metro-128	0.01187162	0	0.03661599	0	0.12785444	0	0.23786497	0	0.74481041	0
Secuencial-Metro-256	0.02434862	0	0.07471681	0	0.26140914	0	0.47922232	0	1.51863251	0
Secuencial-AC-128	0.0118885	0	0.03630571	0	0.13015926	0	0.24174876	0	0.757967	0
Secuencial-AC-256	0.02403321	0	0.07381628	0	0.26248481	0	0.47692354	0	1.50307961	0
Global,128,2,32 AC	0.01186512	0	0.03740296	0	0.13064239	0	0.24175212	0	0.75179014	0
Global,128,1,64 AC	0.01216662	0	0.03750648	0	0.12895861	0	0.23986328	0	0.75629468	0
Global,256,2,64 AC	0.02403724	0	0.07341001	0	0.2646903	0	0.48102677	0	1.50167754	0
Global,256,1,128 AC	0.02499306	0	0.07186005	0	0.24993515	0	0.45770319	0	1.4683805	0
Compartido,128,2,32 AC	0.01167412	0	0.03760276	0	0.13293967	0	0.24412897	0	0.75360692	0
Compartido,128,1,64 AC	0.01166618	0	0.03751254	0	0.13184466	0	0.24494262	0	0.74899721	0
Compartido,256,2,64 AC	0.02343147	0	0.07029381	0	0.25618696	0	0.45926509	0	1.44184403	0
Compartido,256,1,128 AC	0.02343309	0	0.06873243	0	0.24993711	0	0.48649487	0	1.4972954	0
Local,128,2,32 AC	0.01405582	0	0.03124785	0	0.12654028	0	0.22807615	0	0.7185735	0
Local,128,1,64 AC	0.01249394	0	0.03593137	0	0.13434041	0	0.23275118	0	0.72014003	0
Local,256,2,64 AC	0.02394104	0	0.07450573	0	0.25841138	0	0.47673178	0	1.51734428	0
Local,256,1,128 AC	0.02343206	0	0.07030921	0	0.25619063	0	0.46083097	0	1.52664618	0
Compartido,128,1,64 AC	0.01166618	0	0.03751254	0	0.13184466	0	0.24494262	0	0.74899721	0
Global,128,2,32 Metro	0.0124676	0	0.03668928	0	0.1294549	0	0.24205575	0	0.76824338	0
Global,128,1,64 Metro	0.01208477	0	0.03680036	0	0.13353307	0	0.24884129	0	0.75777636	0
Global,256,2,64 Metro	0.02383668	0	0.0747021	0	0.25820274	0	0.47802968	0	1.49112074	0
Global,256,1,128 Metro	0.02343116	0	0.06561277	0	0.2483758	0	0.5311223	0	1.44964225	0
Compartido,128,2,32 Metro	0.01245883	0	0.03904974	0	0.12496932	0	0.23276341	0	0.72482629	0
Compartido,128,1,64 Metro	0.01217918	0	0.03811331	0	0.13223476	0	0.24504087	0	0.76275992	0
Compartido,256,2,64 Metro	0.0241441	0	0.07400806	0	0.26549172	0	0.47712281	0	1.51675532	0
Compartido,256,1,128 Metro	0.02424293	0	0.07539182	0	0.25961161	0	0.47422667	0	1.44497166	0
Local,128,2,32 Metro	0.01562073	0	0.03428657	0	0.12965605	0	0.23118894	0	0.72170324	0
Local,128,1,64 Metro	0.01249373	0	0.03593302	0	0.12805967	0	0.23275688	0	0.72170436	0
Local,256,2,64 Metro	0.02342687	0	0.06873183	0	0.25306246	0	0.45769961	0	1.43717144	0
Local,256,1,128 Metro	0.02414665	0	0.07439873	0	0.27027705	0	0.47712581	0	1.5648118	0

	DSJC250.5		**DSJC500.5**		**le450_15c**		**le450_25c**		**flat300_28**	
	Tiempo	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**
smallest last	0.144186	6042.8	0.57685297	25722.2	0.15067627	9309.2	0.16575511	3342.9	0.19865119	8394.2
Independent set	0.67416017	3264.3	4.24918582	13733.8	1.69596386	7320.3	2.27669196	3277	1.28648229	4735.8
Secuencial-Metro-128	34.0076456	127.6	211.857908	367.6	73.8724133	436.5	71.5198407	84.2	51.0609503	142.3
Secuencial-Metro-256	68.2377067	123.4	424.012502	365.3	144.784495	431.6	139.28404	81.8	97.6175751	138.6
Secuencial-AC-128	33.8509865	109	207.811258	346.7	73.6341539	412.1	73.747267	65.7	53.1880474	124.8
Secuencial-AC-256	70.7350266	107	443.754476	344.4	153.223014	411.5	145.053515	64.5	107.512961	123.7
Global,128,2,32 AC	34.1717166	126.5	213.940944	367.6	72.9817449	433.4	73.1647449	83.3	51.7450323	139.3
Global,128,1,64 AC	34.3502736	126.5	214.641463	365.1	73.133162	437.3	70.9769254	83.7	51.8578098	140.4
Global,256,2,64 AC	64.92534	123.9	413.151073	362.5	139.837441	437.2	139.158007	81.4	99.3718972	138.8
Global,256,1,128 AC	64.9549603	124	411.359283	364.6	139.243994	431.9	138.284741	81.6	97.7830749	137.3
Compartido,128,2,32 AC	34.288105	126	216.863896	363.2	74.5060439	438	72.5581412	82.9	56.5781646	138.3
Compartido,128,1,64 AC	34.5847636	124.2	213.416294	365.5	73.0850838	436.6	72.7328963	83.4	52.5269308	140.1
Compartido,256,2,64 AC	68.1067633	124	427.984928	365	144.212103	433	138.417273	82.6	98.2642722	138.1
Compartido,256,1,128 AC	67.4484771	121.7	429.068003	361.4	139.036223	434.1	142.060453	82.1	100.116905	136.3
Local,128,2,32 AC	32.9921419	125	207.696505	364.8	70.207198	434.6	70.5585664	82.7	50.3506656	142.5
Local,128,1,64 AC	33.0202579	125.2	204.747229	365.8	70.6045974	436.5	72.2004636	82	49.8178983	141.3
Local,256,2,64 AC	67.0653446	123.4	430.594769	362.3	145.049938	431.2	148.267676	82.8	101.732809	138.4
Local,256,1,128 AC	67.0821071	122.8	412.174914	363.9	141.39629	431.6	137.739474	82	98.5904665	138
Compartido,128,1,64 AC	34.5847636	124.2	213.416294	365.5	73.0850838	436.6	72.7328963	83.4	52.5269308	140.1
Global,128,2,32	33.7784859	126.2	211.159676	366.7	71.5544534	434.9	72.8595934	82.3	51.8439337	141.2
Global,128,1,64	33.7664801	123.7	214.284681	365.6	70.9570144	437.9	71.6842514	82.8	52.8520553	140.2
Global,256,2,64	66.5663851	123.5	421.985079	363.5	145.053013	437.5	149.404148	82.5	101.116071	140
Global,256,1,128	65.5345762	123.7	411.801365	366	138.462901	433.1	139.13256	83.6	102.435103	138.6
Compartido,128,2,32	32.2797088	125.5	205.553325	367.1	69.78218	434.8	69.8541036	81.8	54.7158808	139.6
Compartido,128,1,64	33.6483535	125.1	213.496964	366.8	71.8928846	435.7	71.8097772	83.2	51.7925991	141
Compartido,256,2,64	66.5423517	122	430.630632	360.9	141.605158	432.6	142.871373	82.6	100.438336	138.9
Compartido,256,1,128	63.3256643	125.3	421.142926	362.3	142.19439	437	141.475995	80	100.910794	140.2
Local,128,2,32	32.3939242	127	206.529582	363.1	70.2711495	434.9	69.1058066	82.8	48.9696568	139.9
Local,128,1,64	32.3908168	127	207.107711	364.2	68.9168411	439.2	69.597785	82.9	48.9541056	141.6
Local,256,2,64	64.4675533	123.4	415.066547	362	142.391495	432.7	136.831269	80.7	99.8263281	138.3
Local,256,1,128	67.2900504	122.6	444.546514	361.6	144.741161	436.1	141.839243	82.2	99.9947874	139

Tabla 5.11: comparación de resultados de todas las diferentes estrategias para resolver el PCG

Como en las tablas anteriores, en la tabla 5.11 se resalta en gris los mejores resultados con respecto al tiempo y al número de aristas monocromáticas por grafo *benchmark*. Es muy claro la consistencia sobre los resultados, mostrando una convergencia hacia los algoritmos *greedy*, en particular *smallest last* con grafos que son muy sencillos y pequeños (debido al tiempo), ya que éstos algoritmos son conocidos por brindar soluciones rápidas, con resultados que pueden muchas veces llegar a ser muy buenas para grafos pequeños, pero si comparamos con los resultados obtenidos con los algoritmos *greedy* de grafos más grandes, se puede ver la gran diferencia con respecto al número de aristas que da como resultado a las que se muestran con cualquiera de los algoritmos genéticos.

Mientras que los grafos con mayor número de nodos, así como complejidad, se muestra que el algoritmo que dio los mejores resultados en ambos aspectos, es el algoritmo genético secuencial con la búsqueda local de *gradient descent* y una población de 128, si bien se puede observar que hay una pequeña mejora en el número de aristas cuando la población aumenta a 256, pero la diferencia de los resultados de NAM no es tan distinta y lo realiza en el doble de tiempo. Respecto a los resultados con el algoritmo genético paralelo, es posible observar que la configuración más cercana al algoritmo

secuencial, es el que utiliza la memoria local, con la diferencia de una ligera mejora en el tiempo, pero con un mayor número de aristas monocromáticas, por lo que se puede suponer que la memoria local es que es muy pequeña, y por lo tanto, es muy probable que realice varias copias de diferentes datos de la matriz, generando una competencia por los recursos al pasar por el bus, es decir que al tener esa competencia, esto hace que los distintos GPUs o hilos se formen en el bus para obtener los datos a la memoria global, perdiendo por completo la paralelización del algoritmo. Por otro lado, los resultados del número de aristas con los algoritmos genéticos paralelos, depende mucho de la calidad del generador de números aleatorios, ya que influye en la capacidad del algoritmo de búsqueda local para investigar el amplio espacio de soluciones dentro de un esquema discreto, lo cual lo hace más difícil.

Con el objetivo de medir la estabilidad de los algoritmos, se realizó el calculo de la desviación estándar de los 10 resultados obtenidos con cada una de las diferentes semillas. Para ello en primera instancia, se observa que de la tabla 5.2 a la 5.7 muestra el promedio de los resultados, así como el calculo de la desviación estándar de cada grafo, utilizando múltiples semillas.

Por otro lado, a continuación la siguiente tabla (5.12) muestra el calculo de las desviaciones estándar de las estrategias analizadas en este trabajo, como una medida de comparar la estabilidad de cada una de ellas. El objetivo general de esta tabla es comparar las estrategias *greedy*, el algoritmo genético híbrido secuencial y el algoritmo genético híbrido paralelo, por lo cual se tomaron como representantes a las siguientes técnicas: *smallest last* e *independent set*, el algoritmo genético híbrido secuencial con ascenso de colina y una población de 128, debido a que en la tabla 5.11 es posible observar que esta técnica fue la que brindó el mejor desempeño respecto a ambos criterios (tiempo y NAM), y por último el algoritmo genético híbrido paralelo con una población de 256, 1 bloque, 128 hilos, ascenso de colina y memoria global, la razón por la cual fue elegida es debido a que es la configuración CUDA que obtiene los mejores resultados en ambas búsquedas locales tomando en cuenta el número de aristas monocromáticas.

	myciel3		**myciel4**		**2-Fullin-3**		**Jean**		**Anna**	
	Tiempo	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**
smallest last	0.000441375	0	0.00057407	0	0.00174014	0	0.00185308	0	0.00285084	0
Independent set	0.001947615	0	0.00253953	0	0.00824411	3.34165628	0.01739643	5.75036231	0.01637259	2.34757558
Secuencial-AC-128	0.00032388	0	0.00206932	0	0.00371517	0	0.00940866	0	0.00704978	0
Global,256,1,128 AC	0.0080684	0	0.00805894	0	0.00736695	0	0.00754221	0	0.04230482	0

	DSJC250.5		**DSJC500.5**		**le450_15c**		**le450_25c**		**flat300_28**	
	Tiempo	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**	**Tiempo**	**NAM**
smallest last	0.01707276	298.865931	0.03684726	751.224157	0.01718636	299.978444	0.01925917	326.670901	0.02616811	461.482105
Independent set	0.0337703	141.819173	0.11195009	723.799665	0.05547509	166.862585	0.61735864	169.894739	0.46175604	438.893508
Secuencial-AC-128	0.23327789	3.27	0.54793679	4.62	0.60703447	5.97	0.32022311	1.42	0.46444386	2.3
Global,256,1,128 AC	0.64387822	3.16	2.64984948	5.2111	0.56012641	6.17252	0.69125545	2.45855	0.41849487	5.2504

Tabla 5.12: Desviación estándar de los resultados de las distintas estrategias

5.1.2. Experimentos con grafos aleatorios

Para la segunda parte experimental de este trabajo, además de los grafos *benchmark* también fueron creados 160 grafos aleatorios distintos, con los cuales fue posible aterrizar las ideas plantadas en la hipótesis. Lo anterior debido a que al obtener los resultados, fue posible comparar el comportamiento

del conjunto de algoritmos implementados para este trabajo, utilizando las distintas técnicas y configuraciones, dando la oportunidad de llegar a una conclusión estable acerca de los análisis, en base a los resultados del problema de coloreado de grafos.

Como se menciona en la sección 5.1 para crear a los grafos aleatorios fue necesario utilizar un modelo, en este trabajo se utilizó el modelo Gilbert G(n,p), el cual funciona por medio del número de nodos del grafo (n) y una probabilidad (p), en la que cada arista de dichos nodos, tiene esa probabilidad p de que exista o no dentro del grafo. De los 160 grafos aleatorios creados, el número de nodos va de veinte en veinte empezando con 20 hasta 100, y para cada uno fueron creados 10 grafos donde las probabilidades utilizadas variaban de 0.4 a 0.7, en la Tabla 5.13 se observa claramente lo anterior:

Grafo	Probabilidad	Número de nodos
10 Grafos	0.4 - 0.7	20
10 Grafos	0.4 - 0.7	40
10 Grafos	0.4 - 0.7	60
10 Grafos	0.4 - 0.7	80
10 Grafos	0.4 - 0.7	100

Tabla 5.13: Creación de grafos aleatorios utilizados en pruebas.

El siguiente paso, una vez creados los grafos con las especificaciones anteriores, fue la experimentación en el algoritmo genético secuencial, paralelo, así como en los algoritmos *greedy* para el problema de coloreado de grafos. Una de las diferencias con los grafos *benchmarks* fue que al generar las pruebas no fue necesario aplicar 10 semillas distintas, debido a que con los grafos aleatorios fue posible el crear 10 grafos para cada número de nodos y sus probabilidades. Gracias a los resultados de los 10 grafos aleatorios se calculó el promedio de los valores de todos los grafos. Como se realizó con las pruebas con grafos *benchmark*, la recolección de la información una vez más fue bajo las dos métricas anteriores: el tiempo (en segundos) y el número de aristas monocromáticas.

Como hemos estado mencionando, el algoritmo secuencial se realizó para dos tamaños de población: 128 y 256 individuos, utilizando cada una de las estrategias de búsqueda local (ascenso de colina y metrópolis). Para la versión paralela ($CUDA$) se ejecutó el algoritmo con cada grafo aleatorio, así como para cada una de las configuraciones $CUDA$, para los dos tipos de búsqueda local (ascenso de colina y metrópolis), estas también fueron las utilizadas en las pruebas con los grafos *benchmarks* (ver tabla 5.14).

Por último, lo que corresponde a los experimentos de los grafos aleatorios con los algoritmos *greedy*, fueron utilizadas las mismas estrategias *smallest last*, e *independent set*, así como los mismos criterios utilizados con los grafos benchmark, como el tiempo que tarda en dar un grafo ya coloreado con cada una de las estrategias, y sumando como arista monocromática aquella en que los nodos cuya etiqueta (color) fue mayor al número de colores (k).

Configuraciones CUDA
P128,2B,32H memoria global
P128,1B,64H memoria global
P256,2B,64H memoria global
P256,1B,128H memoria global
P128,2B,32H memoria compartida
P128,1B,64H memoria compartida
P256,2B,64H memoria compartida
P256,1B,128H memoria compartida
P128,2B,32H memoria local
P128,1B,64H memoria local
P256,2B,64H memoria local
P256,1B,128H memoria local

Tabla 5.14: Diferentes configuraciones CUDA utilizadas en la experimentación.

Los experimentos con grafos aleatorios se dividió en dos partes, la primera consistió en comparar las distintas configuraciones CUDA y obtener a los representantes de las mejores configuraciones para cada una de las estrategias de búsqueda local. Por otro lado, en la segunda parte se crearon las gráficas comparativas con los resultados de los grafos aleatorios con los algoritmos *greedy*, secuencial, y los representantes de las configuraciones $CUDA$. Para realizar la primera parte, se tomó los resultados de los grafos de 20, 60 y 100 nodos, con el objetivo de identificar una configuración ganadora para las estrategias de ascenso de colina y metrópolis, así como para el tiempo y el número de aristas monocromáticas.

En las Figuras 5.1 y 5.2 se observa los resultados obtenidos de los grafos aleatorios con 20 nodos utilizando el algoritmo genético híbrido paralelo con cada una de las configuraciones $CUDA$. En específico la Figura 5.1 muestra los resultados del tiempo con la estrategia de ascenso de colina, mientras que en la Figura 5.2 muestra los resultados del tiempo con metrópolis, es posible observar que en ambas figuras, existe una sutil diferencia entre las estrategias de ascenso de colina y metrópolis. Analizando a detalle se observó que para ambas estrategias, la configuración ganadora fue la *población 128, 1 bloque, 64 hilos con memoria local*.

Mientras que en las Figuras 5.3 y 5.4, se observan los resultados de los grafos de 20 nodos, pero esta vez respecto al número de aristas monocromáticas. En estas a diferencia de las anteriores, es posible observar a simple vista el cambio entre cada configuración, además de esto, al analizar cual fue la estrategia ganadora de las distintas configuraciones CUDA, el resultado mostró que para ambas estrategias gana la *población de 256, 1 bloque, 128 hilos con memoria local*. Con los resultados se puede establecer que para grafos pequeños como el de 20 nodos, existe una consistencia entre las estrategias respecto a cada una de las mediciones (tiempo y NAM), mientras que si tomamos en cuenta ambas mediciones es posible notar que la mejor respuesta siempre la brinda el utilizar memoria local, variando en la población y el número de hilos.

Por otro lado, al analizar los resultados del algoritmo con los grafos de 60 nodos (Figuras 5.5 y 5.6), esta vez es posible ver muchas diferencias, esto debido a la colocación de las configuraciones en ambas estrategias. Al analizar la configuración ganadora para cada una de ellas, resultó que en el tiempo la mejor configuración utilizando ascenso de colina fue con una *población de 128, 1 bloque, 64 hilos con memoria local*, mientras que para metrópolis gana la *población de 128, 1 bloque, 64 hilos con memoria compartida*. Lo que respecta al número de aristas monocromáticas, se observa un poco más de similitud en las Figuras 5.7 y 5.8, esto puede ser corroborado al ver que la configuración ganadora fue la misma para ambos casos, ganando la configuración de *población 256, 1 bloque, 128 hilos con memoria global*. Llegando a la conclusión que con grafos de 60 nodos en el tiempo y el número de aristas monocromáticas utilizando ambas estrategias, existe una consistencia en el tamaño de la población, bloques e hilos, pero una variación respecto a la memoria que brinda mejores resultados.

Al llegar a los grafos de 100 nodos, los resultados fueron diferentes como se ve en las gráficas 5.9 y 5.10, ya que al momento de analizar los resultados del tiempo, las configuraciones da como resultado que para ascenso de colina, la mejor es con una *población de 128, 1 bloque, 64 hilos con memoria global*, mientras que para metrópolis es con una *población de 128, 2 bloques, 32 hilos y memoria local*, mientras tanto, los resultados con el número de aristas monocromáticas es pronunciada la diferencia que existe entre ambas estrategias, como es posible observar la configuración ganadora para ascenso de colina es una *población de 256, 2 bloques, 64 hilos con memoria global*, mientras que para metrópolis la configuración fue de una *población de 256, 1 bloque, 128 hilos con memoria global*. Un detalle que se observa claramente, es que los resultados muestran congruencia tanto como en el tiempo como en el número de aristas monocromáticas, debido a que con población de 128 el tiempo de búsqueda del algoritmo genético va a ser siempre menor, y una población de 256 permite que existan más instancias del problema dentro del espacio de búsqueda, generando mejores resultados pero en más tiempo.

Una vez obtenido todos los resultados ganadores, se inspeccionó cuales eran las configuraciones que ganaban en dos de tres, dentro de los resultados con los grafos de 20, 60 y 100 nodos para cada estrategia, dividida en las categorías de tiempo y número de aristas monocromáticas, los resultados se muestran en la siguiente tabla:

Ascenso de colina		**Metrópolis**	
criterio	**configuración CUDA**	**criterio**	**configuración CUDA**
Tiempo	P128,1B,64H memoria local	*Tiempo*	P128,2B,32H memoria local
NAM	P256,1B,128H memoria global	*NAM*	P256,1B,128H memoria global

Tabla 5.15: Representantes de las configuraciones CUDA.

Los resultados de la Tabla 5.15, además de ayudar a identificar a la configuración representante que mejor categoriza a cada estrategia. Es posible observar las similitudes que existe en ambas, ya que para el tiempo se mantiene la consistencia planteada anteriormente de que con una menor población, menor será el tiempo de ejecución, al mismo tiempo podemos ver que en ambas gana la memoria local debido a que la transferencia de los datos, así como la competencia por los recursos es menor. Por otro lado, en el número de aristas monocromáticas existe una relación lógica en los resultados, utilizando la

misma configuración ganadora en ambas estrategias, volviendo al punto en el que con una población mayor, existe un incremento en el número de puntos dentro del espacio de búsqueda, brindando una mayor probabilidad de encontrar mejores, mientras que la memoria global tiene una mayor capacidad de almacenamiento que la local.

A continuación se muestran las figuras con los resultados del tiempo y el número de aristas monocromáticas de los grafos aleatorios de 20,60 y 100 nodos con las estrategias de ascenso de colina y metrópolis.

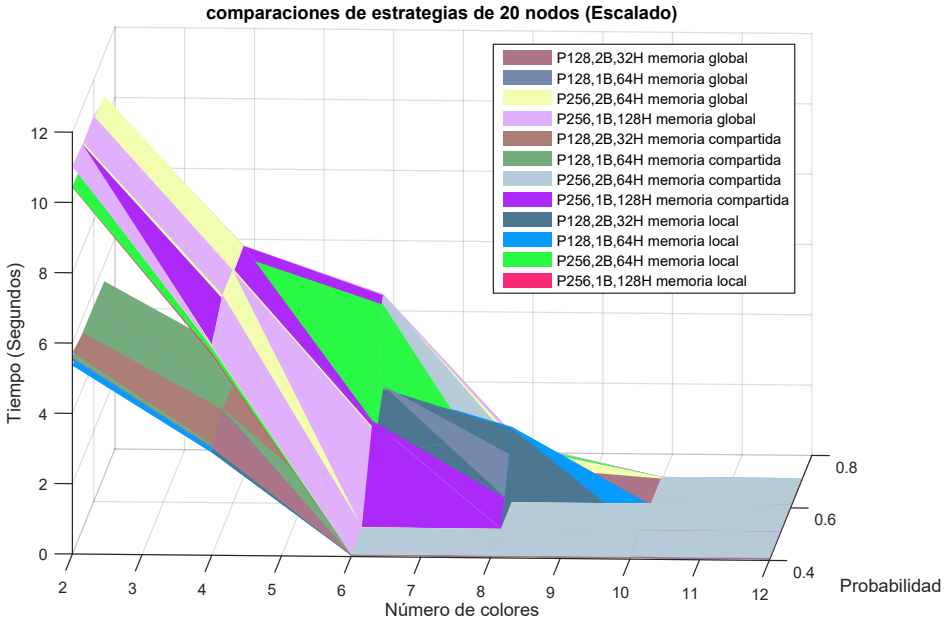

Figura 5.1: Gráfica de las diferentes configuraciones CUDA con 20 nodos y ascenso de colina.

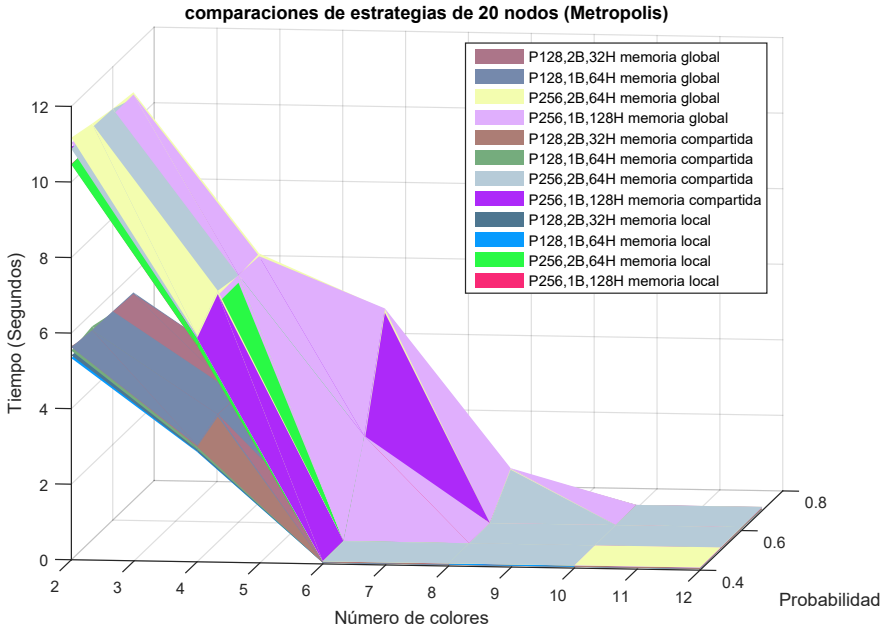

Figura 5.2: Gráfica del tiempo con las diferentes configuraciones CUDA con 20 nodos y metrópolis.

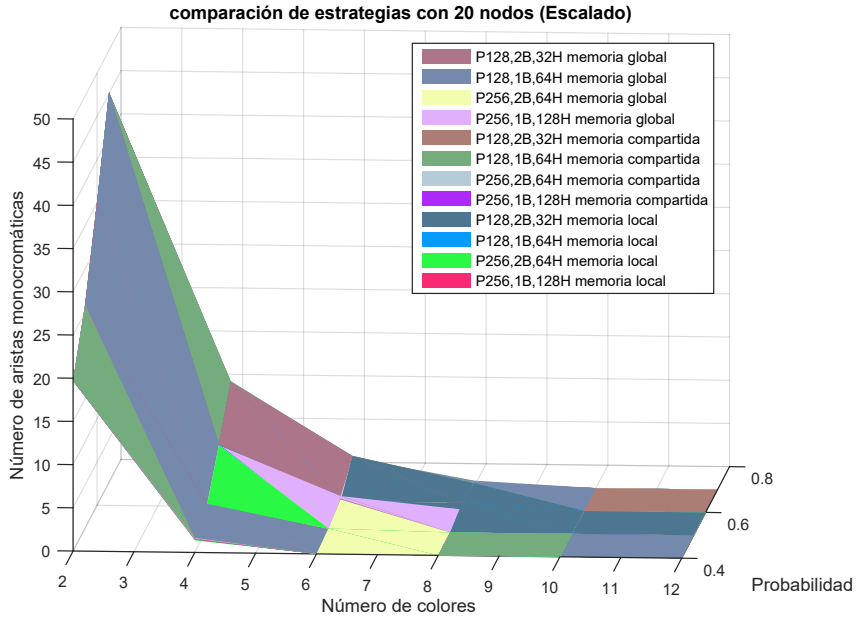

Figura 5.3: Gráfica del NAM con las diferentes configuraciones CUDA con 20 nodos y ascenso de colina.

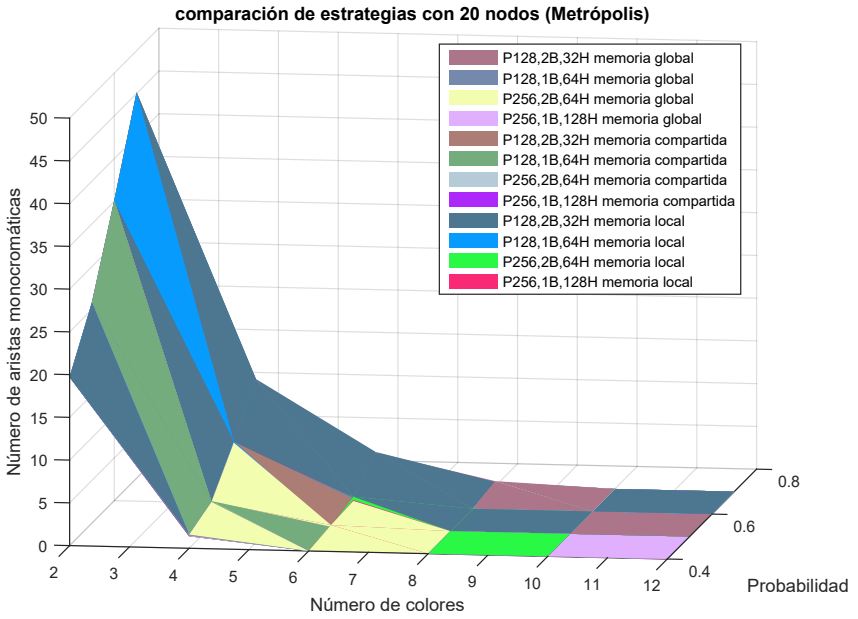

Figura 5.4: Gráfica del NAM con las diferentes configuraciones CUDA con 20 nodos y metrópolis.

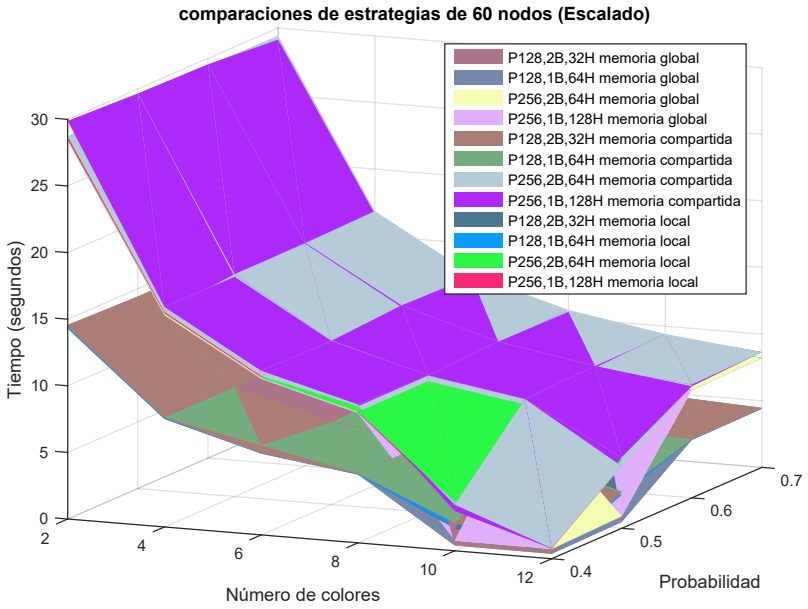

Figura 5.5: Gráfica de las diferentes configuraciones CUDA con 60 nodos y ascenso de colina.

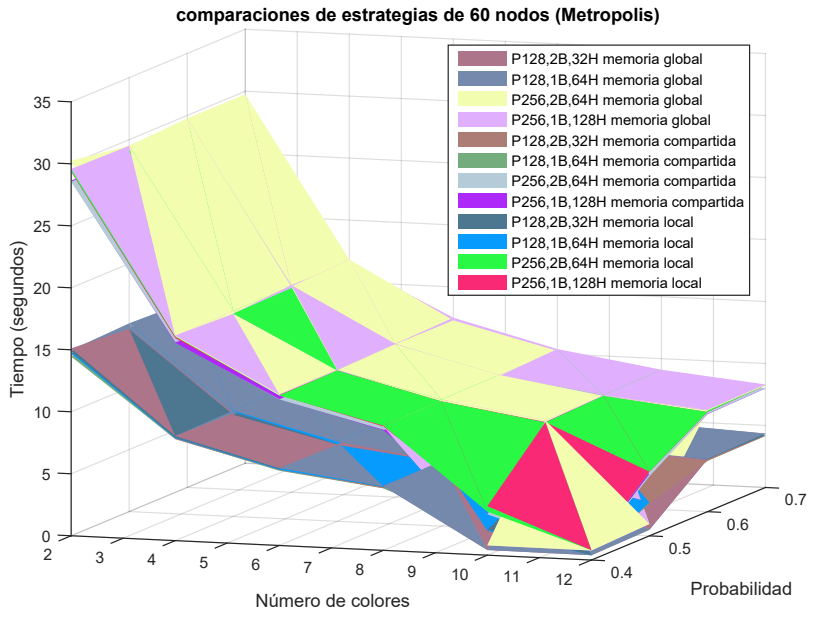

Figura 5.6: Gráfica de las diferentes configuraciones CUDA con 60 nodos y metrópolis.

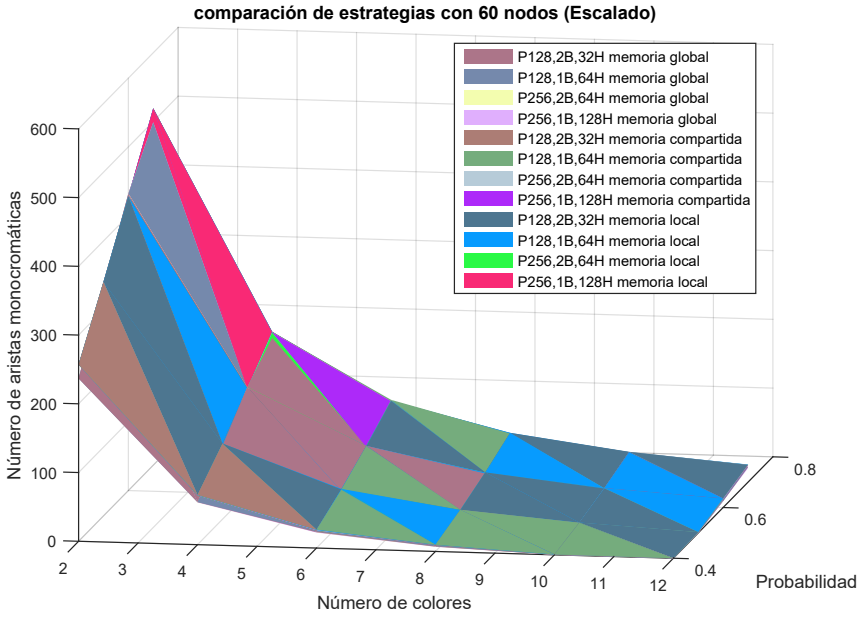

Figura 5.7: Gráfica del NAM con las diferentes configuraciones CUDA con 60 nodos y ascenso de colina.

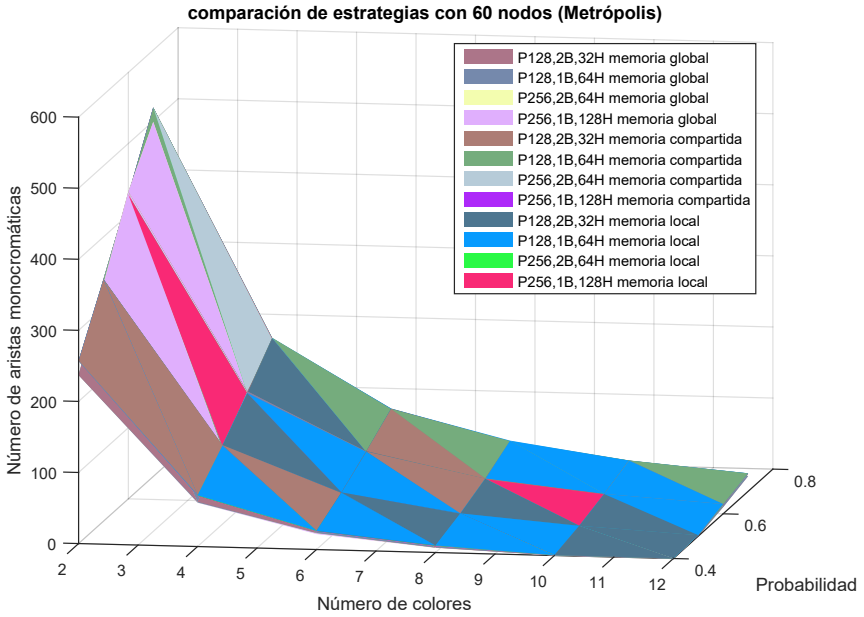

Figura 5.8: Gráfica del NAM con las diferentes configuraciones CUDA con 60 nodos y metrópolis.

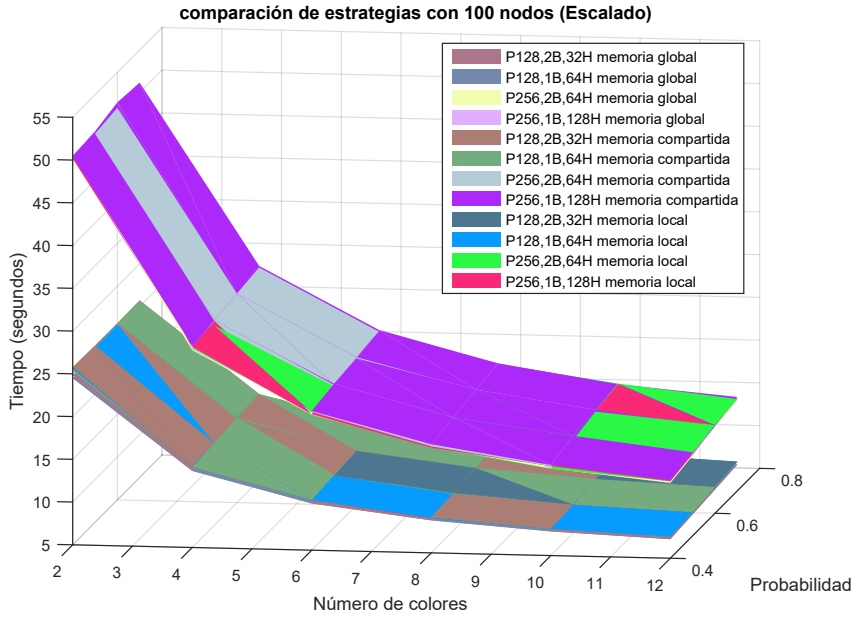

Figura 5.9: Gráfica de las diferentes configuraciones CUDA con 100 nodos y ascenso de colina.

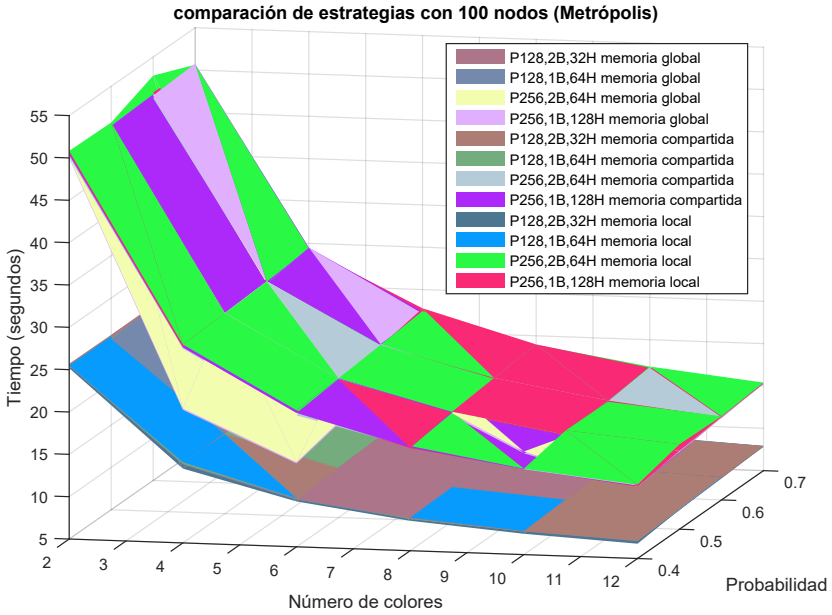

Figura 5.10: Gráfica de las diferentes configuraciones CUDA con 100 nodos y metrópolis.

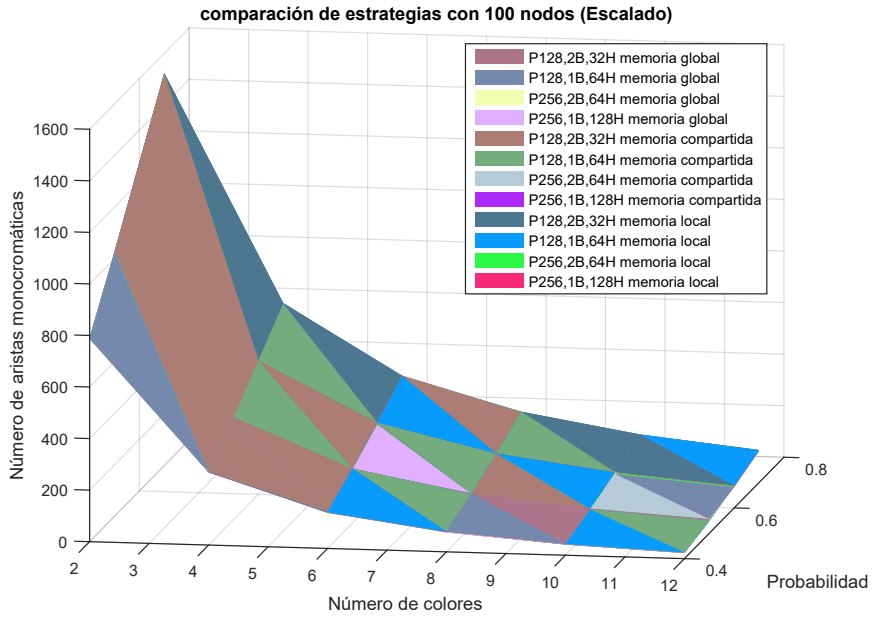

Figura 5.11: Gráfica del NAM con las diferentes configuraciones CUDA con 100 nodos y ascenso de colina.

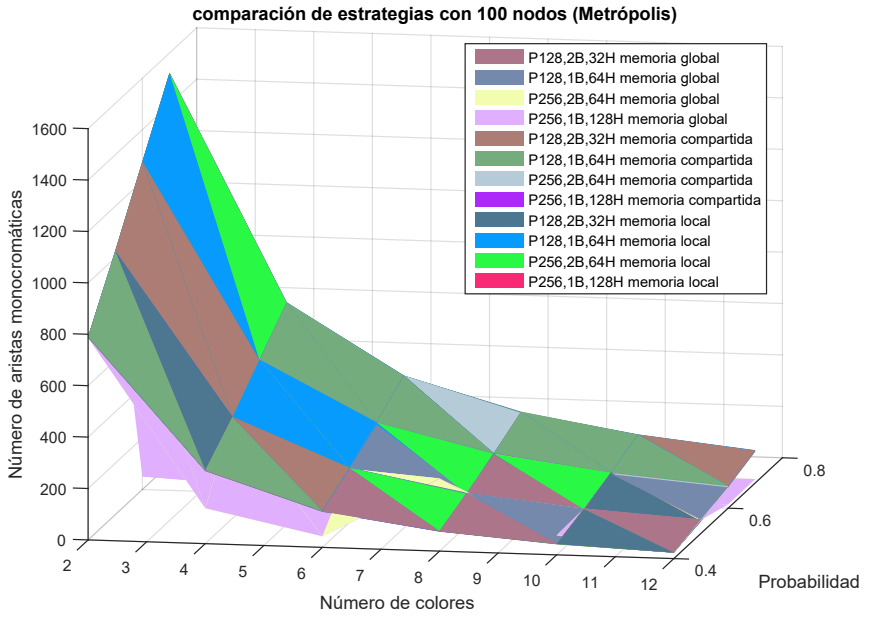

Figura 5.12: Gráfica del NAM con las diferentes configuraciones CUDA con 100 nodos y metrópolis.

En la segunda parte de las pruebas con grafos aleatorios, fue llevar a cabo las gráficas de las comparaciones de los distintos algoritmos utilizados en este trabajo: algoritmos *greedy* (*smallest last* e *independent set*), el algoritmo genético híbrido secuencial con ascenso de colina y metrópolis, y los representantes del algoritmo genético híbrido paralelo con ascenso de colina y metrópolis, obtenidos con las gráficas anteriores (vea el cuadro 5.15). Para estas pruebas al igual que con las configuraciones $CUDA$, se realizaron para los grafos con nodos de 20, 60 y 100.

En las Figuras 5.13 y 5.14, muestran los resultados comparativos de todos los algoritmos que resolvieron el problema de coloreado con los grafos aleatorios de 20 nodos. La primer figura (5.13) muestra el tiempo en segundos que les tomó resolverlo. En ella es muy clara la diferencia que existe de los algoritmos *greedy* sobre los algoritmos genéticos paralelos y los secuenciales, ya que los *greedy* brinda el mejor desempeño gracias a la rapidez con la que brinda los resultados, mientras que en los algoritmos paralelos y los secuenciales existe una reducción del tiempo de ejecución de hasta 25 %. Pero una vez que observamos la gráfica 5.14, la cual muestran los resultados del número de aristas monocromáticas, resalta el cambio observando a simple vista la diferencia entre las planicies de los algoritmos genéticos a los *greedy*, especialmente cuando el número de colores es pequeño y la probabilidad de que exista una arista es más alta, esto debido a que cuando la probabilidad es alta el grafo se encuentra más conectado, lo que significa que es más difícil, y la literatura menciona que los algoritmos *greedy* no garantizan buenos resultados para instancias difíciles de este problema.

Por otro lado, si vemos solamente a los resultados de los algoritmos genéticos híbridos (secuencial y paralelo) es posible observar que existe una variación en los resultados a lo largo de las distintas instancias de grafos, pero es posible ver que el algoritmo paralelo con la configuración de *población de 256, 1 bloque,128 hilos con memoria global y metrópolis* es la que se encuentra por debajo, especialmente cuando la probabilidad es alta y el número de colores es pequeño.

Mientras tanto, en las Figuras 5.15 y 5.16, muestran los resultados de los grafos aleatorios con 60 nodos. Una vez más es posible ver que en la figura del tiempo (5.15) que los algoritmos *greedy* están muy por debajo de los algoritmos genéticos híbridos. Mientras que el algoritmo paralelo da el mejor resultado con una reducción en el tiempo de hasta 23 % respecto al algoritmo secuencial, lo cual es importante y significativo cuando hablamos de instancias de grafos muy difíciles.

Por otro lado, en la figura que muestra los resultados del número de aristas monocromáticas (5.16). Es posible ver que así como con los grafos de 20 nodos, una vez más existe una diferencia de resultados, respecto al número de aristas monocromáticas de los algoritmos genéticos sobre los algoritmos *greedy*, pero esta vez los *greedy* no muestran gran diferencia al aumentar el número de colores, llegando a posibles soluciones cuando el número de colores es muy alto y la probabilidad es de 0.4. Mientras que en los algoritmos genéticos híbridos se observa que los colores de las planicies que quedan por arriba es la de los algoritmos secuenciales, una vez más quedando por debajo el algoritmo genético híbrido paralelo con la configuración de *población de 256, 1 bloque,128 hilos con memoria global y metrópolis.*

Por último, en las Figuras 5.17 y 5.18 muestran los resultados de los grafos aleatorios con 100 nodos. En la figura que muestra el tiempo que le tomó resolver el problema (5.17), existe la misma consistencia de las gráficas con 20 y 60 nodos, debido a que los algoritmos *greedy* siguen brindando el mejor desempeño en el tiempo respecto a los algoritmos genéticos híbridos. Mientras que los algoritmos

genéticos híbridos secuenciales y paralelos muestran una planicie similar a la de 60 nodos cuando tienen una probabilidad de 0.4 a 0.6, y el número de colores es mayor a ocho, el tiempo decae casi hasta alcanzar al algoritmo *greedy*, con la diferencia de que los grafos con 100 nodos esto no ocurre, si no que mantiene una curvatura consistente a medida que aumenta el número de colores. Además nuevamente existe una clara diferencia entre los algoritmos paralelos a los secuenciales, con una reducción en el tiempo de ejecución de hasta 11 %.

Por otro lado, en las figuras de los resultados del número de aristas monocromáticas con 100 nodos (5.18). Como lo hemos estado viendo los algoritmos *greedy* quedan muy por encima de los algoritmos genéticos híbridos. Mientras que al tomar solamente a los algoritmos genéticos híbridos, se muestra una variación de los resultados a medida que el número de colores y probabilidad cambian, pero una vez más en la mayoría de los casos, es visible que los algoritmos secuenciales quedan por encima de los paralelos, así como la consistencia en la que se mantiene el algoritmo genético híbrido paralelo, con la configuración de *población de 256, 1 bloque,128 hilos con memoria global y metrópolis* brindando valores por debajo de los demás.

Demostrando con las figuras que a pesar de que los algoritmos *greedy* generan una notable mejoría en el tiempo, si tomamos en consideración los dos aspectos (tiempo, NAM), el algoritmo que da el mejor rendimiento es el *genético híbrido paralelo*, en específico con una *población de 256, 1 bloque,128 hilos con memoria global y metrópolis*.

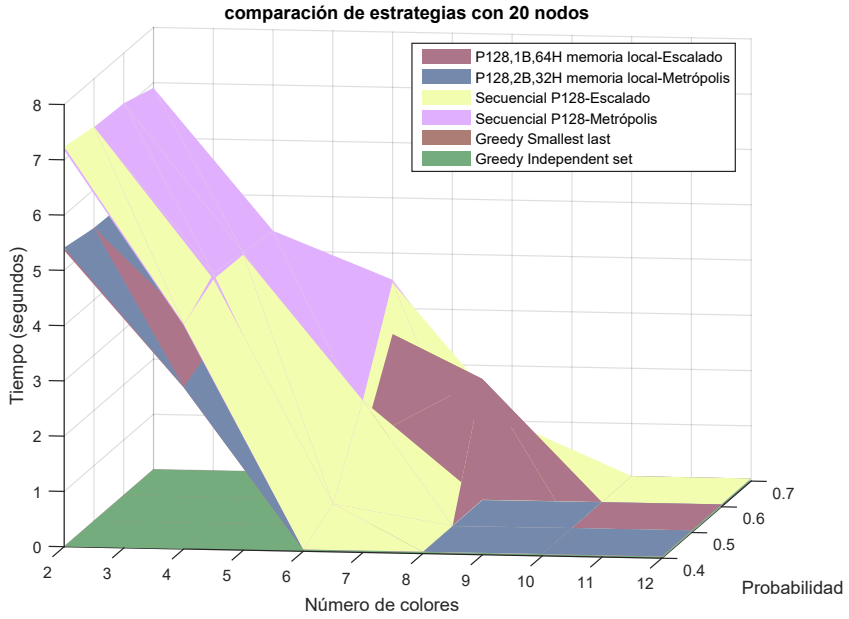

Figura 5.13: Gráfica comparativa del tiempo de los algoritmos con 20 nodos.

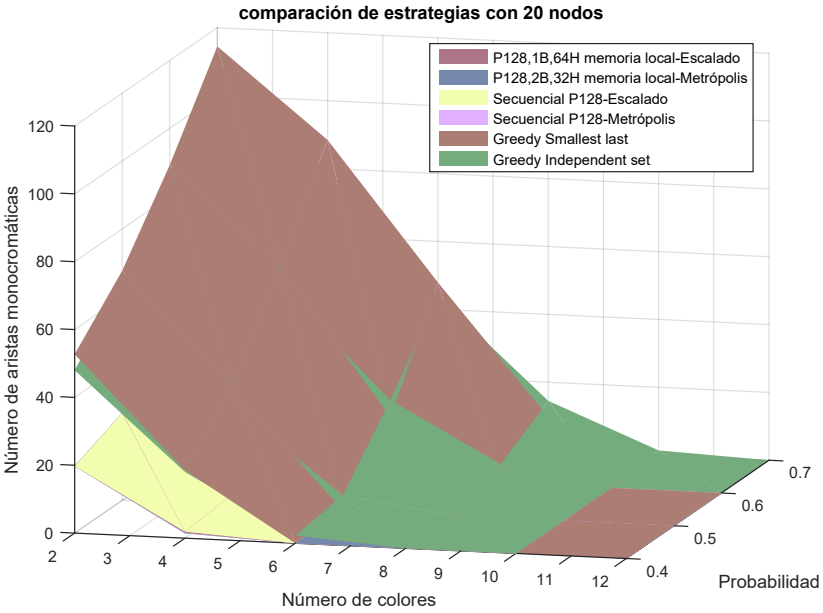

Figura 5.14: Gráfica comparativa del NAM de los algoritmos con 20 nodos.

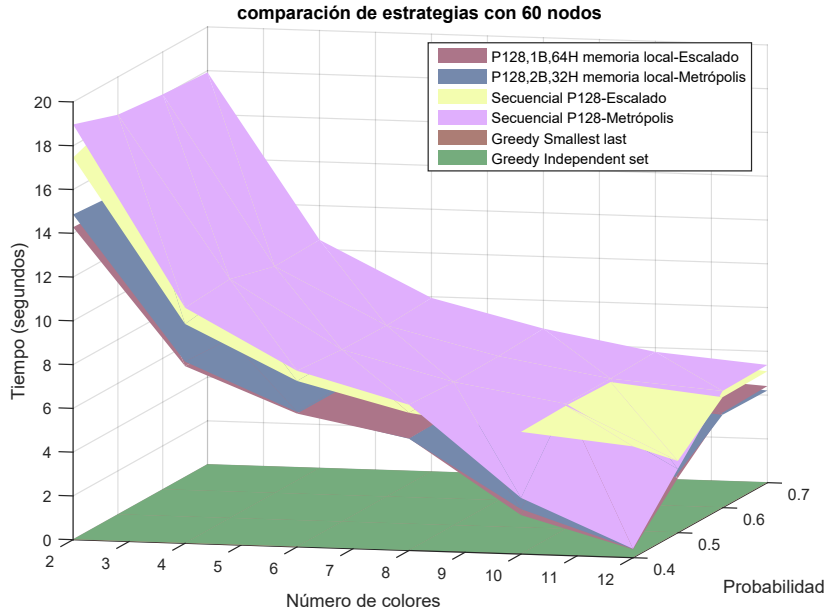

Figura 5.15: Gráfica comparativa del tiempo de los algoritmos con 60 nodos.

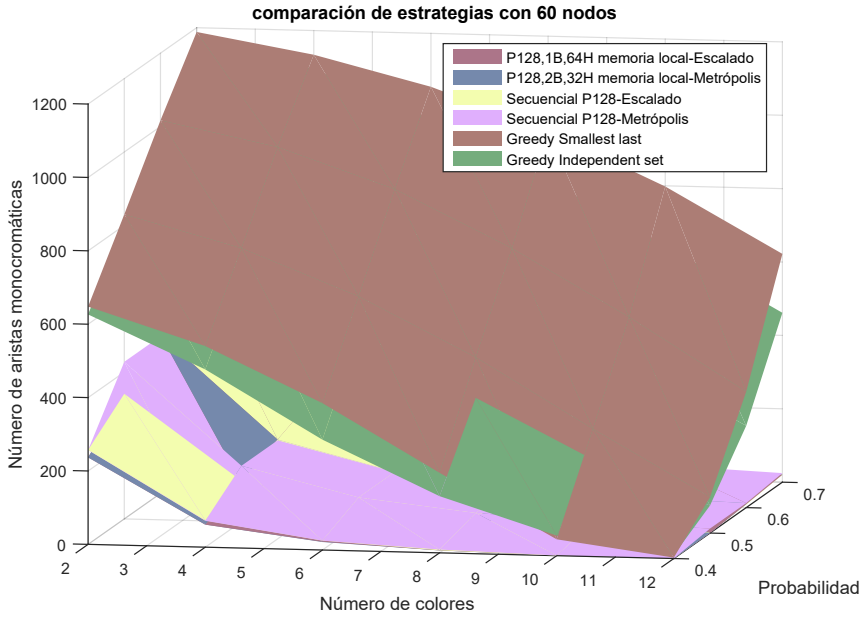

Figura 5.16: Gráfica comparativa del NAM de los algoritmos con 60 nodos.

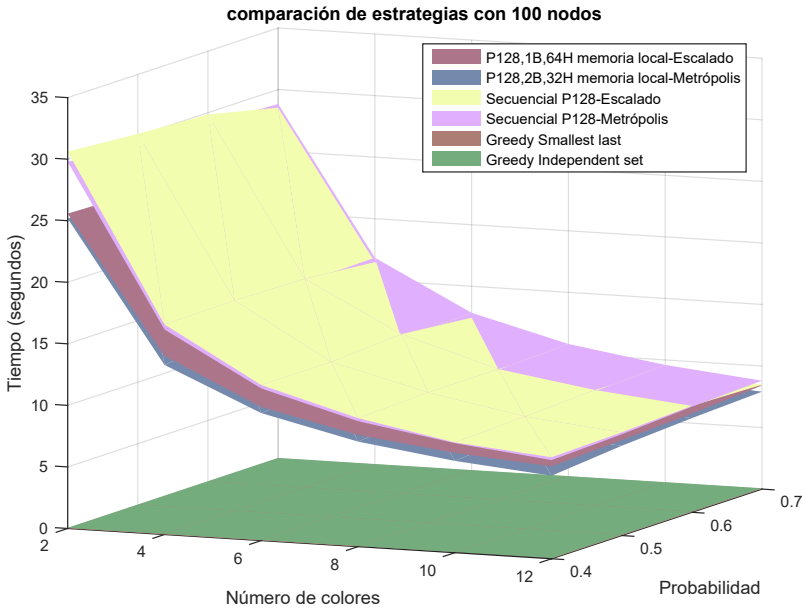

Figura 5.17: Gráfica comparativa del tiempo de los algoritmos con 100 nodos.

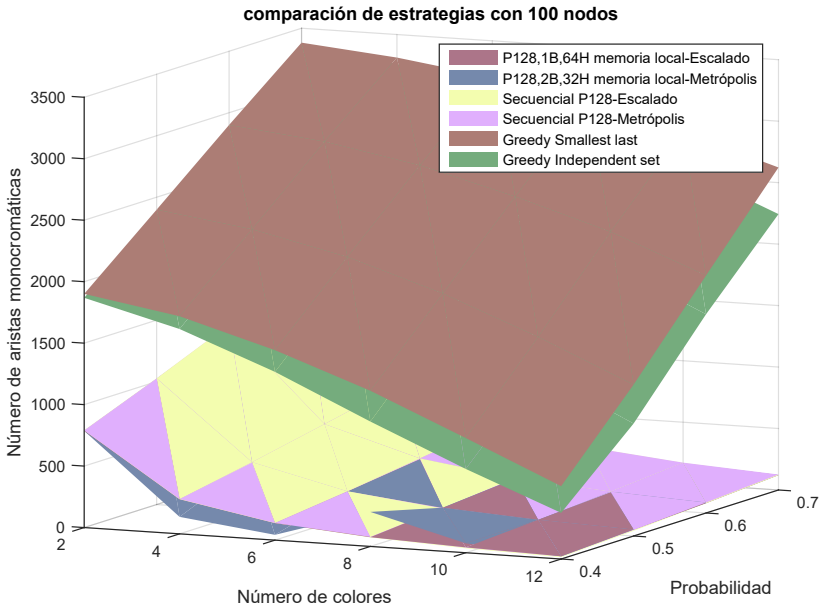

Figura 5.18: Gráfica comparativa del NAM de los algoritmos con 100 nodos.

Conclusiones y Trabajo a futuro

La literatura menciona, que una de las principales dificultades del problema de coloreado de grafos, es debido a que este se encuentra en la clasificación de problemas NP-Difíciles. Por lo cual, existen distintos algoritmos y técnicas para tratar de resolverlos. A lo largo de la historia, a medida que los algoritmos genéticos han demostrado ser competentes en distintas áreas de problemas, surge la inquietud de implementarlos en espacios de búsqueda discretas de los problemas NP-Difíciles. Inicialmente, se realizaron mediante técnicas convencionales como los *mapas de bits* para la representación del individuo, obteniendo resultados no satisfactorios, por lo cual se han propuesto otras iniciativas y mejorar los resultados dentro de este espacio.

Uno de los algoritmos que han sido utilizados para resolver el problema de coloreado de grafos son los *greedy*, debido a su facilidad de implementación, así como la rapidez con la que se obtienen los resultados, pero al realizar un análisis se percibe la mala calidad de los mismos.

En los algoritmos genéticos un aspecto muy importante es la *mutación*, ya que gracias a ella permite que el algoritmo no se estanque en óptimos locales, permitiendo un poco de aleatoriedad al mismo. Debido a que al tratar de resolver problemas combinatorios difíciles con algoritmos evolutivos convencionales no resulta una buena estrategia, múltiples investigaciones proponen utilizar estas estrategias en conjunto con búsquedas locales, ya que brindan una aleatoriedad similar a la que ofrece la mutación. Este tipo de algoritmos fueron llamados *meméticos o híbridos*. Los buenos resultados obtenidos en problemas combinatorios, demuestran su competencia en el área. Debido a que el problema de coloreado de grafos está dentro de un espacio discreto, es muy importante generar una buena representación del individuo ya que sin ella no es posible obtener información relevante del problema. Por ello se ha propuesto que sea mediante "k" bolsas (colores) en las cuales caen los nodos, dando la oportunidad de obtener información del individuo, así como generar una cruza sencilla y significativa.

En Galinier & Hao [9] proponen utilizar la representación del individuo con las "k" bolsas en conjunto con algoritmos greedy para la inicialización de la población, mientras que para la cruza utiliza los algoritmos *greedy* con un enfoque de partición llamado *Greedy Partition Crossover* (GPX). A diferencia del artículo, en el diseño del algoritmo propuesto en este trabajo, en la mutación del individuo fueron utilizados dos estrategias de búsqueda local (escalado y metrópolis), en la cual escalado siempre ofrece aceptar una configuración cuando garantiza un mejor resultado, mientras que metrópolis se basa en una probabilidad de que acepte una configuración dentro del vecindario, aún si no genera una minimización en el número de aristas monocromáticas. Este tipo de búsquedas locales abren la oportunidad a explorar otros caminos dentro del espacio de búsqueda.

El utilizar ambas estrategias dio la oportunidad de observar el comportamiento de cada una dentro del algoritmo genético, así como generar un análisis comparativo de los resultados obtenidos en la cual se garantice cual es la mejor estrategia para este problema.

Una vez diseñado el algoritmo genético híbrido secuencial, se pasó a paralelizar la búsqueda local (mutación en el AG) en $CUDA$. Debido a ciertas restricciones de la libería *numba* de $CUDA$ acerca del tipo de datos que pueden ser utilizadas dentro del *kernel*, fue necesario que para el nuevo algoritmo paralelo se implementara mediante una matriz de adyacencia, y un generador de números

aleatorios paralelo. Las características *CUDA* se basó en implementar diferentes versiones con distintas configuraciones con respecto al número de bloques, hilos y tamaño de población.

Todos los experimentos fueron realizados sobre dos conjuntos de grafos: *benchmarks* y *aleatorios* en los cuales los resultados demostraron que para los grafos *benchmark* (instancias muy difíciles) el algoritmo genético híbrido secuencial con *búsqueda local escalado y una población de 128* es la que da el mejor desempeño, tanto en el tiempo, como en el número de aristas monocromáticas (demostrado en las tablas). Por otro lado, con los grafos aleatorios tomando en cuenta solamente a los algoritmos genéticos se observa que mediante el número de nodos aumenta, existe una variación con los resultados. En los grafos con 20 nodos si tomamos de referencia solamente el tiempo, el algoritmo genético híbrido paralelo con escalado es el que brinda el mejor desempeño, mientras que en el número de aristas monocromáticas el algoritmo genético híbrido secuencial, población 128 y metrópolis es el mejor. Por otro lado, cuando el grafo aumenta a 60 nodos el algoritmo que da el mejor resultado en tanto el tiempo como aristas monocromáticas, es el algoritmo genético híbrido paralelo con escalado. Por último, cuando aumenta a 100 nodos, existe nuevamente una consistencia con el tiempo y con el número de aristas monocromáticas con el algoritmo genético híbrido paralelo y metrópolis.

En conclusión, como se menciona anteriormente, los resultados demostraron que con grafos *benchmarks* el algoritmo secuencial es el que brinda el menor tiempo y número de aristas monocromáticas, la hipótesis que resulta del análisis de estos, es que existe un bajo desempeño en la calidad de números aleatorios paralelos a diferencia con los generadores de números aleatorios secuenciales, afectando la calidad de resultados del algoritmo de búsqueda local dentro del espacio de soluciones. Mientras que con grafos aleatorios, en la mayoría de los casos existe una clara superioridad de los algoritmos genéticos híbridos paralelos, demostrando la hipótesis de que el paralelizar los algoritmos, reduce el tiempo de ejecución.

Los resultados demuestran que además del tiempo, existe una reducción en el número de aristas monocromáticas. Esto no se consideró en la hipótesis, pero los resultados fueron favorecedores en ambos aspectos. Además de lo anterior, al realizar el análisis comparativo de los distintos algoritmos, entre ellos las mejores configuraciones CUDA para cada búsqueda local, demostraron que dentro de los algoritmos genéticos en el tiempo, la mayoría de los casos la configuración con una *población de 128, 2 bloques, 32 hilos con memoria local y metrópolis* es la que da los mejores resultados, mientras que en el aspecto del número de aristas monocromáticas la configuración con una población de *256,1 bloque, 128 hilos con memoria global y metrópolis* es la que brinda el mejor desempeño.

Estos resultados son congruentes con el aspecto analizado, debido a que cuando se está evaluando el tiempo, es importante tener una población más pequeña, así como el uso de la memoria local ya que la transferencia de los recursos es menor. Por otro lado, si queremos una solución de mayor calidad es decir, que si nos enfocamos en el número de aristas monocromáticas es importante una población más extensa que abarque mayores puntos dentro del espacio de búsqueda. En ambos aspectos la constante es que la búsqueda local metrópolis es la que da el mejor resultado, concluyendo que la posibilidad de salir de óptimos locales ayuda a abrir caminos a otros puntos dentro del espacio, como para encontrar mejores soluciones, así como un tiempo más corto para encontrar posibles soluciones.

5.2. Verificación de hipótesis

- Los resultados de los experimentos realizados como parte de este trabajo de tesis muestran que los algoritmos evolutivos híbridos paralelos implementados son capaces, en general, de encontrar mejores soluciones que los algoritmos voraces. Esto se debe, principalmente, a la capacidad de los algoritmos evolutivos de explorar más extensivamente el espacio de posibles soluciones, y de explotar las diferentes regiones del espacio de soluciones a partir de las búsquedas locales. Los algoritmos voraces, por su parte, tienden a quedar atrapados en óptimos locales que pueden ser arbitrariamente malos.

- Con respecto a la hipótesis 2, los resultados experimentales muestran que al implementar de manera paralela la etapa de búsqueda local, es posible reducir el tiempo de ejecución del algoritmo evolutivo híbrido. Sin embargo, en este caso es importante resaltar que dadas las limitaciones de memoria de la plataforma CUDA utilizada, surgen condiciones de competencia que limita la reducción del tiempo de ejecución. En el caso concreto de la implementación realizada, la copia de la matriz de adyacencia desde la memoria principal hasta los diferentes niveles de la jerarquía de memoria de la arquitectura CUDA resultó imponer una sobrecarga con un impacto considerable.

- El análisis de la literatura reveló que un aspecto fundamental para aplicar exitosamente algoritmos evolutivos a problemas de optimización combinatoria es utilizar formas no tradicionales de codificar a los individuos que componen a la población. Por lo anterior, en el presente trabajo se optó por codificar a los individuos por medio de k-particiones de los nodos, donde cada una de las particiones representa el color que les es asignado. Por otro lado, un aspecto fundamental que se identificó a partir del análisis de los trabajos relacionados es que los algoritmos evolutivos, cuando son aplicados a problemas combinatorios, requieren de mecanismos efectivos para explotar el espacio de soluciones. Uno de los mecanismos de explotación, es precisamente la búsqueda local, ya que permite encontrar soluciones maximales cercanas a las soluciones encontradas después de la etapa de cruza.

- A partir del análisis de los trabajos relaciones y de datos preliminares de desempeño de iteraciones tempranas del diseño del algoritmo, se diseñó un algoritmo evolutivo híbrido paralelo cuyos principales componentes son la inicialización de la población, en la que utiliza al individuo de k-particiones junto con una técnica *greedy* llamada *Greedy saturation* utilizando un *heap* y dos criterios de ordenamiento, otro componente es la cruza, en la cual una vez más utiliza una técnica *greedy* llamada *Greedy partition crossover* en las que el individuo se conforma de las clases de colores que contenga el mayor número de nodos de cada padre, y por último las búsquedas locales, en las que se diseño y modificó el algoritmo para que pudiera ser implementado dentro de la plataforma CUDA. Respecto a los aspectos de diseño más destacables, es utilizar estructuras de datos soportadas por las plataformas de cómputo paralelo basadas en tarjetas gráficas, así como el diseño de algoritmos para diferentes esquemas de gestión de memoria como global, compartida y local. Así mismo, se diseñaron alternativas a la búsqueda tabú que no requieran grandes cantidades de memoria. Lo anterior es de suma importancia dadas las restricciones de memoria de las plataformas de cómputo paralelo basadas en procesadores gráficos.

- Para la implementación del algoritmo genético en la plataforma CUDA, fue necesario utilizar python en conjunto con anaconda, las cuales cuentan con la biblioteca *numba* que permite paralelizar las funciones utilizando los GPUs de CUDA. Tomando como base el algoritmo genético secuencial se llevó a cabo la paralelización de las búsquedas locales realizando un conjunto de versiones en cual variaba el tamaño de la población, el número de bloques, el número de hilos, así como las distintas jerarquías de memoria que permite la plataforma CUDA. Debido a las restricciones de los tipos de datos que permite utilizar la plataforma dentro de sus *kernels*, fue muy importante analizar las opciones y utilizar las que se consideraron pertinentes. Al realizar el análisis de las opciones de datos que soporta la plataforma fue clara la carente variedad de los mismos. Dentro de las búsquedas locales era muy importante un generador de números aleatorios, pero debido a que el utilizado dentro del algoritmo secuencial no es soportado por la plataforma CUDA, fue necesario el uso del generador de números aleatorios paralelo, mostrando un bajo desempeño y afectando a la calidad de los resultados del algoritmo genético híbrido paralelo.

- Se utilizaron las instancias myciel3, myciel4, 2-Fullins-3, Jean, Anna, DSJC250.5, DSJC500.5, le450_15c, le450_25c, y flat300_28 para caracterizar el desempeño de las diferentes variantes del

algoritmo propuesto, así como de una serie de algoritmos voraces que componen el estado del arte en el cálculo de funciones de coloreado de grafos. Los experimentos demostraron que los algoritmos evolutivos desarrollados como parte de la presente tesis consistentemente encuentran mejores soluciones que los algoritmos voraces. Los resultados también mostraron, que la paralelización de las etapa de búsqueda local es capaz de reducir el tiempo de ejecución del algoritmo evolutivo. Finalmente, los resultados también muestran que, como se esperaba, los algoritmos voraces son altamente eficientes en términos de su tiempo de ejecución.

- Con el propósito de caracterizar el impacto de incrementar el número de GPUs, tanto en el tiempo de ejecución del algoritmo, como en la calidad de las soluciones (al incrementar el tamaño de la población), se realizaron un conjunto de experimentos donde se varió el número de GPUs utilizadas, que van desde 64 hasta 128. Los resultados revelaron que la variante del algoritmo que más mejoró su desempeño al incrementar el número de GPUs respecto al tiempo fue para búsqueda ascenso de colina con una *población de 128, 1 bloque, 64 hilos con memoria local*, mientras que para metrópolis fue con *población de 128, 2 bloques, 32 hilos con memoria local*. En el número de aristas monocromáticas la mejor configuración en ambas estrategias fue con una *población de 256, 1 bloque, 128 hilos con memoria global*. Por otro lado, es importante resaltar que debido a las condiciones de competencia generadas al compartir tanto la jerarquía de memoria de la arquitectura CUDA, como el uso de los buses de comunicación, no se observó que el tiempo de ejecución del algoritmo propuesto disminuyera de manera proporcional al número de GPUs. Por esto último, es importante seguir investigando sobre nuevas implementaciones que hagan un uso aún más eficiente de la arquitectura CUDA con el fin de mejorar, aún más, el tiempo de ejecución.

- Con el propósito de caracterizar el impacto de utilizar diferentes esquemas de memoria de la tarjeta gráfica en el tiempo de ejecución del algoritmo, se realizaron un conjunto de experimentos donde se consideraron diferentes estrategias para el alojamiento de las estructuras de datos necesarias para implementar la búsqueda local en la tarjeta gráfica. En concreto se experimentó con grafos aleatorios con 20, 60, 100 nodos cada uno con 10 grafos distintos jugando con las probabilidades (0.4 a 0.7) de que exista una arista. Los resultados muestran que la configuración con una *población de 256,1 bloque, 128 hilos con memoria global y metrópolis* fue la que presentó un mejor desempeño. Esto se debe principalmente a que una población grande asegura contener más puntos dentro del espacio de búsqueda, mientras que la búsqueda local metrópolis da la oportunidad de repartir los puntos, además de que la nobleza que brinda esta búsqueda permite no quedar estancados en óptimos locales, como encontrar nuevos caminos a posibles soluciones potenciales.

5.3. Trabajos a futuro

Por último, de los resultados de este trabajo de tesis se propone el uso de nuevas estrategias e hipótesis para un trabajo a futuro. Las propuestas son las siguientes:

1. Aplicar la herramienta de software desarrollada para resolver problemas reales que pueden ser modelados como el problema de coloreado de grafos. En particular problemas en el contexto de planificación de tareas [20], planificación de grupos de trabajo [21], diseño de compiladores [22], reconocimiento de patrones [23], redes de computadoras [24] [25].

2. Diseñar, implementar y caracterizar experimentalmente nuevas estrategias de búsqueda local específicamente diseñadas para plataformas de hardware con memoria restringida como las tarjetas CUDA. En este mismo sentido, es importante seguir investigando sobre algoritmos concurrentes

que reduzcan las condiciones de competencia que se presentan cuando varios núcleos de ejecución intentan acceder a la memoria compartida.

3. Extender la investigación reportada en la presente tesis a otros problemas de optimización combinatoria que pertenezcan a la clase NP-difícil como cobertura de vértices, conjunto independiente, conjunto dominante, ciclo Hamiltoneano, entre otros.

Apéndice

Es posible descargar el código desde la siguiente liga a Github:

- https://github.com/Jessik167/Tesis-GC.Python.git

5.4. Algoritmo genético híbrido secuencial

5.4.1. Main

```
from numba import cuda, vectorize
from numba.cuda.random import create_xoroshiro128p_states
import numpy as np
import networkx as nx
import scipy as sp
import bolsas
import Cruza
import Busquedas_locales
import random
import copy
import math
import time
import winsound

#Datos del Algoritmo genetico
tam_poblacion = 256
numGeneraciones = 20
numColores = 4

'''Lee el grafo desde un archivo'''
def Lee_Grafo(nombre):
return nx.read_edgelist(nombre, nodetype=int)

'''Inicializa la poblacion con estrategia greedy,
 y luego realiza una busqueda local'''
# Crea instancias de colorado del grafo con
 estrategia greedy de tamano de la poblacion
def InicializacionPoblacion(poblacion,
 probabilidades, G,numNodos):
#Lo realiza hasta el tamano de la poblacion
for ind in range(tam_poblacion):
if ind == 0:
poblacion.append(TomaGreedy(G,numNodos))
else:
#crea un individuo
poblacion.append(bolsas.crea_individuo
(G, numColores,numNodos))
# cuenta los nodos de cada bolsa del individuo
probabilidades.append(bolsas.cuenta_nodos
(poblacion[ind], numColores, numNodos))

'''Toma la solucion greedy smallest last
 y la convierte a un individuo en diccionario'''
def TomaGreedy(G,numNodos):
#Utiliza estrategia greedy
d = nx.coloring.greedy_color(G,
strategy='smallest_last')
Bolsas_colores = {k: {} for k in range(numColores)}
for nodo in range(1, numNodos+1):
if nodo in d.keys():
#Pregunta si el color del greedy esta dentro
 del rango de colores
if d[nodo] < numColores:
#Si si, ingresa el nodo en la bolsa
 que eligio el greedy
Bolsas_colores[d[nodo]][nodo] = nodo
else:
#Si no esta, ingresa el nodo en una bolsa aleatoria
Bolsas_colores[random.randint(0, numColores - 1)]
[nodo] = nodo
else:
# Si no esta, ingresa el nodo en una bolsa aleatoria
Bolsas_colores[random.randint(0, numColores - 1)]
[nodo] = nodo
return Bolsas_colores

'''Recibe dos padres, los cruza con el algoritmo GPX,
 retorna un nuevo individuo (hijo)'''
def CruzaPadres(individuos,probabilidades,
AMonoNuevo,numNodos,G):
# Lista que contendra al nuevo individuo
nuevo_indiv = {}
# Cruza a los padres y forma un nuevo individuo
nuevo_indiv = Cruza.GPX(individuos, numColores)
# cuenta los nodos de cada bolsa del individuo
probabilidades.append(bolsas.cuenta_nodos
(nuevo_indiv, numColores, numNodos))
# Calcula el numero de aristas monocromaticas
 del individuo nuevo
AMonoNuevo.append(Busquedas_locales.numeroAristasMono
(G, nuevo_indiv,numColores))

return nuevo_indiv

'''Toma aleatoriamente dos de los indices dentro
 del total de la poblacion,
que seran los padres que se cruzaran'''
def EligePadres(poblacion,numNodos, padres
,probabilidades):
#Lista que contendra a los dos individuos a elegir
individuos = []
indices = ()
#Elige el indice a elegir del padre 1
indices += (random.randint(0, tam_poblacion - 1),)
while True:
r = random.randint(0, tam_poblacion - 1)
if r != indices[0]:
#Elige el indice a elegir del padre 2
indices += (r,)
break
padres.append(indices)
# toma el individuo al azar de la poblacion (padre 1)
individuos.append(copy.deepcopy(poblacion[padres[-1][0]]))
# toma el individuo al azar de la poblacion (padre 2)
individuos.append(copy.deepcopy(poblacion[padres[-1][1]]))
return individuos

'''Manda a llamar a la busqueda local
(metropolis o Escalando la colina)'''
def BusquedaLocal(nuevo_individuo, probabilidad,
 Aristmono, M):
# regresa al individuo despues de realzar la busqueda local
#nuevo_individuo, Arist = Busquedas_locales.
Busqueda_Escalando(M, nuevo_individuo,
probabilidad,Aristmono, numColores)
# regresa al individuo despues de realzar la busqueda local
nuevo_individuo, Arist = Busquedas_locales.
Busqueda_Metropolis(M, nuevo_individuo,
```

```
probabilidad, Aristmono,numColores)
return Arist

'''Reemplaza al peor de los padres elegidos con el hijo,
compara el numero de aristas monocromaticas de los
individuos implicados'''
def ActualizaPoblacion(poblacion, hijo, AristasMono,
AristasMonohijo,probabilidades, probab_hijo, indiceP):
#Si el padre 2 tiene mayor numero de aristas mono que
 el padre 1 entonces...
if AristasMono[indiceP[0]] < AristasMono[indiceP[1]]:
#El hijo reemplaza al padre 2
poblacion[indiceP[1]]= hijo
#actualiza las aristas mono del padre reemplazado
 con las del hijo
AristasMono[indiceP[1]]= AristasMonohijo
probabilidades[indiceP[1]]= probab_hijo
#Si no...
else:
#El hijo reemplaza al padre 1
poblacion[indiceP[0]] = hijo
#actualiza las aristas mono del padre reemplazado
 con las del hijo
AristasMono[indiceP[0]] = AristasMonohijo
probabilidades[indiceP[0]] = probab_hijo

'''Convierte el hijo en un diccionario de colores'''
def convert_hijo(Arr,numNodos):
Bolsas_colores = {k: {} for k in range(numColores)}
for i in range(numColores):
for j in range(numNodos):
if Arr[i][j] == 1:
Bolsas_colores[i][j] = j
return Bolsas_colores

'''Compara los resultados'''
def sonIguales(resultados,tam):
#Toma el primer resultado
primero = resultados[0]
i = 1
#Lo compara con todos los demas
for i in range(tam):
#Si es igual continua
if primero == resultados[i]:
continue
else:
#Si no, termina
return False
#Si termina el ciclo es que todos fueron iguales
return True

'''Actualiza el mejor individuo encontrado con
el menor numero de aristas monocromaticas (NAM)'''
def ActualizaMejor(AMonocromaticas,poblacion,best,
best_ind,ind,termina):
# Guarda al individuo que obtuvo el menor numero
 de aristas monocromaticas
if AMonocromaticas[ind] < best:
best[0] = AMonocromaticas[ind]
best_ind[0] = poblacion[ind]
#return AMonocromaticas[ind], poblacion[ind]
if AMonocromaticas[ind]==0:
termina = True

'''Realiza una busqueda local en la poblacion
 inicial,calcula el numero de aristas monocromaticas
 y actualiza el mejor'''
def Busqueda_Local(G,AMonocromaticas,poblacion,
probabilidades,best,best_ind,termina):
# Realiza la busqueda local en la poblacion inicial
for ind in range(tam_poblacion):
# calcula el numero de aristas monocromaticas
 del individuo
AMonocromaticas.append(Busquedas_locales.
numeroAristasMono(G, poblacion[ind],numColores))

ActualizaMejor(AMonocromaticas,poblacion,best,best_ind,
ind,termina)

AMonocromaticas[ind] = BusquedaLocal(poblacion[ind],
 probabilidades[ind], AMonocromaticas[ind], G)

if AMonocromaticas[ind] == 0:
best = 0
best_ind = poblacion[ind]
return 1
else:
ActualizaMejor(AMonocromaticas, poblacion, best,
 best_ind, ind, termina)
if termina == True:
break
return 0

'''Si tres generaciones no encuentra nada mejor,
 el progama termina'''
def Termina_Generacion(bestAnt,AMonoNuevo,gen,NumGenIgual):
bestAnt.append(AMonoNuevo)
if (gen + 1) %NumGenIgual == 0:
if sonIguales(bestAnt, NumGenIgual):
bestAnt = []
return 0
bestAnt = []

'''Funcion encargada de imprimir al mejor individuo,
 el mejor numero de aristas monocromaticas,
  y el tiempo total del programa'''
def imprimeResultados(best,best_ind,start_time,archivo):
print('\n\nmejor numero de aristas monocromaticas: ')
print(str(best))
print('mejor individuo: ')
print(best_ind)
print("%s seconds" %(time.time() - start_time))
# bolsas.Muestra_Grafo(G)
# bolsas.colorea_grafo(G,best_ind)
# if best == 0:
archivo.write(str(time.time() - start_time)
+ '\t' + str(best[0]) + '\n')
# archivo.close()

#********** Algoritmo Genetico ***************
def main():
# ========================================
# === Definicion de variables y estructuras de datos ===
ind = 0
termina = False
NumGenIgual = 3
best = np.array([np.inf])
bestAnt = []
best_ind = [0]
poblacion = []
probabilidades=[]
AMonocromaticas = []
nuevos_individuos = []
AMonoNuevo = []
probabilidadeshijo = []
ind_padres = []
#Estructuras para los nuevos individuos
nuevos_individuos = []
AMonoNuevo = []
ind_padres = []
probabilidadeshijo = []
mitad_poblacion = int(tam_poblacion / 2)
random.seed(semilla)
# ========================================================
# === Inicia Algoritmo genetico ========================
start_time = time.time()
InicializacionPoblacion(poblacion, probabilidades, G,numNodos)
if Busqueda_Local(G, AMonocromaticas, poblacion, probabilidades,
 best, best_ind, termina):
termina = True
#Continua si arriba NO encontro un individuo con aristas
 monocromaticas igual a cero
if not termina:
for gen in range(numGeneraciones):
for p in range(mitad_poblacion):
Padres = EligePadres(poblacion, numNodos, ind_padres,
 probabilidadeshijo)
nuevos_individuos.append(CruzaPadres(Padres, probabilidadeshijo,
 AMonoNuevo, numNodos, G))
ActualizaMejor(AMonocromaticas, poblacion, best, best_ind,
 ind, termina)
for j in range(mitad_poblacion):
#realiza la busqueda local con el numero de aristas
 monocromaticas del nuevo individuo
AMonoNuevo[j] = BusquedaLocal(nuevos_individuos[j],
 probabilidadeshijo[j],AMonoNuevo[j], G)
#reemplaza al hijo con el peor individuo
ActualizaPoblacion(poblacion, nuevos_individuos[j],
 AMonocromaticas, AMonocromaticas[j], probabilidades,
  probabilidadeshijo[j], ind_padres[j])
ActualizaMejor(AMonocromaticas, poblacion, best,
 best_ind, ind, termina)
if Termina_Generacion(bestAnt, AMonoNuevo, gen, NumGenIgual) == 0:
break
imprimeResultados(best, best_ind, start_time,archivo)
# ============ Termina Algoritmo genetico ===============
```

5.4.2. bolsas.py

```
import networkx as nx
import pandas as pd
import matplotlib.pyplot as plt
import heapq
import random
import numpy as np
from heapq import heappush, heappop, _heapify_max

'''Crea un grafo aleatorio'''
def crea_grafo_aleatorio(numNodos, prob):
#Genera un grafo aleatorio con un numero de nodos
return nx.fast_gnp_random_graph(numNodos,prob)
#y una cierta probabilidad

'''Muestra grafo en una imagen'''
def Muestra_Grafo(G):
pos = nx.circular_layout(G, scale=5)
#Genera la posicion del grafo
#pos = nx.spring_layout(G, scale=5)
#Dibuja el grafo de acuerdo a sus parametros
nx.draw(G, pos, with_labels=True)
plt.draw()  #Crea el dibujo
plt.show()  #muestra el dibujo

'''Funcion que se encarga de leer el archivo
 (benchmark) y crear el grafo'''
def crea_grafo(namefile):
#abre el archivo
fh = open(namefile, 'rb')
#lee el archivo y toma los datos para
H = nx.read_edgelist(fh, nodetype=int)
#crear el grafo con datos enteros
fh.close()  #cierra el archivo
return H #regresa el grafo

'''Crea el heap con los nodos del mayor grado al menor'''
'''Genera una lista que contiene el id del nodo,
 su grado y su numero de clase'''
def crea_Heap(G, numColores,numNodos):
heap = []
#se multiplica por -1 debido a que con el maxheap
 queremos en minimo numero de clases
clase = numColores * -1
for nodo in range(1,numNodos+1):
if nodo in G.nodes:
grad = G.degree(nodo)
#agregamos la tupla(clase,grado,id,
 conjunto clases validas)
heappush(heap, (clase,grad,nodo, set()))
#al heap
else:
# agregamos la tupla(clase,grado,id,
 conjunto clases validas)
heappush(heap, (clase, 0 ,nodo, set()))
#hace un max heap
heapq._heapify_max(heap)
#regresa el heap
return heap

'''Muestra el contenido del heap'''
def ver_heap(heap):
print('Heap: ')
#Mientras el heap No este vacio saca
 uno a uno los elementos del tope
while heap:
print(heappop(heap))

'''Verifica si el nodo actual se encuentra dentro
 de la bolsa i'''
def es_compatible_bolsa(i, G, id, Bolsas):
#verifica que el diccionario (bolsa i) no este vacia
if bool(Bolsas[i]):
#crea la lista con las adyacencias del nodo actual
adj = G[id]
#recorre la lista de adyacencias
for nodo in adj:
#pregunta si el nodo adyacente se encuentra
 dentro de la bolsa i
if nodo in Bolsas[i]:
#si existe una adyacencia en la bolsa, retorna un uno
return 1
#si NO existe adyacencia, o la bolsa esta vacia
 retorna un cero
return 0

'''Disminuye el numero de clases de los nodos
 adyacentes al nodo actual'''
def disminuye_clases(G, heap, id, ind_Bolsa,grado):
if grado > 0:
# crea una lista con los nodos adyacentes al nodo actual
adyacentes = G[id]
# Recorre la lista de nodos adyacentes
for id2 in adyacentes:
# crea una lista con los ids dentro de la tupla
(como estan acomodados en heap)
ids = [ids[2] for ids in heap]
# pregunta si el id de adyacencias
 se encuentra de la lista de 'ids'
if id2 in ids:
# guarda la tupa con los valores
(del indice donde lo encontro)
temp = list(heap[ids.index(id2)])
# mueve el valor viejo al final del heap
heap[ids.index(id2)] = heap[-1]
heap.pop()  # lo elimina
if not ind_Bolsa in temp[3]:
# disminuye en uno la clase
temp[0] = temp[0] - 1
temp[3].add(ind_Bolsa)
# ingresa la tupla con los valores actualizados
heappush(heap, tuple(temp))

def cuenta_nodos(individuo, numColores, numNodos):
probabilidades=[]
probabilidades = {k:len(v)/numNodos for k,
 v in individuo.items()}
return probabilidades

###
def crea_individuo(G, numColores,numNodos):
#Es la lista que contiene el id, grado y clase
heap = []
#Es el diccionario que representan las bolsas de colores
Bolsas_colores = {}
#Es el indice de las bolsas de colores
i = 0

#print('Grafo: ')
#Muestra la lista de elementos del grafo
#print(list(G))
#crea una imagen del grafo con un solo color
#Muestra_Grafo(G)
#toma los datos del grafo y regresa la lista
heap = crea_Heap(G, numColores,numNodos)
#Crea un diccionario donde cada key es un color y el valor
Bolsas_colores = {k: {} for k in range(numColores)}
#recorre el heap hasta que este vacio
while heap:
#toma los datos del heap (el primer nodo)
clase,grado,id,conj = heappop(heap)
# Muestra los elementos del heap (envia copia del heap)
#ver_heap(heap[:])
#busca la bolsa que no contenga adyacencias
while i < numColores:
#if es_compatible_bolsa(i, G, id, Bolsas_colores):
if i in conj:
i = i + 1
else: break
#inserta el nodo actual en la bolsa
que NO contiene adyacencias
if i < numColores:
Bolsas_colores[i][id] = id
#El nodo contiene adyacencias en todas las bolsas
 (lo ingresa en una aleatoria)
else:
Bolsas_colores[random.randint(0, numColores-1)][id] = id
# disminuye en uno las clases adyacentes
disminuye_clases(G, heap, id, i, grado)
# Vuelve a ordenar el heap
#heapq._heapify_max(heap)
i = 0
return Bolsas_colores
```

5.4.3. Cruza.py

```
import random

'''Encuentra la bolsa con la maxima cardinalidad
 y regresa el indice donde la bolsa'''
def Max_cadinalidad(individuo):
#encuentra el indice donde encuentra la
 cardinalidad maxima
ind=max(individuo, key=lambda k: len(individuo[k]))
return ind

'''Elimina los nodos del nuevo individuo del
 individuo del cual NO obtuvo los nodos'''
def quita_bolsa(Bolsa, individuo, numColores):
#recorre la lista de nodos de la bolsa que se
 ingresaron en el nuevo individuo
for valor in list(Bolsa):
#recorre todas las bolsas del individuo
for i in range(numColores):
#pregunta si el nodo del nuevo individuo
 se encuentra en la bolsa i
if valor in individuo[i]:
#Si el valor se encuentra en la bolsa, lo elimina
del individuo[i][valor]
#y termina el ciclo
break

'''Recibe dos individuos, genera la cruza
 mediante una estrategia greedy de cardinalidad maxima
 y forma un nuevo individuo'''
def GPX(individuos, numColores):
nuevo_individuo = {}
#recorre las bolsas de colores
for l in range(numColores):
#toma a los padres por turnos, para que sea balanceado
if l%2 == 0:
#obtiene el indice de la bolsa con cardinalidad
 maxima del padre 1
ind = Max_cadinalidad(individuos[0])
#ingresa la bolsa al nuevo individuo
nuevo_individuo[l] = dict(individuos[0][ind])
#elimina los nodos en el padre 2
quita_bolsa(individuos[0][ind], individuos[1], numColores)
#elimina la bolsa del padre 1
individuos[0][ind].clear()
else:
#obtiene el indice de la bolsa con cardinalidad
 maxima del padre 2
ind = Max_cadinalidad(individuos[1])
#ingresa la bolsa al nuevo individuo
nuevo_individuo[l] = dict(individuos[1][ind])
#elimina los nodos en el padre 1
quita_bolsa(individuos[1][ind], individuos[0], numColores)
#elimina la bolsa del padre 2
individuos[1][ind].clear()
#Recorre nuevamente las bolsas de colores
for l in range(numColores):
#checa si la bolsa esta vacia
if individuos[0][l]:
#toma cada valor en la bolsa
for nodo in individuos[0][l].keys():
#Lo inserta en una bolsa aleatoria
nuevo_individuo[random.randint(0, numColores-1)][nodo]=nodo
#Retorna el nuevo individuo
return nuevo_individuo
```

5.4.4. Busquedas_locales.py

```
import networkx as nx
import numba
import math
import numpy as np
import random

T = 2.7
k = 1.38064852
iteraciones = 20
busqueda_vecindario = 1000

'''Mueve un nodo al azar a otra bolsa
 (se mueve en el vecindario), solo realiza
  el movimiento si genera un progreso'''
def vecino_escalando(G,individuo, probabilidades,
AristasMono, numColores):
for i in range(busqueda_vecindario):
while True:
#Elige una bolsa con probabilidad a su numero de nodos
bolsaProbabilistica = bolsaAleatoriaProbabilidad
(probabilidades, numColores)
#verifica que la bolsa que eligio no este vacia
if individuo[bolsaProbabilistica]:
break
#Toma el diccionario con los nodos de la bolsa elegida
dic = dict(individuo[bolsaProbabilistica])
#Elige un nodo al azar de la bolsa
nodo = random.choice(list(dic.keys()))
# calcula el numero de aristas monocromaticas
 del nodo en la bolsa elegida
monoAct = num_mono_bolsa(dic, nodo, G)
BolsaNueva = BolsaAleatoria(bolsaProbabilistica,
 numColores)
monopost = num_mono_bolsa(individuo[BolsaNueva],
 nodo, G)
if monopost != 0 and monopost < monoAct:
delta = monopost - monoAct
AristasMono = AristasMono + delta
# Elimina el nodo de la bolsa
del individuo[bolsaProbabilistica][nodo]
# Inserta el nodo en otra bolsa al azar
individuo[BolsaNueva][nodo] = nodo
if AristasMono == 0:
break
return AristasMono

'''Mueve un nodo al azar a otra bolsa
 (se mueve en el vecindario), no desecha malos
  movimientos si no que elige en base a una probabilidad
   si realiza o no el movimiento'''
def vecino_metropolis(G,individuo, probabilidades,
AristasMono, numColores):
for i in range(busqueda_vecindario):
while True:
#Elige una bolsa con probabilidad a su numero de nodos
bolsaProbabilistica = bolsaAleatoriaProbabilidad
(probabilidades, numColores)
#verifica que la bolsa que eligio no este vacia
if individuo[bolsaProbabilistica]:
break
#Toma el diccionario con los nodos de la bolsa elegida
dic = dict(individuo[bolsaProbabilistica])
#Elige un nodo al azar de la bolsa
nodo = random.choice(list(dic.keys()))
#calcula el numero de aristas monocromaticas del
 nodo en la bolsa elegida
monoAct = num_mono_bolsa(dic, nodo, G)
BolsaNueva = BolsaAleatoria
(bolsaProbabilistica, numColores)
monopost = num_mono_bolsa(individuo[BolsaNueva],
 nodo, G)
delta = monopost - monoAct
if probabilidadAceptar(delta):
AristasMono = AristasMono + delta
# Elimina el nodo de la bolsa
del individuo[bolsaProbabilistica][nodo]
# Inserta el nodo en otra bolsa al azar
individuo[BolsaNueva][nodo] = nodo
if AristasMono == 0:
break
return AristasMono

'''Formula de Boltzmann para determinar
 si acepta o no el cambio de vecindario'''
def probabilidadAceptar(delta):
if delta < 0:
return True
else:
P = math.exp(-delta/k*T)
# selecciona un numero al azar del cero al uno
r = random.uniform(0, 1)
if r < P:
return True
else:
return False

'''Elige una bolsa aleatoria que no sea
 la que ya eligio probabilisticamente'''
def BolsaAleatoria(bolsa,numColores):
while True:
#Elige una bolsa aleatoria
rand = random.randint(0, numColores - 1)
#si la bolsa es diferente a la elegida
if rand != bolsa:
#Regresa el indice de la bolsa
return rand

'''Elige una bolsa con probabilidad basada
 en el numero de nodos contenidos en la bolsa'''
def bolsaAleatoriaProbabilidad (probabilidades, numColores):
#selecciona un numero al azar del cero al uno
r = random.uniform(0, 1)
l = 0
#recorre hasta el numero de bolsas
for i in range(numColores):
#si cae entre l y la probabilidad de la bolsa i
if (r >= l and r < l + probabilidades[i]):
#retorna el indice i
return i
else:
#si no a la variable l le suma probabilidad de la bolsa i
l = l + probabilidades[i]

'''Calcula el numero de aristas monocromaticas de la bolsa,
 dado un nodo, una bolsa y el grafo'''
def num_mono_bolsa( bolsa, nodo, G):
#Empieza con el NAM en cero
num_mono = 0
if nodo in G.nodes:
#Toma el primer nodo en la bolsa
for i in bolsa:
if i in G.nodes:
#Pregunta si existe adyacencia entre el nodo mandado
 y el nodo de la bolsa
if G.has_edge(nodo, i) or G.has_edge(i, nodo):
#Aumenta en uno el NAM
num_mono = num_mono + 1
#Regresa el total de NAM
return num_mono

'''Calcula el numero monocromatico del individuo'''
def numeroAristasMono(G, individuo, numColores):
num_mono = 0
#Toma las bolsas del individuo
for i in range(numColores):
#Toma los nodos contenidos en la bolsa actual
nodos_bolsa = list(individuo[i])
#Toma en orden los nodos
for j in range(len(nodos_bolsa)-1):
r = j+1
#Toma el nodo siguiente al actual
while r < len(nodos_bolsa):
#Pregunta si hay una adyacencia de ambas formas
if G.has_edge(nodos_bolsa[j], nodos_bolsa[r])
 or G.has_edge(nodos_bolsa[r], nodos_bolsa[j]):
 #suma en uno las aristas monocromaticas
num_mono = num_mono + 1
#Toma el nodo que sigue
r = r+1
#regresa el total de NAM
return num_mono

'''Recibe el grafo para obtener las adyacencias
 de los nodos, el individuo que contiene la
 informacion, las probabilidades que contiene la
 probabilidad de las bolsas de acuerdo al numero
 de nodos contenidos, Las aristas que contiene
 el numero de aristas monocromaticas del individuo,
 y el numero de colores para recorrer las bolsas'''
def Busqueda_Metropolis(G, individuo, probabilidades,
Aristas, numColores):
#Si el ultimo evaluado es diferente de cero continua
 con la busqueda local
if Aristas != 0:
#Realiza la busqueda Metropolis
AristasDesp = vecino_metropolis(G, individuo, probabilidades,
Aristas, numColores)
return individuo, AristasDesp
else:
return individuo, Aristas
```

```
'''Recibe el grafo para obtener las adyacencias
 de los nodos, el individuo que contiene la informacion,
 las probabilidades que contiene la probabilidad de las
 bolsas de acuerdo al numero de nodos contenidos, las
 aristas que contiene el numero de aristas
 monocromaticas del individuo, y el numero de
 colores para recorrer las bolsas'''
def Busqueda_Escalando(G, individuo, probabilidades,
 Aristas, numColores):
# Si el ultimo evaluado es diferente de cero continua
 con la busqueda local
if Aristas != 0:
AristasDesp = vecino_escalando(G, individuo, probabilidades,
Aristas, numColores)
return individuo, AristasDesp
else:
return individuo, Aristas
```

5.5. Algoritmo genético híbrido paralelo

El algoritmo genético híbrido paralelo contiene las mismas clases que el secuencial, como la cruza y bolsas, con la diferencia en de que agrega y cambia algunas configuraciones para utilizar las bibliotecas de *numba*, en las funciones *main* y búsquedas locales, como se muestra a continuación.

5.5.1. Main

```
from numba import *
from numba import cuda, vectorize
from numba.cuda.random import create_xoroshiro128p_states
import numpy as np
import networkx as nx
import scipy as sp
import bolsas
import Cruza
import Busquedas_locales
import random
import copy
import math
import time
import winsound

#Datos Algoritmo Genetico
tam_poblacion = 128
numGeneraciones = 20
numColores = 2
semilla = 1

metropolis_gpu = cuda.jit(device=True)(Busquedas_locales.
Busqueda_MetropolisCUDA)
Escalando_gpu = cuda.jit(device=True)(Busquedas_locales.
Busqueda_EscalandoCUDA)
```

Llamada del kernel utilizando memoria global

```
'''Manda a llamar a la busqueda local en CUDA(metropolis
 o Escalando la colina)'''
@cuda.jit
def BusquedaLocal_CUDA(nuevo_individuos, probabilidades,
Aristmono,MatrizAdjacencia, H,numNodos,tam_pobla,
rng_states):
bx = cuda.blockIdx.x
thx = cuda.threadIdx.x
bw = cuda.blockDim.x
id = thx + bx * bw
```

Llamada del kernel utilizando memoria compartida

```
@cuda.jit
def BusquedaLocal_CUDA(nuevo_individuos, probabilidades,
Aristmono,MatrizAdjacencia, H,numNodos,tam_pobla,
rng_states):
bx = cuda.blockIdx.x
thx = cuda.threadIdx.x
bw = cuda.blockDim.x
id = thx + bx * bw

individuoCuda = cuda.shared.array(shape= (numColores,
NODOS),dtype= int32)
probabilidadCuda = cuda.shared.array(shape = numColores,
 dtype = float32)
MatrizAdjCuda = cuda.shared.array(shape=(NODOS,NODOS),
 dtype=int32)

if id < tam_pobla:
individuoCuda = nuevo_individuos[id]
probabilidadCuda = probabilidades[id]
AristasCuda = Aristmono[id]
MatrizAdjCuda= MatrizAdjacencia
# regresa al individuo despues de realzar la busqueda
local
#metropolis_gpu(MatrizAdjCuda, individuoCuda,
 probabilidadCuda,AristasCuda,numColores,numNodos,
 rng_states,id)
# regresa al individuo despues de realzar la busqueda
 local
Escalando_gpu(MatrizAdjCuda, individuoCuda,
 probabilidadCuda,AristasCuda,numColores,numNodos,
 rng_states,id)

for i in range(numColores):
for j in range(NODOS):
nuevo_individuos[id,i,j]=individuoCuda[i,j]
Aristmono[id] = AristasCuda
cuda.syncthreads()
```

Llamada del kernel utilizando memoria local

```
@cuda.jit
def BusquedaLocal_CUDA(nuevo_individuos, probabilidades,
Aristmono,MatrizAdjacencia, H,numNodos,tam_pobla,
rng_states):
bx = cuda.blockIdx.x
thx = cuda.threadIdx.x
bw = cuda.blockDim.x
id = thx + bx * bw

individuoCuda = cuda.local.array(shape= (numColores,
 NODOS),dtype= int32)
probabilidadCuda = cuda.local.array(shape = numColores,
 dtype = float32)
MatrizAdjCuda = cuda.local.array(shape=(NODOS,NODOS),
 dtype=int32)

if id < tam_pobla:
individuoCuda = nuevo_individuos[id]
probabilidadCuda = probabilidades[id]
AristasCuda = Aristmono[id]
MatrizAdjCuda= MatrizAdjacencia
# regresa al individuo despues de realzar la busqueda
 local
#metropolis_gpu(MatrizAdjCuda, individuoCuda,
probabilidadCuda,AristasCuda,numColores,numNodos,
rng_states,id)
#regresa al individuo despues de realzar la busqueda
 local
Escalando_gpu(MatrizAdjCuda, individuoCuda,
 probabilidadCuda,AristasCuda, numColores, numNodos,
  rng_states,id)

for i in range(numColores):
for j in range(NODOS):
nuevo_individuos[id,i,j]=individuoCuda[i,j]
Aristmono[id] = AristasCuda
cuda.syncthreads()
# regresa al individuo despues de realzar la busqueda
 local
metropolis_gpu(MatrizAdjacencia,nuevo_individuos[id],
 probabilidades[id], Aristmono[id], numColores,
 numNodos,rng_states,id)
# regresa al individuo despues de realzar la busqueda
 local
#Escalando_gpu(MatrizAdjacencia,nuevo_individuos[id],
 probabilidades[id], Aristmono[id], numColores,
 numNodos,rng_states,id)
cuda.syncthreads()
```

```
'''Convierte el hijo en un diccionario de colores'''
def convert_hijo(Arr,numNodos):
Bolsas_colores = {k: {} for k in range(numColores)}
for i in range(numColores):
for j in range(numNodos):
if Arr[i][j] == 1:
Bolsas_colores[i][j+1] = j+1
return Bolsas_colores

'''Actualiza la poblacion por medio del arreglo numpy
 utilizado para CUDA'''
def ActualizaPoblacionCUDA(poblacion,hijo,AristasMono,
 AristasMonoHijo,probabilidades,probabilidadHijo,indicesP,
 numNodos):
#Si el padre 2 tiene mayor numero de aristas mono que el
 padre 1 entonces...
if AristasMono[indicesP[0]] < AristasMono[indicesP[1]]:
#El hijo reemplaza al padre 2
poblacion[indicesP[1]]= convert_hijo(hijo,numNodos)
#actualiza las aristas mono del padre reemplazado
 con las del hijo
AristasMono[indicesP[1]]= AristasMonoHijo
probabilidades[indicesP[1]]= probabilidadHijo
#Si no...
else:
#El hijo reemplaza al padre 1
poblacion[indicesP[0]] = convert_hijo(hijo,numNodos)
#actualiza las aristas mono del padre reemplazado
 con las del hijo
AristasMono[indicesP[0]] = AristasMonoHijo
probabilidades[indicesP[0]] = probabilidadHijo

'''convierte de la estructura en python a
 un arreglo numpy
 para poder ser utilizado en CUDA'''
def convierte_individuonumpy(individuo,probabilidades,
Amono,indi_np,prob_np,AmonoNp,num_indiv,bolsas):
for i in range(num_indiv):
AmonoNp[i] = Amono[i]
for j in range(bolsas):
prob_np[i][j]= probabilidades[i][j]
for r in individuo[i][j]:
indi_np[i][j][r-1] = 1

def main():
#=== Definicion de variables y estructuras de datos ===
ind = 0
mitad_poblacion = int(tam_poblacion / 2)
termina = False
NumGenIgual = 3
best= np.array([np.inf])
bestAnt = []
best_ind = [0]
poblacion = []
probabilidades=[]
AMonocromaticas = []
nuevos_individuos = []
AMonoNuevo = []
probabilidadeshijo = []
ind_padres = []

#Estructuras de los nuevos individuos
nuevos_individuos = []
AMonoNuevo = []
ind_padres = []
probabilidadeshijo = []
#Lista que contendra a los dos padres a elegidos
Padres = []
# ================================
#=== Definicion de variables para CUDA ===
Matriz_Adyacencia = nx.to_numpy_matrix(G,
nodelist=sorted(G.nodes()), dtype=int)
nuevos_individuosNp = np.zeros((mitad_poblacion,
 numColores, numNodos), dtype=np.int32)
proba_hijoNp = np.zeros((mitad_poblacion, numColores),
 dtype=np.float32)
AMonoNuevoNp = np.zeros((mitad_poblacion),
dtype=np.int32)
TotalHilos = bloques * hilos
rng_states = create_xoroshiro128p_states
((bloques * hilos)* bloques, seed=semilla)
# ================================
#=== Inicia Algoritmo genetico ===
start_time = time.time()
InicializacionPoblacion(poblacion, probabilidades,
 G, numNodos)
if Busqueda_Local(G, AMonocromaticas, poblacion,
 probabilidades, best, best_ind, termina):
termina = True
#Continua si NO encontro un individuo con aristas
 monocromaticas igual a cero
if not termina:
for gen in range(numGeneraciones):
for p in range(mitad_poblacion):
Padres = EligePadres(poblacion,numNodos,ind_padres,
probabilidadeshijo)
nuevos_individuos.append(CruzaPadres(Padres,
probabilidadeshijo,AMonoNuevo, numNodos, G))
ActualizaMejor(AMonocromaticas, poblacion, best,
 best_ind,ind, termina)
convierte_individuonumpy(nuevos_individuos,
 probabilidadeshijo, AMonoNuevo, nuevos_individuosNp,
 proba_hijoNp,AMonoNuevoNp,mitad_poblacion, numColores)

#Envia al dispositivo CUDA
d_individuo = cuda.to_device(nuevos_individuosNp)
d_matriz = cuda.to_device(Matriz_Adyacencia)
d_prob = cuda.to_device(proba_hijoNp)
d_AMono = cuda.to_device(AMonoNuevoNp)

# realiza la busqueda local en CUDA
BusquedaLocal_CUDA[bloques, hilos](d_individuo, d_prob,
 d_AMono, d_matriz, hilos, numNodos, mitad_poblacion,
 rng_states)

#recibe en CPU
h_individuos = d_individuo.to_host()
h_matriz = d_matriz.to_host()
h_prob = d_prob.to_host()
h_AMono = d_AMono.to_host()

for j in range(mitad_poblacion):
# reemplaza al hijo con el peor individuo
ActualizaPoblacionCUDA(poblacion, nuevos_individuosNp[j],
 AMonocromaticas, AMonoNuevoNp[j],probabilidades,
 proba_hijoNp,ind_padres[j],numNodos)
ActualizaMejor(AMonocromaticas, poblacion, best,
 best_ind, ind, termina)
if Termina_Generacion(bestAnt,AMonoNuevo,gen,
 NumGenIgual) == 0:
break
imprimeResultados(best,best_ind,start_time,archivo)
#=== Termina Algoritmo genetico ===
```

5.5.2. Busquedas_locales_paralelas.py

```
import networkx as nx
import numba
from numba import cuda, prange
from numba.cuda.random import
 xoroshiro128p_uniform_float32
import math
import numpy as np
import random


T = 2.7
k = 1.38064852
iteraciones = 20
busqueda_vecindario = 1000

'''Formula de Boltzmann para determinar si acepta o no
 el cambio de vecindario'''
@numba.jit()
def probabilidadAceptarCUDA(delta, rng_states, id):
if delta < 0:
return True
else:
P = math.exp(-delta/k*T)
# selecciona un numero al azar del cero al uno
r = xoroshiro128p_uniform_float32(rng_states, id)
if r < P:
return True
else:
return False

'''Elige una bolsa aleatoria que no sea la que ya
 eligio probabilisticamente'''
@numba.jit()
def BolsaAleatoriaCUDA(bolsa,numColores,rng_states,id):
while True:
#Elige una bolsa aleatoria
rand = int(xoroshiro128p_uniform_float32(rng_states, id)*
numColores)
#si la bolsa es diferente a la elegida
if rand != bolsa:
#Regresa el indice de la bolsa
return rand

'''Elige una bolsa con probabilidad basada en el numero
 de nodos contenidos en la bolsa'''
@numba.jit()
def bolsaAleatoriaProbabilidadCUDA (probabilidades,
 numColores,rng_states,id):
#selecciona un numero al azar del cero al uno
r = xoroshiro128p_uniform_float32(rng_states, id)
l = 0
#recorre hasta el numero de bolsas
for i in range(numColores):
#si cae entre l y la probabilidad de la bolsa i
if (r >= l and r < l + probabilidades[i]):
return i    #retorna el indice i
else:
#si no a la variable l le suma probabilidad de la bolsa i
l = l + probabilidades[i]
return i

'''Calcula el numero de aristas monocromaticas de la
 bolsa, dado un nodo, una bolsa y
 la matriz de adyacencia'''
@numba.jit()
def num_mono_bolsaCUDA( bolsa, nodo, M, numNodos):
#Empieza con el NAM en cero
num_mono = 0
#Toma el primer nodo en la bolsa
for i in range(numNodos):
#Si el nodo se encuentra en la bolsa
if bolsa[i] == 1:
#Pregunta si existe adyacencia entre el nodo mandado
 y el nodo de la bolsa
if M[nodo, i] == 1 or M[i, nodo] == 1:
#Aumenta en uno el NAM
num_mono = num_mono + 1
#Regresa el total de NAM
return num_mono

'''Verifica si la bolsa elegida esta vacia o no, toma
 la bolsa y recorre la busqueda sobre todos los nodos,
 si en todos es cero,entonces la bolsa esta vacia'''
@numba.jit()
def es_vacia(Bolsa,numNodos):
sum = 0
for i in range(numNodos):
sum += Bolsa[i]
return sum

bolsaProbabilidad_gpu = cuda.jit(device=True)
 (bolsaAleatoriaProbabilidadCUDA)
esVacia_gpu = cuda.jit(device=True)(es_vacia)
NAMBolsa_gpu = cuda.jit(device=True)(num_mono_bolsaCUDA)
bolsaAleatoria_gpu = cuda.jit(device=True)
 (BolsaAleatoriaCUDA)
probAcepta_gpu = cuda.jit(device=True)
 (probabilidadAceptarCUDA)

@numba.jit()
def Busqueda_MetropolisCUDA(M,individuo,probabilidades,
 AristMono,numColores,numNodos,rng_states,id):
if AristMono != 0:
for i in prange(busqueda_vecindario):
r = 0
vacia = 0
while vacia == 0:
# Elige una bolsa con probabilidad a su numero de nodos
bolsaProbabilistica = int(bolsaProbabilidad_gpu
(probabilidades,numColores,rng_states,id))
#verifica que la bolsa no este vacia
vacia = esVacia_gpu(individuo[bolsaProbabilistica],
numNodos)
#Selecciona un nodo al azar
while r == 0:
#selecciona un numero al azar del cero al numero de nodos
nodo = int(xoroshiro128p_uniform_float32(rng_states, id)
*numNodos)
r = individuo[bolsaProbabilistica][nodo]
# calcula el numero de aristas monocromaticas del nodo
 en la bolsa elegida
monoAct = NAMBolsa_gpu(individuo[bolsaProbabilistica],
 nodo, M,numNodos)
BolsaNueva = bolsaAleatoria_gpu(bolsaProbabilistica,
 numColores,rng_states,id)
monopost = NAMBolsa_gpu(individuo[BolsaNueva],
 nodo, M,numNodos)
delta = monopost - monoAct
if probAcepta_gpu(delta,rng_states,id):
AristMono = AristMono + delta
# Elimina el nodo de la bolsa
individuo[bolsaProbabilistica][nodo] = 0
# Inserta el nodo en otra bolsa al azar
individuo[BolsaNueva][nodo] = 1
if AristMono == 0:
break

@numba.jit()
def Busqueda_EscalandoCUDA(M,individuo,probabilidades,
AristMono,numColores,numNodos,rng_states,id):
if AristMono != 0:
for i in prange(busqueda_vecindario):
r = 0
vacia = 0
while vacia == 0:
# Elige una bolsa con probabilidad a su numero de nodos
bolsaProbabilistica = int(bolsaProbabilidad_gpu
(probabilidades,numColores,rng_states,id))
#verifica que la bolsa no este vacia
vacia = esVacia_gpu(individuo[bolsaProbabilistica],
numNodos)
#Selecciona un nodo al azar
while r == 0:
#selecciona un numero al azar del cero al numero de nodos
nodo = int(xoroshiro128p_uniform_float32(rng_states, id)
*numNodos)
r = individuo[bolsaProbabilistica][nodo]
# calcula el numero de aristas monocromaticas del nodo
 en la bolsa elegida
monoAct = NAMBolsa_gpu(individuo[bolsaProbabilistica],
 nodo, M,numNodos)
BolsaNueva = bolsaAleatoria_gpu(bolsaProbabilistica,
 numColores,rng_states,id)
monopost = NAMBolsa_gpu(individuo[BolsaNueva], nodo,
 M,numNodos)
delta = monopost - monoAct
if monopost > monoAct:
AristMono = AristMono + delta
# Elimina el nodo de la bolsa
individuo[bolsaProbabilistica][nodo] = 0
# Inserta el nodo en otra bolsa al azar
individuo[BolsaNueva][nodo] = 1
if AristMono == 0:
break

'''Recibe el grafo para obtener las adyacencias de los
 nodos, el individuo que contiene la informacion, las
 probabilidades que contiene la probabilidad de las
 bolsas de acuerdo al numero de nodos contenidos, Las
 aristas que contiene el numero de aristas monocromaticas
```

```
 del individuo, y el numero de colores para recorrer las
 bolsas'''
def Busqueda_Metropolis(G, individuo, probabilidades,
 Aristas, numColores):
#Si el ultimo evaluado es diferente de cero continua
 con la busqueda local
if Aristas != 0:
#Realiza la busqueda Metropolis
AristasDesp = vecino_metropolis(G, individuo,
 probabilidades,Aristas, numColores)
return individuo, AristasDesp
else:
return individuo, Aristas

'''Recibe el grafo para obtener las adyacencias de los
 nodos, el individuo que contiene la informacion,
 las probabilidades que contiene la probabilidad de
 las bolsas de acuerdo al numero de nodos contenidos,
 Las aristas que contiene el numero de aristas
 monocromaticas del individuo, y el numero de colores
 para recorrer las bolsas'''
def Busqueda_Escalando(G, individuo, probabilidades,
 Aristas, numColores):
# Si el ultimo evaluado es diferente de cero continua
 con la busqueda local
if Aristas != 0:
AristasDesp = vecino_escalando(G, individuo,
 probabilidades,Aristas, numColores)
return individuo, AristasDesp
else:
return individuo, Aristas
```

Bibliografía

[1] Karp, Richard M, et al. "Reducibility among combinatorial problems, June 2016", *Complexity of computer computations*, Springer, pp. 85–103, 1972.

[2] Blum, C., & Roli, A. (2003)."Metaheuristics in combinatorial optimization: Overview and conceptual comparison", *ACM Computing Surveys (CSUR)*,35(3), 268-308.

[3] Cormen, T. H. (2009). *Introduction to algorithms. MIT press.*

[4] Kleinberg, J., & Tardos, E. (2006). "Algorithm design", *Pearson Education*,2014.

[5] Beasley, D., Bull, D. R., & Martin, R. R. (1993). "An overview of genetic algorithms: Part 1, fundamentals", *University computing*,15(2), 56-69.

[6] Mühlenbein, H. (1989, July). "Parallel genetic algorithms, population genetics and combinatorial optimization", *In Workshop on Parallel Processing: Logic, Organization, and Technology*,(pp. 398-406). Springer, Berlin, Heidelberg.

[7] Reinhard Diestel. "Graph theory", *Graduate texts in mathematics*, springer, 5th edición, 2017.

[8] Puchinger, J., & Raidl, G. R. (2005, June). "Combining metaheuristics and exact algorithms in combinatorial optimization: A survey and classification. In International Work-Conference on the Interplay Between Natural and Artificial Computation", (pp. 41-53). Springer, Berlin, Heidelberg.

[9] Galinier, P., & Hao, J. K. (1999). "Hybrid evolutionary algorithms for graph coloring". *Journal of combinatorial optimization*, 3(4), 379-397.

[10] Dorigo, M.,& Stützle, T. (2010). "Ant colony optimization: overview and recent advances. In Handbook of metaheuristics". (pp. 227-263), *Springer*, Boston, MA.

[11] Voß, S., Martello, S., Osman, I. H., & Roucairol, C. (Eds.). (2012). "Meta-heuristics: Advances and trends in local search paradigms for optimization". Springer Science & Business Media.

[12] Glover, F., Kelly, J. P., & Laguna, M. (1995). "Genetic algorithms and tabu search: hybrids for optimization". *Computers & Operations Research*, 22(1), 111-134.

[13] Pardalos, P. M., Mavridou, T., & Xue, J. (1998). "The graph coloring problem: A bibliographic survey. In Handbook of combinatorial optimization". (pp. 1077-1141), *Springer*, Boston, MA.

[14] Chiarandini, M., & Stützle, T. (2002, September). "An application of iterated local search to graph coloring problem". *In Proceedings of the Computational Symposium on Graph Coloring and its Generalizations*, (pp. 112-125).

[15] Jensen, T. R., & Toft, B. (2011). "Graph coloring problems (Vol. 39)". *John Wiley & Sons.*

[16] Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., ... & Wessel, D. (2009). "A view of the parallel computing landscape". *Communications of the ACM*, 52(10), 56-67.

[17] Goldberg, D. E., & Holland, J. H. (1988). "Genetic algorithms and machine learning. Machine learning", 3(2), 95-99.

[18] Krasnogor, N., & Smith, J. (2005). "A tutorial for competent memetic algorithms: model, taxonomy, and design issues". *IEEE Transactions on Evolutionary Computation*, 9(5), 474-488.

[19] Hennessy, J. L., & Patterson, D. A. (2011). "Computer architecture: a quantitative approach". *Elsevier*.

[20] Leighton, Frank Thomson. "A graph coloring algorithm for large scheduling problems". *Journal of research of the national bureau of standards*, 84.6 (1979), 489-506.

[21] Gamache, Michel, Alain Hertz, and Jérôme Olivier Ouellet. "A graph coloring model for a feasibility problem in monthly crew scheduling with preferential bidding". *Computers and operations research*, 34.8, (2007), 2384-2395.

[22] Chaitin, Gregory J. "Register allocation and spilling via graph coloring.". *ACM Sigplan Notices*, Vol. 17, No. 6, ACM, 1982.

[23] Conte, Donatello, et al. "Thirty years of graph matching in pattern recognition.". *International journal of pattern recognition and artificial intelligence*. 18.03, (2004), 265-298.

[24] Zhu, Xudong, Linglong Dai, and Zhaocheng Wang. "Graph coloring based pilot allocation to mitigate pilot contamination for multi-cell massive MIMO systems.". *IEEE Communications Letters*, 19.10, (2015), 1842-1845.

[25] Checco, Alessandro, and Doug J. Leith. "Fast, Responsive Decentralized Graph Coloring.". *IEEE/ACM Transactions on Networking*, 25.6, (2017), 3628-3640.

[26] Alsmirat, Mohammad A., et al. "Accelerating compute intensive medical imaging segmentation algorithms using hybrid CPU-GPU implementations.". *Multimedia Tools and Applications*, 76.3, (2017), 3537-3555.

[27] Ji, Yimu, et al. "Design and Development of Scientific Computing-Oriented Cloud Service Based on CUDA.". *LISS 2013. Springer, Berlin, Heidelberg*, 2015, 197-201.

[28] Abadi, Martín, et al. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems." *arXiv preprint arXiv:1603.04467*, (2016).

[29] Nickolls, J., Buck, I., Garland, M., & Skadron, K. (2008)."Scalable parallel programming with CUDA". Queue, 6(2), 40-53.

[30] Cheng, M. Grossman and T. McKercher. "Professional CUDA C programming", *Indianapolis: John Wiley and Sons*, 2014.

[31] Nickolls, Jhon and Buck Ian (2008). "Scalable Parallel Programing with CUDA".

[32] Procesamiento paralelo CUDA . "Qué es CUDA-NVIDIA", Nvidia.es,2017,[Online]Available:http://www.nvidia.es/object/cuda-parallel-computing-es.html.

[33] Rao, S. S. (2009). Engineering optimization: theory and practice. John Wiley & Sons.

[34] cuda2012programming. "Programming guide", *Cuda, C*,2012.

[35] J. sanders y E. Kandrot. "CUDA by Example an introduction to general-purpose GPU programming", *Boston: Addison-Wesley*, 2011.

[36] Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., & Hwu, W. M. W. (2008, February). "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA". *In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, (pp. 73-82), ACM.

[37] Yang, C. T., Chang, T. C., Wang, H. Y., Chu, W. C., & Chang, C. H. (2011, May). "Performance comparison with OpenMP parallelization for multi-core systems. In Parallel and Distributed Processing with Applications (ISPA)", *2011 IEEE 9th International Symposium on (pp. 232-237). IEEE.*

[38] Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., ... & Wessel, D. (2009). "A view of the parallel computing landscape.". *Communications of the ACM*, 52(10), 56-67.

[39] Trudeau, R. J. (2013). "Introduction to graph theory". *Courier Corporation.*

[40] Jensen, T. R., & Toft, B. (2011). "Graph coloring problems (Vol. 39)". *John Wiley & Sons.*

[41] Eiben, A. E., & Smith, J. E. (2003). "Introduction to evolutionary computing (Vol. 53)". *Heidelberg: springer.*

[42] Coley, D. A. (1999). "An introduction to genetic algorithms for scientists and engineers". *World Scientific Publishing Company.*

[43] Thomas Bäck. Optimal Mutation Rates in Genetic Search. In Stephanie Forrest, editor, Proceedings of the Fifth International Conference on Genetic Algorithms, pages 2–8. Morgan Kaufmann Publishers, San Mateo, California, July 1993.

[44] Terence C. Fogarty. Varying the Probability of Mutation in the Genetic Algorithm. In J. David Schaffer, editor, Proceedings of the Third International Conference on Genetic Algorithms, pages 104–109, San Mateo, California, 1989. Morgan Kaufmann Publishers.

[45] Neumann, F., & Witt, C. (2010, september). "Bioinspired Computation in Combinatorial Optimization", *Natural Computing Series.*

[46] Hochbaum, D. S. (1997). "Approximation algorithms for NP-hard problems", PWS Pub. Co., Boston.

[47] Glover F., Laguna M., Marti R.(2003) "Scatter Search and Path Relinking: Advances and Applications. In: Glover F., Kochenberger G.A. (eds) Handbook of Metaheuristics. International Series in Operations Research & Management Science"., vol 57.*Springer*, Boston, MA.

[48] Yu, Xinjie, & Gen, Mitsuo (2010). "Introduction to evolutionary algorithms". *Springer Science & Business Media.*

[49] Brown, J. R. (1972). "Chromatic scheduling and the chromatic number problem". *Management Science*, 19(4-part-1), 456-463.

[50] C. S. Adjiman, C. A. Schweiger, C. A. Floudas (auth.), Ding-Zhu Du, Panos M. Pardalos (1999). "Handbook of Combinatorial Optimization". *Springer US*, Volume 1–3.

[51] Brélaz, D. (1979). "New methods to color the vertices of a graph". *Communications of the ACM*, 22(4), 251-256.

[52] Welsh, D. J., & Powell, M. B. (1967). "An upper bound for the chromatic number of a graph and its application to timetabling problems". *The Computer Journal*, 10(1), 85-86.

[53] Matula, D. W., Marble, G., & Isaacson, J. D. (1972)."Graph coloring algorithms". *In Graph theory and computing* (pp. 109-122).

[54] D.S. Johnson, C.R. Aragon, L.A. McGeoch, and C. Schevon (1991). "Optimization by Simulated Annealing: An Experimental Evaluation, part II, Graph Coloring and Number Partitioning". *Operations Research*, 39(3) pp. 378-406.

[55] J.C. Culberson (1992). "Iterated Greedy Graph Coloring and the Difficulty Landscape". *Technical Report TR 92-07*, University of Alberta Dept of Computer Science, Edmonton, Alberta Canada T6G 2H1, ftp: ftp.cs. ualberta. ca/pub /TechReports.

[56] C. Avanthay, A. Hertz, and N. Zufferey (2003). "A Variable Neighborhood Search for Graph Coloring". *Eur. J. Oper.* Res. Vol. 151, pp. 379–388.

[57] González-Velarde, J. L. and M. Laguna (2002). "Tabu search with simple ejection chains for coloring graphs". *Ann. Oper.* Res. Vol. 117, pp. 165-174.

[58] Baghel, M., Agrawal, S., & Silakari, S. (2012). "Survey of metaheuristic algorithms for combinatorial optimization". *International Journal of Computer Applications*, 58(19).

[59] M. Chams, A. Hertz, and D. de Werra. (1987). "Some Experiments with Simulated Annealing for Coloring Graphs". *European Journal of Operational Research*, 32(2) (1987) pp. 260-266.

[60] Salari, E., & Eshghi, K. (2008). "An ACO algorithm for the graph coloring problem". *Int. J. Contemp. Math. Sciences*, 3(6), 293-304.

[61] Lü, Z., & Hao, J. K. (2010). "A memetic algorithm for graph coloring". *European Journal of Operational Research*, 203(1), 241-250.

[62] Lewandowski, G., & Condon, A. (1994). "Experiments with parallel graph coloring heuristics and applications of graph coloring". *DIMACS Series in Discrete Mathematics.*

[63] Łukasik, S., Kokosiński, Z., & Świetoń, G. (2007, September). "Parallel simulated annealing algorithm for graph coloring problem. In International Conference on Parallel Processing and Applied Mathematics". *Springer*, Berlin, Heidelberg. (pp. 229-238).

[64] Analytics, C. (2019). "Numba for CUDA GPUs — Numba 0.42.0-py3.6-macosx-10.7-x86_64.egg documentation.". *[online] Numba.pydata.org.*, Disponible en: http://numba.pydata.org/numba-doc/latest/cuda/index.html.