Diseño de un algoritmo memético para el problema de coloración del vértice

María Camila Oliveros Cala y Aylin Brigett Amairany Suárez Quiroz

Trabajo de grado para optar al título de ingeniero industrial

Director

Henry Lamos Díaz

Ph.D en Física-matemática

Universidad Industrial de Santander

Facultad de Ingenierías Físico-mecánicas

Escuela de Estudios Industriales y Empresariales

Bucaramanga

2022

Agradecimientos

A Dios por guiar cada uno de nuestros pasos y por darnos esa fortaleza para seguir adelante.

A nuestras mamás Marina Cala Cala y Ruth Quiroz Rey por su apoyo incondicional, por la confianza, por su sacrificio y esfuerzo, por estar siempre a pesar de las adversidades y por brindarnos su cariño y amor, logrando que este sueño se haga realidad.

A nuestras familias, por su comprensión y apoyo.

A nuestros compañeros y amigos más cercanos que estuvieron a nuestro lado en todo momento.

A nuestra alma máter la UIS por la formación recibida durante estos años tanto a nivel profesional como personal.

Al profesor Henry Lamos por orientarnos y ser paciente.

Al grupo de investigación ÓPALO por la oportunidad de desarrollar este proyecto.

Tabla de Contenido

Pág.

Introducción 13
1. Planteamiento del problema 16
2. Objetivos 18
2.1 Objetivo general 18
2.2 Objetivos específicos 18
3. Revisión de la literatura 19
3.1 Análisis bibliométrico 19
3.2 Análisis preliminar de la literatura 23
4. Marco teórico 35
4.1 Optimización combinatoria 35
4.2 Complejidad computacional 37
4.2.1 Clases de complejidad 37
4.2.1.1 P (Polinomial). 37
4.2.1.2 NP. 37
4.2.1.3 NP Complete (NP Completo). 37
4.2.1.4 NP Hard (NP difícil o complejo). 37
4.3 Teoría de grafos 38
4.3.1 Definiciones básicas 38
4.3.1.1 Grafo. 38
4.3.1.2 Grafo dirigido 39
4.3.1.3 Vértices adyacentes 39

4.3.1.4 Grado de un vértice. 39

4.4 Problema de coloración del vértice 40

4.4.1 Definiciones 40

4.4.1.1 Número cromático $\chi(G)$. 40

4.4.1.2 Coloración válida o legal. 40

4.4.1.3 Vértices en conflicto. 40

4.4.1.4 Coloración completa. 41

4.4.1.5 Conjunto independiente o estable. 41

4.4.1.6 Clase de color. 41

4.4.1.7 Grado cromático de un vértice. 41

4.4.1.8 Camarilla de un grafo. 41

4.5 Aplicaciones del problema de coloración del vértice 41

4.5.1 Problema de programación de cursos (CTP) 41

4.5.2 Problema de programación de exámenes (ETP) 42

4.5.3 Problema de programación de Job-Shop 42

4.6 Métodos de solución para el problema de coloración del vértice 42

4.6.1 Algoritmos exactos 43

4.6.1.1 Branch & Bound. 43

4.6.1.2 Branch & Cut. 43

4.6.2 Algoritmos aproximados 44

4.6.2.1 Heurísticas 44

4.6.2.1.1 Algoritmo LF (“largest first”). 45

4.6.2.1.2 Algoritmo SL (“smallest last”). 45

4.6.2.1.3 Algoritmo ID (“incidence degree”). ... 45

4.6.2.1.4 Algoritmo DSATUR (“saturation degree”). ... 45

4.6.2.1.5 Algoritmo MIS (“maximal independent sets”). ... 45

4.6.2.1.6 Algoritmo RLF (“recursive largest first”). ... 46

4.6.2.2 Metaheurísticas. ... 46

4.6.2.2.1 Búsqueda local. ... 46

4.6.2.2.2 Algoritmos evolutivos. ... 48

4.6.2.2.3 Algoritmos genéticos. ... 49

4.6.2.2.4 Colonia de hormigas. ... 50

4.6.2.2.5 Algoritmo memético. ... 51

5. Formulación del modelo matemático ... 52

5.1 Subíndices ... 53

5.2 Parámetros ... 53

5.3 Variables ... 53

5.4 Función objetivo ... 54

5.5 Restricciones ... 54

6. Construcción del algoritmo ... 55

6.1 Algoritmos genéticos ... 57

6.2 Restricciones ... 59

6.2.1 R1: Un profesor no puede dictar dos cursos al tiempo ... 59

6.2.1.1 Vecindario N1 optimiza a R1 ... 60

6.2.2 R2: En un salón no se pueden dictar dos cursos al tiempo ... 61

6.2.2.1 Vecindario N2 optimiza a R2 ... 62

6.2.3 R3: Para un profesor no programar lunes y miércoles a la misma hora 63

6.2.3.1 Vecindario N3 optimiza a R3 64

6.2.4 R4: Se deben programar todos los cursos 65

6.2.4.1 Vecindario N4 optimiza a R4 66

6.2.5 R5: Un profesor debe dictar un curso para el cual esté capacitado 67

6.2.5.1 Vecindario N5 optimiza a R5 68

6.3 Función de ajuste o de costo 69

6.4 Función objetivo 70

6.5 Penalización 71

6.6 Two points crossover 72

6.7 Mutación 76

7. Validación del algoritmo... 77

7.1 Diseño factorial 77

7.1.1 Análisis de varianza primera instancia 78

7.1.2 Análisis de varianza segunda instancia... 82

8. Implementación del algoritmo 87

8.1 Evaluación de los resultados 88

8.2 Observaciones 101

9. Conclusiones 101

10. Recomendaciones 102

Referencias Bibliográficas 104

Lista de Tablas

Pág.

Tabla 1. Cumplimiento de los objetivos 15

Tabla 2. Factores y niveles del diseño experimental 78

Tabla 3. Matriz del diseño del experimento........ 78

Tabla 4. Análisis de varianza de la para el máximo la primera instancia 78

Tabla 5. Análisis de varianza de la para el tiempo de ejecución primera instancia........ 81

Tabla 6. Análisis de varianza de la para el máximo segunda instancia 83

Tabla 7. Análisis de varianza de la para el tiempo de ejecución segunda instancia 85

Tabla 8. Resultados de los subproblemas 88

Tabla 9. Cumplimiento en asignación de cursos por profesor subproblema 1 88

Tabla 10. Cumplimiento en los días programados por profesor subproblema 1 89

Tabla 11. Cumplimiento en los periodos por profesor subproblema 1........ 90

Tabla 12. Cumplimiento en asignación de cursos por profesor subproblema 2 91

Tabla 13. Cumplimiento en los días programados por profesor subproblema 2 93

Tabla 14. Cumplimiento en los periodos por profesor subproblema 2........ 94

Tabla 15. Cumplimiento en asignación de cursos por profesor subproblema 3 94

Tabla 16. Cumplimiento en los días programados por profesor subproblema 3 95

Tabla 17. Cumplimiento en los periodos por profesor subproblema 3........ 96

Tabla 18. Cumplimiento en asignación de cursos por profesor subproblema 4 97

Tabla 19. Cumplimiento en los días programados por profesor subproblema 4 98

Tabla 20. Cumplimiento en los periodos por profesor subproblema 4........ 99

Lista de Figuras

Pág.

Figura 1. Palabras clave 19

Figura 2. Publicaciones por año 20

Figura 3. Publicaciones por autor 21

Figura 4. Publicaciones por país 22

Figura 5. Grafo 39

Figura 6. Grafo dirigido y no dirigido 39

Figura 7. Población inicial 56

Figura 8. Simulador del Algoritmo 57

Figura 9. Parámetros de prueba 58

Figura 10. Restricción 1 60

Figura 11. Vecindario N1 Optimiza a R1 61

Figura 12. Restricción 2 62

Figura 13. Vecindario N2 Optimiza a R2 63

Figura 14. Restricción 3 64

Figura 15. Vecindario N3 Optimiza a R3 65

Figura 16. Restricción 4 66

Figura 17. Vecindario N4 Optimiza a R4 67

Figura 18. Restricción 5 68

Figura 19. Vecindario N5 Optimiza a R5 69

Figura 20. Función de ajuste o de costo 70

Figura 21. Función objetivo 71

Figura 22. Crossover 73

Figura 23. Función de selección 74

Figura 24. Función para obtener los porcentajes 75

Figura 25. Operador de Selección de padres 75

Figura 26. Operador de mutación 77

Figura 27. Diagrama de Pareto de los efectos para el máximo primera instancia 79

Figura 28. Diagrama de efectos principales para el máximo primera instancia 80

Figura 29. Diagrama de Pareto de los efectos para el tiempo de ejecución primera instancia 81

Figura 30. Diagrama de efectos principales para el tiempo de ejecución primera instancia 82

Figura 31. Diagrama de Pareto de los efectos para el máximo segunda instancia 83

Figura 32. Diagrama de efectos principales para el máximo segunda instancia 84

Figura 33. Diagrama de Pareto de los efectos para el tiempo de ejecución segunda instancia 85

Figura 34. Diagrama de efectos principales para el tiempo de ejecución segunda instancia 86

Lista de Apéndices

(Los apéndices están adjuntos y puede visualizarlos en la base de datos de la biblioteca UIS)

Apéndice A. Algoritmo memético en Matlab

Apéndice B. Datos para validación del algoritmo

Apéndice C. Diseño experimental para las instancias de validación

Apéndice D. Datos a implementar

Apéndice E. División de los datos del caso de estudio

Apéndice F. Resultados del algoritmo para el caso de estudio

Apéndice G. Artículo de carácter publicable

Resumen

Título: Diseño de un algoritmo memético para resolver el problema de coloración del vértice[*]

Autores: María Camila Oliveros Cala y Aylin Brigett Amairany Suarez Quiroz[**]

Palabras Clave: problema de coloración del vértice, teoría de grafos, algoritmo memético, programación de horarios.

Descripción:

El problema de coloración del vértice consiste en colorear todos los vértices de un grafo de manera que los vértices que están conectados por una arista o borde no compartan el mismo color, esto dado que son vértices adyacentes y el hecho de tener el mismo color representa un conflicto en la coloración, a su vez se busca minimizar el número de colores usados. En el mundo real dicho problema es usado para resolver problemas tales como asignación de asientos, tareas, horarios, la planificación de actividades, la asignación de recursos, en problemas de control de producción, para proyectar redes de ordenadores, en la logística, robótica, genética, sociología, el diseño de redes, el cálculo de rutas óptimas y los problemas de almacenamiento, donde se tiene un conjunto de elementos que deben ser asignados a determinados recursos mientras se optimiza una función objetivo y se satisfacen las restricciones planteadas.

En la presente investigación se usa el problema de coloración del vértice para resolver un problema de programación de horario a través de un algoritmo memético programado en Matlab, dicho algoritmo es validado con datos aleatorios donde a través de un diseño factorial completo se evalúan los factores que influyen en el desempeño del mismo con el fin de decidir el valor que van a tomar al implementar el algoritmo en el caso de estudio de Escuela de Estudios Industriales y Empresariales de la Universidad Industrial de Santander, se concluye que el algoritmo tiene un mejor desempeño al emplear un conjunto reducido de datos y a su vez se reduce el tiempo de ejecución por lo cual se decide dividir los datos del caso de estudio en 4 subproblemas obteniendo resultados aceptables.

---

[*] Trabajo de Grado

[**] Facultad de Ingenierías Físico-mecánicas. Escuela de Estudios Industriales y Empresariales. Ingeniería Industrial. Director: Ph.D. Henry Lamos Díaz.

Abstract

Title: Design of a memetic algorithm to solve the vertex coloring problem*

Authors: María Camila Oliveros Cala y Aylin Brigett Amairany Suarez Quiroz **

Key Words: vertex coloring problem, graph theory, memetic algorithm, timetabling

Description:

The vertex coloring problem consists of coloring all the vertices of a graph in such a way that the vertices that are connected by an edge do not share the same color since they are adjacent vertices, and the fact of having the same color represents a conflict in coloration, in turn, it seeks to minimize the number of colors used. In the real world, this problem is used to solve problems such as seat allocation, tasks, schedules, activity planning, resource allocation, production control problems, design of computer networks, logistics, robotics, and son on. Genetics, sociology, network design, the calculation of optimal routes, and storage problems, where a set of elements must be assigned to specific resources while an objective function is optimized and the posed restrictions are satisfied.

In the present investigation, the vertex coloring problem is used to solve a timetabling problem through a memetic algorithm programmed in Matlab. This algorithm is validated with random data, and through a complete factorial design, the factors that influence are evaluated in order to decide the value that they are going to take when implementing it in the case study of the School of Industrial and Business Studies of the Industrial University of Santander, it is concluded that the algorithm has a better performance when using a reduced set of data and in turn, the execution time is reduced for which it is decided to divide the data of the case study into four subproblems obtaining acceptable results.

---

* Degree Work

**Facultad de Ingenierías Físico-mecánicas. Escuela de Estudios Industriales y Empresariales. Ingeniería Industrial. Director: Ph.D. Henry Lamos Díaz.

Introducción

El problema de coloración de vértices es uno de los más estudiados y aplicados dentro de la teoría de grafos y nace a partir del estudio del problema de los cuatro colores, en el cual se afirma que cuatro colores son suficientes para colorear las regiones de un mapa, de forma que las regiones limítrofes no compartan el mismo color. El problema de coloración de vértices es un problema de optimización combinatoria del tipo NP-hard en el cual se busca asignar un color a cada vértice del grafo, de tal manera que los vértices adyacentes no compartan el mismo color, esto minimizando el número de colores a usar (Moalic & Gondran, 2018).

El presente problema tiene diversas aplicaciones en el mundo real, por ejemplo, se usa para resolver problemas de programación, almacenamiento, asignación de registros, programación de horarios, asignación de frecuencias, redes de comunicación, plataformas de trenes, entre otros. En estos problemas los vértices representan los elementos que se van a programar y las aristas la incompatibilidad entre los elementos. Debido a sus múltiples aplicaciones el problema de coloración de vértices sigue siendo estudiado en la actualidad y es un área de interés cuyo estudio contribuye a la solución de problemas propios de la ingeniería industrial como la programación de horarios, de exámenes o programación de tareas en un taller de trabajo.

En las últimas décadas, numerosos investigadores han estudiado el problema de coloración de vértices y para resolverlo se han planteado diversos algoritmos que emplean técnicas exactas, heurísticas y metaheurísticas. Uno de los enfoques más recientes y prometedores se basa en la hibridación que incorpora un algoritmo de búsqueda local en el

marco de un algoritmo evolutivo para lograr así un mejor equilibrio entre intensificación y diversificación (Moalic & Gondran, 2018).

En esta investigación se va a diseñar un algoritmo memético para dar solución al problema de coloración de vértices; el algoritmo propuesto se implementará y validará a través del software Matlab y será aplicado a un problema de programación de horarios.

Tabla 1

Cumplimiento de los objetivos

Objetivo	Cumplimiento
Realizar una revisión de la literatura sobre las aplicaciones y métodos de solución del problema de coloración de vértices.	Capítulo 3
Formular el modelo matemático para resolver el problema de coloración de vértices.	Capítulo 5
Diseñar un algoritmo memético para el problema de coloración de vértices.	Capítulo 6 Apéndice A
Implementar y validar el algoritmo diseñado por medio del software Matlab usando instancias de la literatura benchmarking.	Capítulo 8 Apéndice A-B
Aplicar el algoritmo en el problema de programación de horarios.	Capítulo 8
Elaborar un artículo de carácter publicable con los resultados obtenidos en la investigación.	Apéndice G

1. Planteamiento del problema

La teoría de grafos es una de las ramas más importantes de las matemáticas modernas, nace en 1937 como resultado del trabajo del filósofo, físico y matemático suizo Leonhard Euler quien resolvió el problema de los puentes de Königsberg (Pena, 2017), el cual consistía en encontrar un camino en el que, pasando una única vez por cada uno de los puentes, se pudiera regresar al punto de partida. A partir de este punto dicha teoría se comienza a aplicar para el análisis de diversos problemas, hoy en día tiene gran importancia en áreas como lo son las ciencias sociales, físicas, ingeniería de comunicación y juega un papel importante en las ciencias de la computación, tales como inteligencia artificial, lenguajes formales, teoría de cambio y lógica de diseño, gráficos por computadora, sistemas operativos, compiladores, y organización y recuperación de información, en lo que respecta al modelado de problemas, indicando sus características de manera muy objetiva.

A lo largo de los años el desarrollo de la teoría de grafos ha sido motivado en gran parte por sus diversas aplicaciones. Está estrechamente ligada a otros campos de la matemática como la teoría de conjuntos, de teoría de grupos, la combinatoria y la programación matemática. Más allá del interés teórico, la teoría de grafos tiene una significativa importancia práctica debido a las numerosas situaciones de la vida real en las cuales surgen problemas que pueden ser modelados como un problema de coloreo de grafos.

Uno de los problemas más estudiados y aplicados de la teoría de grafos es el problema de coloración de vértices (PCV), el cual es un problema de optimización combinatoria cuyo objetivo es asignar un color a cada vértice de la gráfica utilizando la mínima cantidad de colores

posibles, pero de tal forma que los vértices adyacentes no compartan el mismo color. El problema se describe de la siguiente manera: Dado un grafo no dirigido G = (V, E) con un conjunto V de vértices y un conjunto E de aristas, un coloreo válido de G corresponde a una partición de V en k conjuntos independientes, donde un conjunto independiente es un subconjunto de vértices no adyacentes de G. La coloración del grafo tiene como objetivo encontrar la k más pequeña para un grafo dado G de modo que G tenga un color k valido. Este k mínimo se representa como $\chi(G)$ y corresponde al número cromático.

El problema de coloración de vértices tiene diversas aplicaciones en la vida cotidiana, se usa para resolver problemas tales como la asignación de asientos, tareas, horarios, la planificación de actividades, la asignación de recursos, en problemas de control de producción, para proyectar redes de ordenadores, en la logística, robótica, genética, sociología, el diseño de redes, el cálculo de rutas óptimas y los problemas de almacenamiento, donde es necesario partir del conjunto de vértices (aristas) de un grafo asociado de tal forma que los vértices (aristas) adyacentes pertenezcan a diferentes conjuntos de la partición. Problemas propios del área de ingeniería industrial en los que cierta cantidad de elementos deben ser asignados a determinados recursos mientras se optimiza una función objetivo y se satisfacen unas restricciones pueden ser resueltos al modelarse como un problema de coloración del vértice. Entre estos problemas se encuentra: problema de programación de cursos (CTP), problema de programación de exámenes (ETP), problema de programación de trabajos Job-Shop (JSSP).

2. Objetivos

2.1 Objetivo general

Diseñar un algoritmo memético para el problema de coloración del vértice

2.2 Objetivos específicos

- Realizar una revisión de la literatura sobre las aplicaciones y métodos de solución del problema de coloración de vértices.
- Formular el modelo matemático para resolver el problema de coloración de vértices.
- Diseñar un algoritmo memético para el problema de coloración de vértices.
- Implementar y validar el algoritmo diseñado por medio del software Matlab usando instancias de la literatura benchmarking.
- Aplicar el algoritmo en el problema de programación de horarios.
- Elaborar un artículo de carácter publicable con los resultados obtenidos en la investigación.

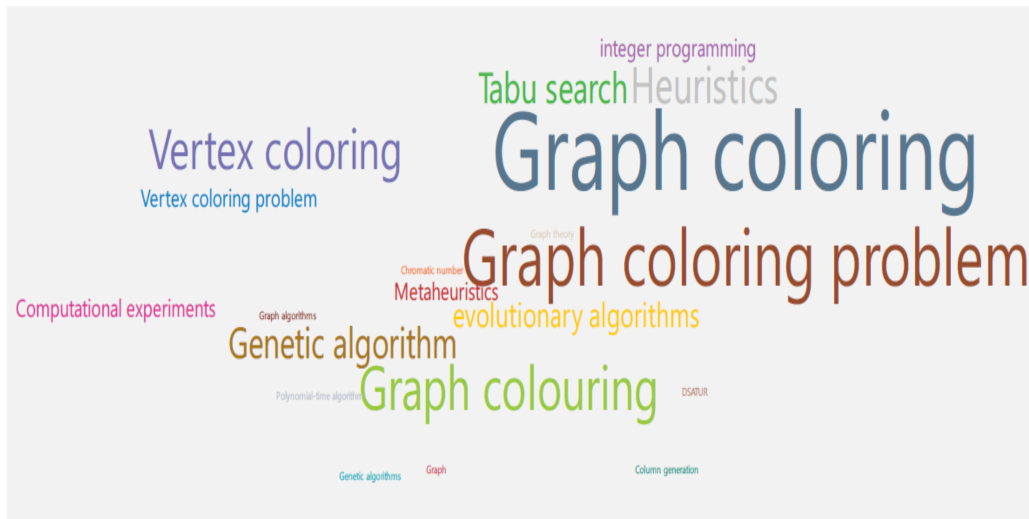
3. Revisión de la literatura

3.1 Análisis bibliométrico

La base de datos seleccionada para realizar la búsqueda de literatura fue Scopus utilizando la siguiente ecuación de búsqueda: (("k coloring problem" OR "vertex coloring problem" OR "graph coloring problem") AND (algorithm OR "genetic algorithm" OR "memetic algorithm")), se aplicaron filtros en el área temática, tipo de documento e idioma y se obtuvieron 377 artículos de los cuales se seleccionaron 87 ya que al leer el abstract y la introducción se pudo observar que son los que van acorde a la temática de la presente investigación y fue a estos artículos a los que se les realizó el análisis bibliométrico que se presenta a continuación:

Figura 1

Palabras clave

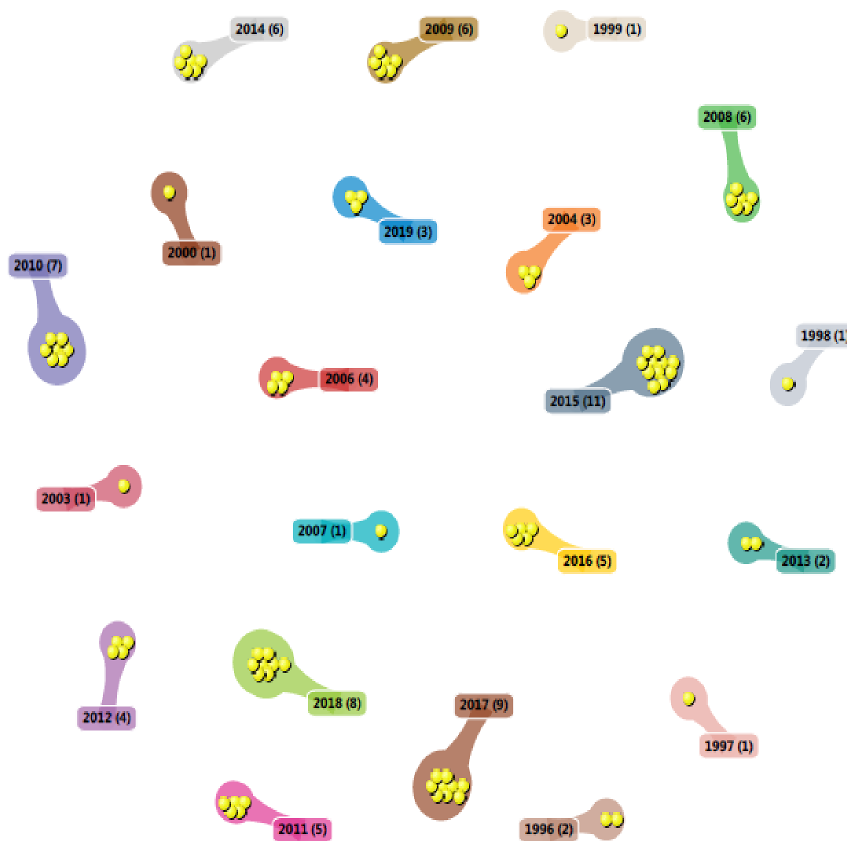


Nota. Adaptado de: vantage point.

En la Figura 1 se pueden observar las palabras claves más usadas por los autores de los artículos analizados, el tamaño de las palabras indica su nivel de uso, las más empleadas son: “Graph coloring”, “Graph coloring problem”, “Graph colouring”, “Vertex coloring” y “Heuristics”.

Figura 2

Publicaciones por año



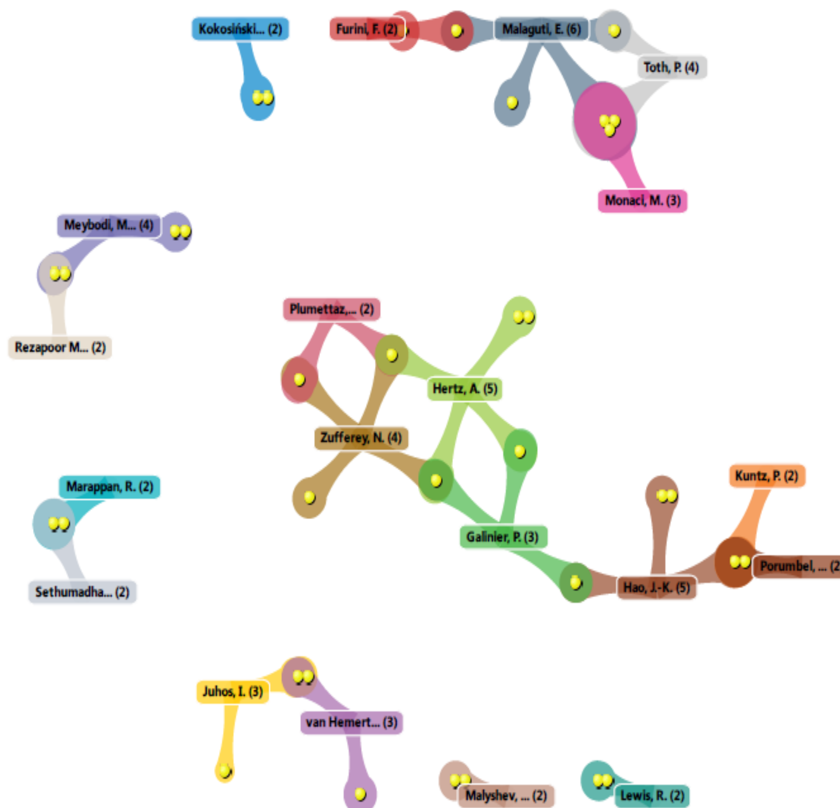
Nota. Adaptado de: vantage point.

En la figura 2 se puede observar que desde el año 1996 se han realizado publicaciones sobre el tema de estudio, en los años 2001, 2002 y 2005 no se realizaron publicaciones, pero en

los demás años se realizó al menos una, siendo el 2015 el año con mayor cantidad de publicaciones con un total de 11 artículos, se puede ver que en los últimos años se han publicado un número considerable de artículos por lo cual se evidencia que es un tema de interés para los investigadores.

Figura 3

Publicaciones por autor



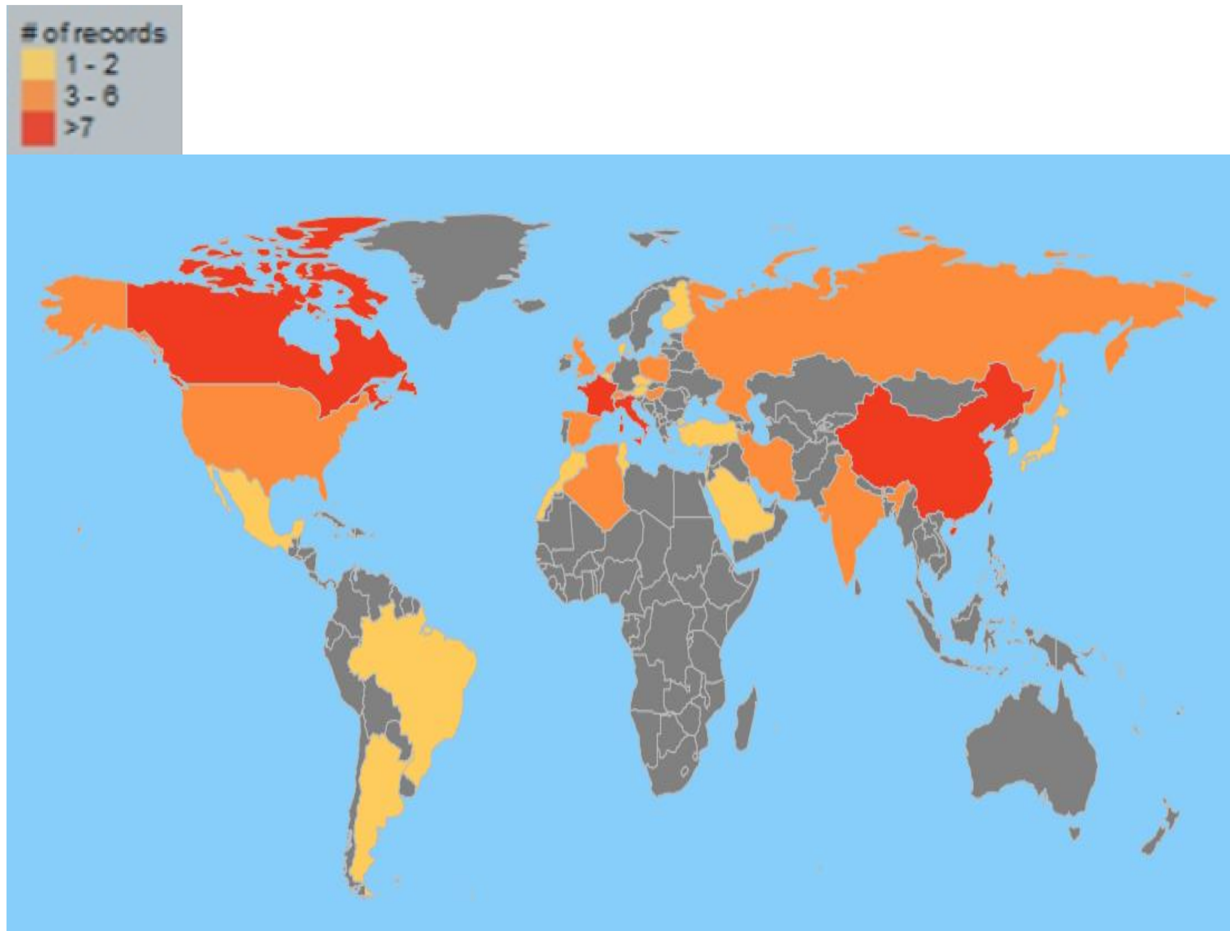
Nota. Adaptado de: vantage point.

En la figura 3 se encuentran los 20 autores con mayor cantidad de publicaciones sobre el tema de estudio, se puede observar que el autor con mayor cantidad de artículos publicados es Malaguti, E. con 6 publicaciones de las cuales 5 han sido en colaboración con otros autores lo

cual también sucede con los demás autores, se puede notar que la mayoría de ellos realiza sus investigaciones en colaboración con otros colegas.

Figura 4

Publicaciones por país



Nota. Adaptado de: vantage point.

En la Figura 4 se pueden observar los países donde se han realizado publicaciones sobre el tema de interés. Los países que han realizado 7 o más publicaciones son: Francia, China, Canadá e Italia; entre 3 y 6 artículos se han publicado en India, Irán, Reino Unido, España, Estados Unidos, Argelia, Polonia, Hungría, Países Bajos, Rusia y Suiza, los países color amarillo

en el gráfico han realizado entre 1 y 2 publicaciones mientras que los países en gris no han realizado ninguna.

3.2 Análisis preliminar de la literatura

El problema de coloración de vértices ha sido estudiado por numerosos investigadores e incluso hoy en día sigue siendo tema de estudio debido a sus diversas aplicaciones en diferentes áreas, es un problema de optimización combinatoria de tipo NP-hard por lo cual encontrar una solución óptima atrae el interés de numerosos investigadores, para este propósito se han planteado gran variedad de enfoques, sin embargo, para resolver el problema, sigue siendo necesario desarrollar algoritmos rápidos y seguros (Pahlavani & Eshghi, 2011).

Dado un grafo no dirigido $G = (V, E)$, donde V es un conjunto de vértices y E es un conjunto de aristas o bordes, colorear los vértices de G consiste en asignar un color a cada vértice de modo que los vértices adyacentes tengan colores diferentes (Moalic & Gondran, 2018). El objetivo del problema es encontrar el número mínimo de colores necesarios para colorear el grafo G, denominado número cromático $\chi(G)$.

Una k coloración para el grafo G es una función $\Phi: V \rightarrow \Gamma$. Donde $\Gamma = \{1, 2, ..., k\}$ es un conjunto de enteros y cada uno representa un color. Se tiene una coloración válida si no hay dos puntos finales de cada arista que tengan el mismo color. Es decir que para todo $[u, v] \in E$ $\Phi(u) \neq \Phi(v)$. Si los puntos finales u y v de cualquier arista tienen el mismo color, los vértices u y v están en conflicto (Rezapoor Mirsaleh & Meybodi, 2016).

Para solucionar el problema de coloración de vértices se han utilizado algoritmos exactos y técnicas heurísticas, esto dependiendo del tamaño del grafo. Para los grafos pequeños

aplicando algoritmos exactos se logra obtener soluciones óptimas, pero en el caso de grafos grandes que tengan más de 100 vértices se hace necesario aplicar técnicas heurísticas con las cuales se puede llegar a soluciones casi óptimas (Galán, 2017). El número de enfoques exactos que se han propuesto para resolver este problema es pequeño comparado con la gran cantidad de algoritmos heurísticos que han sido desarrollados (Malaguti & Toth, 2010). Dentro del enfoque heurístico se encuentran los algoritmos codiciosos y las metaheurísticas tales como búsqueda tabú, algoritmos evolutivos y algoritmos híbridos. Los principales algoritmos desarrollados para resolver el problema de coloración de vértices se explican a continuación; primero se abordan los enfoques exactos y posteriormente los métodos heurísticos.

Galán (2017) describe los primeros algoritmos exactos propuestos, el primero de ellos es un algoritmo de enumeración implícito propuesto por Brown en 1972, donde para resolver el problema se colorea un vértice del grafo a la vez haciendo uso de un color ya asignado a otro vértice o un nuevo color, en 1979 Brélaz diseña el algoritmo DSTUR desarrollando la idea propuesta por Brown, este algoritmo funciona subdividiendo el problema en subproblemas los cuales representan una coloración parcial del grafo, cuando una de estas coloraciones usa k colores y $k \geq UB$ (límite superior de la cantidad de colores necesarios) el subproblema se puede comprender, cuando se han coloreado todos los vértices y $k < UB$ el límite superior es reemplazado por k. Por cada subproblema surgen hasta k+1 subproblemas y cuando es factible se asigna uno de los k colores al siguiente vértice y se colorea con el color k+1 si $k+1<UB$. El siguiente vértice por colorear es el que tiene el mayor grado cromático (número de colores diferentes de sus vecinos), en caso de empate se selecciona el vértice de mayor grado (número de aristas que inciden en el vértice) en el subgrafo incoloro.

Malaguti y Toth, (2010) explican las dos mejoras para el algoritmo DSATUR propuestas por Sewell en 1996, la primera consiste en colorear una camarilla (subgrafo completo de G) máxima en el grafo y haciendo uso de algoritmos heurísticos calcular el límite superior; la segunda mejora es una técnica de desempate diferente en la cual se selecciona el vértice que minimice el número de colores disponibles para los vértices incoloros restantes. A través de experimentos computacionales se comprueba la efectividad de este algoritmo y se observa que puede resolver un conjunto de instancias mayor que las del algoritmo original.

Otros autores han propuesto métodos exactos basados en modelos de programación de enteros, Malaguti y Toth (2010) también describen un algoritmo de Branch & Price basado en la formulación "set covering" VCP-SC propuesto por Mehrota y Trick en 1996, dicha formulación tiene como objetivo minimizar el número total de conjuntos independientes (colores), cada conjunto independiente se asocia con una variable binaria que toma el valor de 1 si los vértices del conjunto tienen el mismo color. Al comparar este algoritmo DSATUR incluyendo las mejoras propuestas por Sewell se puede observar que ambos pueden resolver grafos aleatorios de hasta 70 vértices y grafos geométricos aleatorios de hasta 250 vértices.

Méndez-Díaz y Zabala (2006) proponen un algoritmo Branch & Cut el cual es una refinación del algoritmo Branch & Bound en el cual las relajaciones de la programación lineal se fortalecen con desigualdades válidas en todo el árbol de enumeración, el algoritmo es denominado BC-Col y para reducir el número de nodos en el árbol de enumeración implementa las siguientes consideraciones: generar límites duales y superiores de buena calidad, reglas adecuadas para particionar conjuntos de factibilidad y diseño de estrategias eficientes para explorar el árbol de enumeración. Al probar el algoritmo se llega a la conclusión de que es un

algoritmo prometedor pare resolver el problema de coloración de vértices con gran capacidad de mejora diseñando algoritmos de separación más efectivos, incorporando nuevas familias de desigualdades válidas y utilizando una herramienta de poda más efectiva.

San Segundo (2012) propone una regla de desempate llamada PASS que se ejecuta con mayor facilidad que la de Sewell, en dicha regla el objetivo es reducir los dominios de color de los vértices que tienen menos colores disponibles. El autor sugiere no aplicar la regla PASS en todos los pasos, solo en aquellos donde el número de colores disponibles para colorear el vértice sea menor a un cierto parámetro μ calculado en cada paso debido a que esto mejora el rendimiento del algoritmo y lo catalogan como un algoritmo exacto eficiente para resolver el problema de coloración de vértices en aplicaciones de la vida real.

Furini, Gabrel y Ternier (2012) explican que en los algoritmos DSATUR basados en Branch & Bound propuestos hasta el momento los investigadores se centran en mejorar las técnicas de selección del nuevo vértice a colorear por lo que el enfoque que desarrollan en su investigación es mejorar el límite inferior ya que este valor se estima al comienzo y nunca es actualizado. Utilizan tres límites inferiores: número de camarilla, número de Lovász Theta y número de estabilidad de un grafo auxiliar. Al realizar los experimentos computacionales concluyen que utilizar el número de camarilla resulta computacionalmente costoso y el tiempo computacional es alto, el número de Lovász Theta este también resulta computacionalmente costoso ya que casi siempre alcanza el tiempo límite y en cuanto al número de estabilidad para instancias de alta densidad puede probar la optimalidad en el nodo raíz, lo cual es bastante útil en el algoritmo DSATUR ya que, para este, dichas instancias resultan difíciles. Para futuras investigaciones proponen “desarrollar reglas más sofisticadas para desencadenar el cálculo del

límite solo en la parte más prometedora del árbol de enumeración o cuando la brecha entre el límite inferior y el límite superior es baja" (Furini, Gabrel, & Ternier, 2016).

Los algoritmos codiciosos fueron la base de los primeros algoritmos heurísticos desarrollado para resolver el problema de coloración de vértices (Malaguti & Toth, 2010), estos algoritmos construyen las coloraciones de manera secuencial coloreando un vértice del grafo a la vez hasta obtener una coloración completa, para seleccionar el siguiente vértice a colorear y el color a usar siguen una serie de reglas propias de cada algoritmo.

El algoritmo secuencial codicioso SEQ, es el más simple y en este los vértices son etiquetados, al primer vértice se le asigna la primera clase de color (conjunto independiente de vértices de un color) y al siguiente vértice se le asigna la clase de color indexada más baja que no tenga vértices adyacentes al vértice que se va a colorear (Malaguti & Toth, 2010).

Adegbindin et al. (2016) describen el algoritmo recursivo más largo primero RLF propuesto por Leighton en 1979, en el cual todos los vértices posibles son asignados a la primera clase de color antes de continuar con la siguiente. Se definen 3 subconjuntos: C, la siguiente clase de color a construir, U los vértices incoloros y W los vértices incoloros que tienen al menos un vecino en C. El primer vértice que se agrega a C es el que tiene mayor número de vecinos en U, el resto del grafo se colorea así: mientras U no esté vacío se agrega a C el vértice en U que tiene mayor número de vecinos en W, en caso de empate se escoge el vértice cuyo número de vecinos en U es menor.

Una técnica metaheurística bastante estudiada para resolver el problema de coloración del vértice es la búsqueda local, en la cual se parte de una solución inicial y esta va siendo mejorada

a través de un método de búsqueda local que se mueve desde la coloración actual hacia una mejor. Varios investigadores han propuesto diferentes técnicas de búsqueda local para la solución del problema, Galinier & Hertz (2006) describen el primer método propuesto por Chams et al. En 1987, este es un algoritmo de recocido simulado que explora entre un conjunto de k coloraciones buscando minimizar el número de aristas en conflicto, Chams et al. además proponen un algoritmo que funciona en dos fases: en la primera utilizan un algoritmo codicioso para crear varias clases de color extrayendo conjuntos estables del grafo y en la segunda fase aplican el algoritmo de recocido simulado. Se concluyó que para grafos grandes la extracción de los conjuntos estables ofrece mejores resultados.

Moalic & Gondran (2018) explican el método de búsqueda local propuesto por Hertz y de Werra en 1987 llamado Tabucol, el cual utiliza la búsqueda tabú que fue introducida en 1986 por Fred Glover. Tabucol posee tres características básicas que son el espacio de búsqueda y función objetivo, el vecindario y la estrategia de movimiento. En cuanto al espacio de búsqueda se considera un número fijo k de colores y se tienen en cuenta coloraciones incorrectas, el objetivo es encontrar una coloración en la cual se minimicen las aristas en conflicto. El vecindario de una coloración son aquellas soluciones en las cuales el color de un vértice en conflicto es diferente al de la solución actual. Por último, la estrategia de movimiento consiste en escoger el mejor vecino en cada iteración que no esté en la lista tabú incluso si la función objetivo es peor, en la lista tabú son almacenados los colores que se han asignado a los vértices y se prohíbe que dicha asignación sea aplicada en determinado número de iteraciones.

Johnson et al. en 1991 realizan una comparación de tres estrategias de recocido simulado y además proponen un algoritmo XRLF en el que integra la extracción de conjuntos estables con

una versión mejorada del algoritmo RLF, como explican Galinier & Hertz (2006) la extracción se realiza hasta que el grafo tenga menos de 70 vértices y el grafo restante se colorea con un algoritmo exacto. En la primera estrategia de recocido simulado analizada las posibles soluciones no tienen un número fijo de clases de color definido y el objetivo es minimizar el número de colores y de bordes en conflicto, el segundo algoritmo. La segunda estrategia tampoco tiene un número de colores fijos y considera como soluciones todas las coloraciones legales, en este método es necesario utilizar una serie de movimientos complicados llamados intercambios de cadena Kempe. El ultimo método es la estrategia de penalización utilizada por Chams et al. Al comparar estas estrategias con el algoritmo propuesto se concluye que el nuevo método funciona mejor en ciertos grafos grandes.

Años más tarde, como explican Torkestani & Meybodi, (2010) un algoritmo de búsqueda local iterado (ILS) es propuesto por Lourenco et al. en 2002 para resolver el problema de colorear grafos. Posteriormente los investigadores Voudouris y Tsang en 2003 se unen a esta búsqueda y proponen una Búsqueda Local Guiada (GLS), que es un método de búsqueda metaheurística, para resolver los problemas de optimización combinatoria.

Por otra parte, en 2006 Caramia et al. proponen un algoritmo de búsqueda local prioritario, denominado CHECKCOL, en el cual el tiempo de ejecución del algoritmo se disminuye al evitar búsquedas innecesarias en grandes partes del grafo que no implican ningún progreso en la solución. Además, en este algoritmo, a cada vértice del grafo se le asigna dinámicamente una prioridad. “Estas prioridades se utilizan para definir un nuevo y más eficaz esquema de memoria a largo plazo, que se integra con el esquema de memoria a corto plazo implicado por los puntos de control. El concepto de punto de control junto con la prioridad tiene

un impacto significativo en la calidad de la solución, la conciencia de la caché, y el tiempo de funcionamiento del algoritmo". (Akbari Torkestani & Meybodi, 2011).

Caramia y Dell'Olmo en 2008 proponen dos fases de búsqueda local para colorear vértices (Akbari Torkestani & Meybodi, 2011), el algoritmo ejecuta alternativamente dos funcionalidades que interactúan rigurosamente, una de ellas es la fase estocástica y la otra es una búsqueda local determinista. La fase estocástica como tal se basa en un muestreo aleatorio sesgado en el que las coloraciones factibles se construyen frecuentemente. Mientras que en la fase determinista cada vértice se asigna al color que causa el menor aumento de la función de penalización. En este algoritmo, la función objetivo intenta minimizar la función de penalización.

Otros algoritmos metaheurísticos que se han desarrollado son los algoritmos evolutivos, algoritmos genéticos, algoritmos meméticos, colonia de hormigas y una gran cantidad de algoritmos híbridos que resultan más eficientes para resolver el problema. A continuación, se describen los principales algoritmos en estas categorías:

Fleurent & Ferland, (1996) describen un algoritmo genético que se basa en la codificación basada en orden propuesto por Davis en 1991, en este algoritmo los individuos son representados como permutaciones de los vértices, los vértices son coloreados de manera secuencial con un algoritmo codicioso, en el orden definido por la permutación. Además, Davis define un operador de cruce y un operador de mutación, para generar una permutación de descendencia entre dos padres; además es generada una cadena aleatoria de bits de tamaño |V|, en las posiciones cuyo valor es 1 en la descendencia se toman los valores que tiene el padre 1 en esa misma posición, las demás posiciones se completan en orden con los valores del padre 2 que

no han sido asignados en la descendencia. En cuanto al operador de mutación se seleccionan dos posiciones al azar en la permutación y las posiciones contenidas entre estas dos, incluyéndolas se reordenan de manera aleatoria.

Costa et al. (1995) y Fleurent y Ferland (1996) exploran los algoritmos genéticos combinados con una búsqueda local (GLS) para resolver el problema. La diferencia entre este método y el algoritmo genético puro es que en este caso el operador de mutación es reemplazado por un operador de búsqueda local, con este método obtuvieron soluciones ligeramente mejores que las resultantes al usar solo un método de búsqueda local como Tabucol, esto se atribuye a la diversificación que se introduce al hacer uso de operadores de cruce.

Posteriormente Galinier y Hao en 1999 proponen un algoritmo evolutivo llamado HEA que trabaja con un valor k fijo y combina el algoritmo Tabucol mejorado con un operador de cruce especial GPX para resolver problemas de coloración de vértices. Tal como explican Moalic & Gondran (2018) el operador de cruce GPX funciona bajo dos principios, el primero es una coloración es una partición de los vértices en clases de color y segundo las clases de color más grandes deben ser heredadas por los hijos, primero se transmite al hijo la clase de color más grande del primer padre (en caso de empate se selecciona aleatoriamente) y luego la clase de color más grande del segundo padre. Si después de k pasos se tienen vértices sin colorear estos son asignados a una clase de color aleatoria.

Costa y Hertz (1999) proponen el algoritmo ANTCOL, el cual es un algoritmo evolutivo que se basa en el comportamiento natural de las hormigas para colorear un grafo, cada hormiga de la colonia construye una solución factible en cada ciclo, la experiencia acumulada durante las construcciones se almacena en una matriz que se actualiza al final de cada ciclo, para la elección

del siguiente color a usar y vértice a colorear los investigadores prueban varios algoritmos codiciosos y llegan a la conclusión que el mejor es el algoritmo RLF.

Myszkowski (2008) propone un algoritmo evolutivo GRACOM para ser aplicados en problemas de programación del mundo real. El algoritmo consta de una función de aptitud parcial pff y de dos operadores genéticos especializados: IBIS como operador de mutación y BCX como operador de cruce. La función pff brinda información sobre la cantidad de conflictos que se tienen en determinada coloración lo que además de permitir analizar la coloración determina cuales vértices tienen mejores colores (menos conflictos) y cuáles no. IBIS funciona mediante un operador de lógica difusa que determina dos valores: la probabilidad de color restante con la que se decide si el vértice debe ser recoloreado y el tamaño de torneo que da el número de colores actuales entre los cuales se elige el mejor. El objetivo principal del operador BCX es conectar dos coloraciones para construir una mejor coloración con menos conflictos, el operador analiza el color de cada vértice en ambos padres y de acuerdo con el número de conflictos selecciona el color del vértice con menos conflictos. El autor concluye que el algoritmo es de alta eficiencia y es una buena base para investigaciones futuras.

Lü y Hao (2010) proponen un algoritmo memético llamado MACOL que integra un procedimiento de búsqueda tabú con un algoritmo evolutivo. El algoritmo tiene cuatro componentes: un generador de la población inicial, procedimiento simple de búsqueda tabú, multi-operador de cruce principal y la regla de actualización de la población. Para generar la población inicial se usa una versión aleatoria del algortimo DANGER, el próximo vértice a colorear se selecciona de acuerdo a la “medida dinámica de peligros del vértice”y se le asigna el color que tiene menos probabilidad de ser asignado a sus vecinos. Para realizar el cruce,

proponen un nuevo operador AMPaX que se puede ver como una extensión de GPX, a diferencia de GPX en AMPaX para producir una descendencia se pueden usar dos o más padres y en cada paso se elige de forma adaptativa un padre y clase de color para asignar a la descendencia. Para actualizar la población la regla de actualización tiene en cuenta la calidad de la solución y la diversidad de las soluciones, el objetivo es no insertar en la población soluciones de baja calidad o muy similares a otros individuos de la población. Se prueba que el algoritmo propuesto es efectivo y logra resultados competitivos respecto a otros algoritmos, además se resalta la importancia de usar mecanismos para la diversificación.

Rezapoor Mirsaleh y Meybodi (2016) proponen un algoritmo memético basado en autómatas de aprendizaje celular denominado CLAMACOL. En el algoritmo propuesto el entorno local lo constituyen autómatas de aprendizaje vecinos de cualquier celda mientras que el entorno global consiste en un grupo de memes, cada uno equipado con un determinado método de búsqueda local y representado por un conjunto de autómatas de aprendizaje que contiene el respectivo historial de búsqueda. El grafo de entrada es representado por un IOCLA isomorfo, cada vértice del grafo se asocia a una celda del IOCLA y cada celda contiene un autómata de aprendizaje cuyas acciones constituyen el conjunto de colores con los cuales se puede colorear el vértice. CLAMACOL consta de varias etapas y en cada una de ellas se encuentra localmente un color para cada celda y sus vecinos. En cada etapa el autómata de aprendizaje elije un color dependiendo del vector de probabilidad de acción, el color es aplicado al entorno local y al global. El entorno local produce una respuesta favorable o desfavorable dependiendo de los colores seleccionados por los autómatas de aprendizaje vecinos. El entorno global recibe los colores seleccionados por las celdas y genera una nueva solución combinando estos colores,

posteriormente compara la idoneidad de la nueva solución con la anterior que es mejorada con el método de búsqueda local, si la nueva solución es más idónea que la anterior se reemplaza, de lo contrario permanece igual. El algoritmo aplicado a grafos de referencia difíciles de colorear y se concluye que funciona mejor que otros algoritmos conocidos tanto en tiempo de ejecución como en cantidad de colores.

En 2018 Rezapoor Mirsaleh y Meybodi proponen un nuevo algoritmo memético basado en el enfoque de algoritmo genético Michigan en el cual toda la población hace parte de la solución, es decir cada cromosoma representa un color. El algoritmo que proponen para resolver el problema de coloración del vértice es llamado MLAMACOL, en el cada cromosoma evoluciona localmente a través de operadores evolutivos y es mejorado con un método de búsqueda local, consta de una sección genética, una sección memética y se emplea búsqueda local. En la sección genética cada vértice es asociado con un cromosoma que representa el color del vértice con un número entero, el cromosoma inicial se crea escogiendo de manera aleatoria un color, al comienzo de cada generación un operador de mutación es aplicado. En la sección memética se tiene un meme por cada vértice del grafo, cada meme está equipado por un autómata de aprendizaje y guarda el historial de la búsqueda local de cada cromosoma. El método de búsqueda local es aplicado en cada cromosoma en base a la información genética y memética del cromosoma y los cromosomas de los vértices vecinos. Al comparar MLAMACOL con otros algoritmos se concluye que supera a estos algoritmos en número de colores y tiempo de ejecución.

Moalic y Gondran (2018) proponen un algoritmo memético basado en el algoritmo híbrido HEA de Galinier y Hao. Este nuevo algoritmo HEAD funciona con una población de

solo dos individuos y propone una nueva forma para gestionar la diversidad. Los autores presentan primero el algoritmo HEAD’ que es descrito como dos algoritmos Tabucol funcionando en paralelo los cuales interactúan a través de un operador de cruce, primero se inicializan las dos soluciones y con el operador de cruce se introduce cierta diversidad, los dos descendientes se mejoran con Tabucol y los dos padres son reemplazados por las mejores soluciones. El segundo algoritmo presentado es HEAD, en este se agregan otras dos soluciones candidatas elite 1 y elite 2, después de cierto número de generaciones los dos individuos de la población se vuelven bastante similares por lo cual la idea es reemplazar una de las dos soluciones candidatas por una solución encontrada previamente por el algoritmo. La solución elite1 es la mejor solución encontrada durante el ciclo actual y la solución elite2 la mejor solución encontrada durante el ciclo anterior. Al final de cada ciclo, la solución elite2 reemplaza a uno de los individuos de la población. Los resultados obtenidos al implementar el algoritmo sugieren que este puede ser aplicado de manera exitosa a otros problemas donde se empleen operadores de cruce estocásticos o asimétricos.

4. Marco teórico

4.1 Optimización combinatoria

La Investigación de Operaciones (OR) es un área de investigación interdisciplinaria con enlaces a Economía, Matemáticas, Negocios, Estadística, Ingeniería, e Informática. El objetivo de OR es encontrar soluciones para problemas de aplicación mundial. Esto se realiza principalmente a través de la construcción de un modelo matemático donde su objetivo es

encontrar la solución óptima para la función objetivo planteada, su análisis a través de diferentes herramientas y los resultados que se obtienen, para plantear las recomendaciones con respecto a los problemas en el mundo real. (Kostuch, 2003, p. 5). Por lo tanto, los problemas de optimización se dividen en 2, aquellos problemas de optimización que tratan variables de decisión continua y los que tratan con variables de decisión discretas. En Estos últimos encontramos los Problemas de optimización combinatoria.

La Optimización combinatoria es el proceso de búsqueda del valor máximo o mínimo de la función objetivo, cuya estructura es discreta y presenta una complejidad computacional y diversas aplicaciones en el mundo real.

La solución debe ser una combinación que:

- Respeta todas las restricciones propuestas.

- Maximiza / Minimiza el valor de la función objetivo.

La optimización combinatoria está relacionada con la modelación y los diferentes métodos de solución de problemas definidos en un conjunto discreto. En general, los problemas de optimización combinatoria son clasificados de acuerdo con su complejidad computacional, y esto ha llevado al desarrollo de muchos algoritmos para hallar la solución óptima del problema. La optimización combinatoria no sólo es útil para comprender la complejidad de los algoritmos sino también permite verificar si una propuesta de solución de un problema de optimización discreta es óptima. (Aguilar Imitola & Perez Diaz, 2013) llevado al desarrollo de muchos algoritmos para hallar la solución óptima del problema.

4.2 Complejidad computacional

La complejidad computacional se presenta debido a la dificultad del problema que se va a tratar, las variables y las restricciones con las que se plantea el problema. Está determinada como tal, por las especificaciones que se plantean en un modelo de computación, el modo de computación y los recursos (espacio o tiempo). "Según Papadimitriou la complejidad se determina por la dificultad del cálculo que está directamente relacionado con el uso de los recursos" (Aguilar Imitola & Perez Diaz, 2013).

4.2.1 Clases de complejidad

4.2.1.1 P (Polinomial). La complejidad de este tipo de problemas permite dar una solución determinista a través de la máquina de Turing en un tiempo polinomial. Por esta razón es que estos algoritmos de complejidad polinómica son más abordables en la práctica.

4.2.1.2 NP. Se pueden desarrollar problemas en un tiempo polinomial no determinista a través de la máquina de Turing.

4.2.1.3 NP Complete (NP Completo). La complejidad de este tipo de problemas requiere de un tiempo exponencial para ser solucionado. Un problema P es NP Completo si $P \in NP$ y todos los problemas de clase NP pueden ser reducidos a un problema P en un tiempo polinomial, esto implica que son problemas difíciles de resolver dentro de la clase NP (Aguilar Imitola & Perez Diaz, 2013).

4.2.1.4 NP Hard (NP difícil o complejo). Son problemas similares a los NP completo, pero son muy difíciles de resolver, por lo que necesitan tiempo exponencial o incluso mucho mayor para poder ser resueltos.

4.3 Teoría de grafos

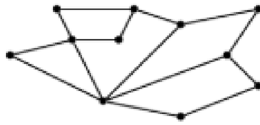
La teoría de grafos es un área de las matemáticas discretas que se encarga de resolver problemas que tienen que ver con cierto de número de puntos que están unidos a través de ciertos trazos. El trabajo de Leonhard Euler, en 1736, sobre el problema de los puentes de Königsberg es considerado el primer resultado de la teoría de grafos, dicho problema planteaba si era posible trazar una ruta que recorriera los 7 puentes de la ciudad, pasando una sola vez por cada puente y regresando al punto inicial, Euler encontró que no era posible, pero de este estudio nace el concepto de grafo euleriano. En 1852 Francis Guthrie planteó el problema de los cuatro colores que busca saber si es posible, utilizando solamente cuatro colores, colorear cualquier mapa de países de tal forma que dos países vecinos nunca tengan el mismo color. Este problema, que no fue resuelto hasta un siglo después por Kenneth Appel y Wolfgang Haken y puede ser considerado como el nacimiento de la teoría de grafos (Pena, 2017). En la actualidad la teoría de grafos sigue siendo estudiada para resolver diversos tipos de problemas de la vida real y para su estudio se deben tener en cuenta ciertos términos que serán explicados a continuación.

4.3.1 Definiciones básicas

4.3.1.1 Grafo. Un grafo G es un par $G=(V, E)$, donde V representa un conjunto de puntos en el espacio conocidos como nodos o vértices y E representa un conjunto de pares no ordenados de vértices denominados aristas o arcos. Dos vértices conectados por la misma arista son llamados vértices adyacentes. Un ejemplo de grafo se presenta a continuación:

Figura 5

Grafo

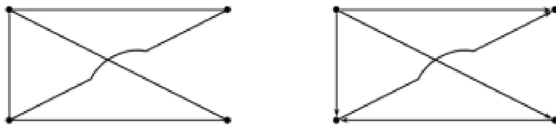


Nota. Adaptado de Caicedo, A., Wagner, G. y Méndez, R. (2010).

4.3.1.2 Grafo dirigido. Un grafo dirigido u orientado es aquel en el que las aristas tienen una dirección por lo que en cada arista existe un vértice de origen y un vértice destino. Cuando las aristas no tienen dirección el grafo es denominado no dirigido y en este caso los vértices de una arista se pueden nombrar sin ningún orden especifico. En la siguiente figura se observan un grafo dirigido y no dirigido:

Figura 6

Grafo dirigido y no dirigido



Nota. Adaptado de Caicedo, A., Wagner, G. y Méndez, R. (2010).

4.3.1.3 Vértices adyacentes. Se dice que dos vértices son adyacentes cuando están conectados por la misma arista.

4.3.1.4 Grado de un vértice. El grado de un vértice corresponde al número de aristas que inciden en un vértice.

4.4 Problema de coloración del vértice

Dado un grafo no dirigido $G = (V, E)$ con un conjunto de vértices V y un conjunto de bordes E, colorear los vértices de un grafo consiste en asignar a cada vértice un color de manera que los vértices adyacentes tengan diferentes colores. El objetivo el problema de coloración del vértice consiste en encontrar el número mínimo de colores necesarios para colorear el grafo G.

Siendo k un entero positivo correspondiente al número de colores una coloración k de un grafo G, c es una función que asigna a cada vértice un número entero entre 1 y k de la siguiente manera:

$$c: V \rightarrow \{1, 2 ..., k\}$$

$$v \rightarrow c(v)$$

El valor $c(v)$ corresponde al color del vértice v. Los vértices asignados al mismo color $i \in \{1, 2 ..., k\}$ definen una clase de color que se denota como Vi (Moalic & Gondran, 2018).

4.4.1 Definiciones

4.4.1.1 Número cromático $\chi(G)$. Número mínimo de colores necesarios para colorear un grafo de manera que se obtenga una coloración válida o legal.

4.4.1.2 Coloración válida o legal. Es aquella que respeta las siguientes restricciones binarias: $\forall (u, v) \in E, c(u) \neq c(v)$ que implican que no haya dos puntos finales en una arista con el mismo color.

4.4.1.3 Vértices en conflicto. Cuando los puntos finales de una arista u y v tienen el mismo color, los vértices u y v están en conflicto.

4.4.1.4 Coloración completa. Se da cuando a todos los vértices se les ha asignado un color, en caso contrario la coloración es parcial.

4.4.1.5 Conjunto independiente o estable. Es un subconjunto de vértices en el que ninguno de ellos es adyacente por lo que se les puede asignar el mismo color.

4.4.1.6 Clase de color. Es un subconjunto en el que se encuentran todos los vértices asignados a un color.

4.4.1.7 Grado cromático de un vértice. Corresponde al número de colores asignado a los vértices adyacentes de un determinado vértice.

4.4.1.8 Camarilla de un grafo. Subconjunto de vértices completamente conectados, su tamaño representa un límite inferior válido para el problema.

4.5 Aplicaciones del problema de coloración del vértice

El problema de coloración del vértice tiene diversas áreas de aplicación como en programación de horarios, asignación de registros, asignación de frecuencias, gestión de la cadena de suministro, plataformas de trenes, gestión del flujo de tráfico aéreo, asignación de recursos de red, entre otras. A continuación, se describen las aplicaciones que pueden ser empleadas en el área de ingeniería industrial.

4.5.1 Problema de programación de cursos (CTP)

En este tipo de problemas se debe asignar una serie de eventos que pueden ser clases a un conjunto de recursos como salones de clase y tiempo mientras se satisfacen una serie de restricciones que pueden ser propias del problema, por ejemplo, no se pueden asignar dos clases

a un mismo salón en el mismo intervalo de tiempo, también puede haber restricciones específicas de la institución que deben ser cumplidas.

4.5.2 Problema de programación de exámenes (ETP)

Este problema consiste en programar una serie de exámenes en determinado periodo de tiempo, se deben cumplir determinadas restricciones, principalmente dos exámenes no pueden ser programados al mismo tiempo, también se debe tener en cuenta el tiempo de estudio entre exámenes y un límite de exámenes por tiempo. Al resolver el problema de programación de exámenes como un problema de programación de vértices los exámenes corresponden a los vértices y la restricción de programar dos exámenes al mismo tiempo estaría representada por una arista, el conjunto de intervalos de tiempo puede ser representado por un conjunto de colores usados y una coloración no es legal cuando dos exámenes se programan al tiempo.

4.5.3 Problema de programación de Job-Shop

En este problema se tienen un conjunto de tareas con un tiempo de procesamiento dado que se deben asignar a un conjunto de máquinas. Una programación válida es aquella en la que se conserva el orden de las tareas, cada máquina procesa una tarea a la vez y por último dos tareas no pueden ser procesadas en paralelo.

4.6 Métodos de solución para el problema de coloración del vértice

Debido a que el problema de la coloración de vértices es uno de los problemas más atrayentes de la optimización combinatoria NP-hard, encontrar una solución óptima tiene gran

interés para los diferentes grupos de investigadores. Para este propósito se han utilizado algoritmos exactos y aproximados dependiendo del tamaño del grafo.

4.6.1 Algoritmos exactos

"Los algoritmos exactos son aquellos que siempre determinan la solución óptima de un problema computacional" (Pena, 2017). Debido a que el problema de coloración de vértices es de tipo NP-hard los algoritmos exactos que han sido propuestos son solo capaces de colorear grafos aleatorios pequeños que tengan hasta 100 vértices, estos algoritmos siempre resultan en una solución óptima (Galán, 2017), aunque presentan algunos inconvenientes, como, por ejemplo, pueden ser demasiado lentos y no tienen tiempo de complejidad polinomial para resolver los problemas de tipo NP-hard. Dado que los grafos que surgen de problemas reales son demasiado grandes la aplicación de algoritmos exactos en estos es limitada.

4.6.1.1 Branch & Bound. Este algoritmo resuelve el problema basándose en la división y la exploración. Se divide con el fin de disminuir el espacio de búsqueda y encontrar soluciones de manera más fácil, la división se realiza de forma sucesiva hasta obtener en cada subespacio una solución entera. En la exploración se determinan soluciones parciales para cada subespacio y con base a estas soluciones el algoritmo va descartando subespacios. La mejor solución de las encontradas en los subespacios es el óptimo global (Tapias, Galeano, & Hincapie, 2011).

4.6.1.2 Branch & Cut. Este algoritmo es una refinación de Branch & Bound, se usan planos de corte en cada subproblema, "El esquema general de solución es generar cortes globalmente válidos (tanto para los nodos en donde se introduce, como para el problema original) en cada nodo del árbol de ramificación para obtener una secuencia de aproximaciones a

la cubierta convexa del programa entero, esto es, ajustarse cada vez mejor al conjunto de soluciones factibles" (Guerrero, 2010).

4.6.2 Algoritmos aproximados

Debido a que los métodos exactos no pueden ser aplicados para grandes grafos, los investigadores han propuesto métodos que si bien no dan una solución exacta del problema brindan soluciones que se acercan bastante a la solución óptima. Los métodos aproximados se dividen en dos grandes grupos: heurísticas y metaheurísticas.

4.6.2.1 Heurísticas. Las heurísticas hacen referencia a aquellas técnicas que se emplean para obtener soluciones de alta calidad a un costo computacional razonable a pesar de no ser las soluciones óptimas ni saber que tan lejos están de estas (De Antonio Suárez, 2011). Las primeras heurísticas que se desarrollaron para resolver el problema de coloración del vértice son los algoritmos codiciosos, estos algoritmos construyen las coloraciones de manera secuencial coloreando un vértice del grafo a la vez hasta obtener una coloración completa, para seleccionar el siguiente vértice a colorear y el color a usar siguen una serie de reglas propias de cada algoritmo. Las principales ventajas de estos algoritmos son la facilidad de su implementación y la rapidez, pero sus soluciones comparadas con las que se obtienen empleando algoritmos metaheurísticos no son de buena calidad ya que se suelen necesitar más colores de los necesarios (número cromático) para colorear un grafo, por lo que generalmente son usados en algoritmos híbridos para realizar los procedimientos de inicialización. Los algoritmos codiciosos más importantes son:

4.6.2.1.1 Algoritmo LF ("largest first"). Este algoritmo inicialmente ordena los vértices según sus grados de manera no creciente, el grado de un vértice es el número de aristas que inciden en el vértice. Con este algoritmo se busca que los vértices que cuentan con más restricciones sean coloreados en primer lugar.

4.6.2.1.2 Algoritmo SL ("smallest last"). El algoritmo SL comienza ordenando los vértices de tal manera que el último sea el de menor grado, el siguiente paso es eliminar dicho vértice y sus aristas y se repite este proceso secuencialmente.

4.6.2.1.3 Algoritmo ID ("incidence degree"). En el algoritmo ID se selecciona arbitrariamente el vértice con mayor grado y este es el primero que se colorea, para elegir el siguiente vértice a colorear se construye dinámicamente un conjunto de vértices candidatos que contiene los que aún no han sido coloreados y tienen al menos un vértice vecino con color. Los vértices en este conjunto se ordenan de forma no creciente según su grado de incidencia (número de vecinos coloreados) y el primer vértice de este conjunto se selecciona para ser coloreado, en caso de empate se selecciona el vértice con mayor número de vecinos sin color.

4.6.2.1.4 Algoritmo DSATUR ("saturation degree"). Este algoritmo funciona de igual forma que el algoritmo ID pero en este caso no se usa el grado de incidencia sino el grado de saturación del vértice, el cual corresponde al número de colores diferentes presentes en los vértices vecinos con color.

4.6.2.1.5 Algoritmo MIS ("maximal independent sets"). Sea c un color posible para colorear los vértices del grafo, $c \in \{1,2,3\ldots\}$ en el algoritmo MIS se asigna primero el color c a todos los vértices posibles antes de asignar el color c+1 en el subgrafo sin colorear resultante. Al vértice con el grado más pequeño se le asigna el color c=1 y este junto con sus vecinos es

eliminado del grafo, este procedimiento es aplicado para los colores siguientes mientras el grafo aún tenga vértices incoloros (Galán, 2017).

4.6.2.1.6 Algoritmo RLF ("recursive largest first"). El algoritmo RLF al igual que el algoritmo MIS asigna primero el color c a todos los vértices posibles antes de asignar el color c+1 y asigna el color c=1 al vértice de mejor grado. En este caso para seleccionar el siguiente vértice a colorear los vértices incoloros se dividen en dos conjuntos: U1 donde se encuentran los vértices que no tienen vecinos coloreados y U2 donde están los vértices que cuentan con uno o más vecinos coloreados, el vértice a colorear es elegido el vértice en U1 con un mayor número de vecinos en U2, en caso de empate se selecciona el vértice que tenga el menor grado. En caso de que el conjunto U1 esté vacío se selecciona el siguiente color y se continúa con el proceso hasta que todo el grafo esté coloreado (Galán, 2017).

4.6.2.2 Metaheurísticas. Las metaheurísticas son técnicas que buscan mejorar los resultados que se tienen al aplicar heurísticas para obtener un mayor rendimiento. Para grafos grandes, las técnicas metaheurísticas producen las mejores soluciones conocidas en cuanto a la minimización de los colores usados, pero entre sus desventajas están la dificultad en la programación y el tiempo elevado que es empleado para su ejecución hasta alcanzar la convergencia. Las principales técnicas de optimización en las cuales se basan las metaheurísticas son: la búsqueda local, algoritmos genéticos, algoritmos de colonia de hormiga, algoritmos meméticos y algoritmos híbridos, dichas técnicas son descritas a continuación.

4.6.2.2.1 Búsqueda local. En esta metaheurística se parte desde una solución inicial la cual puede ser mejorada a través de un método de búsqueda local que se mueve desde la coloración actual hacia una mejorada. En la búsqueda local se definen tres criterios que se

denominan estrategia de búsqueda: un conjunto de soluciones candidatas (espacio de búsqueda), un vecindario y una función que evalúa las soluciones. De manera general se definen cuatro estrategias de búsqueda:

- Estrategia legal: aquí el espacio de búsqueda S contiene todas las coloraciones legales y el objetivo es encontrar una solución en la que se usen la menor cantidad de colores.
- Estrategia legal parcial con k fija: el número de colores k es fijo, en el espacio de búsqueda se tienen todas las k-coloraciones legales y el objetivo es encontrar una solución en la cual se coloreen todos los vértices.
- Estrategia de penalización con k fija: el número de colores k es fijo, el espacio contiene todas las k-coloraciones incluso las que no son legales y el objetivo es encontrar una k-coloración legal.
- Estrategia de penalización: en el espacio de búsqueda están todas las coloraciones incluidas las que no son legales y el objetivo es encontrar una coloración legal que use el menor número de colores posibles. (Galinier & Hertz, 2006)

La búsqueda tabú es el método de búsqueda local más utilizado para resolver los problemas de optimización combinatoria. “La búsqueda tabú es una técnica iterativa de búsqueda local que trata de evitar que las soluciones caigan en óptimos locales. Para esto se utilizan unas estructuras de memoria de corto y largo plazo, acompañadas de criterios de aspiración. En esta técnica en una iteración se pretende pasar de una solución a la mejor solución vecina, sin importar si esta es mejor o peor que la solución actual. El criterio de terminación puede ser un cierto número máximo de iteraciones o un valor de la función por optimizar”. (C. Hjorring, 2009).

Tabucol fue el primer algoritmo de búsqueda tabú propuesto y es uno de los más estudiados y utilizado por los investigadores. A continuación, se describe el funcionamiento del Tabucol mejorado explicado por Galinier y Hertz (2006):

1. Se construye la solución inicial al azar.

2. Considera únicamente movimientos críticos, los movimientos críticos son aquellos que al cambiar de una solución a otra el color de un solo vértice en conflicto es diferente.

3. Considera todos los movimientos legales en lugar de solo una muestra aleatoria, el movimiento crítico que se selecciona es el mejor entre todos los posibles y en caso de empate se selecciona al azar.

4. Un criterio de aspiración elemental es utilizado, en este criterio si se llega a una coloración legal el estado tabú de un movimiento se cancela y la búsqueda puede ser detenida. En otras versiones de Tabucol se han usado criterios de aspiración más complejos.

5. Aumenta el tamaño de la búsqueda tabú proporcional al número de vértices en conflicto, la tenencia tabú d depende de dos parámetros L y λ, $d = L + \lambda F(s)$ donde $F(s)$ es el número de vértices en conflicto en la solución actual.

6. Las estructuras de datos eficientes pueden reducir drásticamente el esfuerzo computacional.

7. Tabucol es una búsqueda tabú básica, sin estrategias a largo plazo como la intensificación o la diversificación.

4.6.2.2.2 Algoritmos evolutivos. Los algoritmos evolutivos se basan en la teoría de la evolución natural de Darwin para solucionar problemas de optimización combinatoria. En estos algoritmos la población pasa por un proceso de evolución en el que se evidencian dos fases: la

primera fase es la de autoadaptación y en esta las soluciones cambian su estructura interna sin tener ninguna interacción con otros miembros de la población, la segunda fase es la de cooperación, en la cual las soluciones de la población actual intercambian información entre ellas para así producir nuevas soluciones que heredan sus mejores atributos (Costa & Hertz, 1997). En esta categoría se encuentran los algoritmos genéticos genético y colonia de hormigas que se explican más adelante.

4.6.2.2.3 Algoritmos genéticos. Los algoritmos genéticos (GA) fueron planteados por John Holland en la década de 1960. El objetivo de Holland no era diseñar algoritmos para solucionar problemas específicos, sino estudiar profundamente el fenómeno de adaptación y desarrollar formas en que los mecanismos de adaptación natural puedan introducirse a los sistemas informáticos. (Mitchell, 1998). Como lo planteó Holland, los algoritmos genéticos constituyen una clase de procedimientos de optimización en los que las poblaciones de soluciones individuales evolucionan por generaciones de una manera inspirada por la evolución y la selección natural, en cada generación el cruce entre padres seleccionados produce una descendencia a la cual se le aplica un operador de mutación con probabilidad Pm, luego de esto un operador elimina los peores individuos para que el tamaño de la población siga siendo el inicial (Fleurent & Ferland, 1996).

Los algoritmos genéticos se han venido aplicando a una variedad de problemas de optimización combinatoria con gran éxito. En los algoritmos genéticos se hace uso de tres tipos de operadores: de selección, de cruce y de mutación.

- Operadores de selección: Se seleccionan los cromosomas de la población para su reproducción. Cuanto más encaja el cromosoma será más probable que se seleccione para reproducirse (Mitchell, 1998).
- Operadores de cruce: Este operador elige aleatoriamente un locus (es el lugar específico del cromosoma donde está localizado un gen u otra secuencia de ADN) e intercambia las posteriores antes y después de ese locus. entre dos cromosomas para crear dos descendientes (Mitchell, 1998).
- Operadores de mutación: La mutación de un individuo ocasiona que alguno de sus genes, por lo general uno sólo, varíe su valor de forma aleatoria.

En algunos casos se presenta que en los algoritmos genéticos no todos los cromosomas de los individuos tengan la misma longitud, por ende, no todos ellos codifican el mismo conjunto de variables. En este caso existen mutaciones adicionales como puede ser añadir un nuevo gen o eliminar uno ya existente.

4.6.2.2.4 Colonia de hormigas. Es una metaheurística que trata un conjunto de técnicas de optimización las cuales están inspiradas en el comportamiento colectivo de alimentación de las hormigas, las cuales tienen la capacidad de encontrar el camino corto entre la fuente del alimento y el nido a través de la comunicación mediante rastros de feromonas artificiales (Sianturi, 2019).

Durante muchos años, los ingenieros han estado interesados en comportamiento de los insectos sociales para definir nuevos modelos de resolución colectiva de problemas. El sistema Ant, desarrollado recientemente por Colorni et al., es un algoritmo evolutivo que tiene sus raíces en el comportamiento colectivo de una hormiga colonia. En la fase de cooperación de un

algoritmo de hormigas cada La solución de la población se examina con el objetivo de actualizar una memoria global haciendo un seguimiento de importantes estructuras de X que han sido explotadas con éxito en el pasado. La fase de autoadaptación utiliza un problema específico método constructivo para crear una nueva población de soluciones sobre la base de la memoria global. Este principio de búsqueda se puede adaptar fácilmente para abordar ATP generales.

4.6.2.2.5 Algoritmo memético. Los algoritmos meméticos son metaheurísticas híbridas que utilizan una búsqueda local (como en las técnicas de seguimiento del gradiente) dentro de un algoritmo basado en población (como los algoritmos evolutivos).

Los orígenes de los algoritmos meméticos (MA) se remontan a finales de los años ochenta, a pesar de que algunos trabajos en décadas anteriores también tienen similares características. Por otra parte, cabe destacar que había técnicas relacionadas como lo eran el recocido simulado (SA) o la búsqueda tabú (TS). En general, estas técnicas hacen uso de heurísticas subordinadas para llevar a cabo el proceso de optimización, motivo por el cual se le otorgó el término 'metaheurísticas' para denominarlas (Moscato & Cotta Porras, 2003).

En MA, la mutación del algoritmo evolutivo (EA) se reemplaza por un algoritmo de búsqueda local. Es muy importante tener en cuenta que la mayor parte del tiempo de ejecución del algoritmo memético se emplea en la búsqueda local. Estas hibridaciones combinan los beneficios de la población y los métodos, que son apropiados para la diversificación por medio de un operador cruzado, y métodos de búsqueda locales, que son mejores para la intensificación. (Moalic & Gondran, 2018).

Cabe destacar, que un MA mantiene constantemente al problema planteado una población de diversas soluciones, las cuales se denominan: agentes. Estos agentes se

interrelacionan entre sı en un marco de competición y de cooperación, de manera muy parecida a lo que ocurre en la naturaleza entre los individuos de una misma especie. Cuando consideramos la población de agentes en su conjunto, esta interacción puede ser estructurada en una sucesión de grandes pasos temporales denominados generaciones (Moscato & Cotta Porras, 2003). La idea principal de los MA está fundamentada en las mejoras individuales de las soluciones agentes que se interrelacionan entre sí en un proceso que contiene fases de cooperación competiciones del tipo poblacional. (Aguilar Imitola & Perez Diaz, 2013).

5. Formulación del modelo matemático

En el modelo matemático desarrollado que se describe a continuación, se tienen en cuenta las siguientes consideraciones:

- Se considera que todos los cursos tienen una intensidad horaria semanal de 4 horas.
- Cada periodo programado hace referencia a una franja de dos horas. Comenzando con horas pares, ejemplo: el primer periodo inicia 6 am y finaliza a las 8 am, el siguiente inicia a las 8 am y finaliza a las 10 am.
- En el modelo solo se programa de lunes a miércoles dado que los cursos programados el lunes se repiten el miércoles, los cursos del martes se repiten el jueves y los cursos del miércoles el viernes, teniendo en cuenta que lo que se programa el miércoles no es lo mismo que lo programado el lunes.
- Teniendo en cuenta que se programa de lunes a miércoles y los periodos corresponden a dos horas, un curso se debe programar una sola vez.

5.1 Subíndices

i: i-ésimo curso a ser programado. $i = \{1, 2, \dots, C\}$

j: j-ésimo salón disponible. $j = \{1, 2, \dots, S\}$

k: k-ésimo profesor disponible. $k = \{1, 2, \dots, P\}$

l: l-ésimo día a programar cursos en la semana. $l = \{1, 2, \dots, 3\}$

m: m-ésimo periodo disponible en un día. $m = \{1, 2, \dots, T\}$

5.2 Parámetros

DP_{klm}: Disponibilidad del profesor k el día l en el periodo m. Toma el valor de 1 si el profesor está disponible y es 0 de lo contrario.

P_{klm}: Penalización por asignar a un profesor k en un día l y periodo m donde notifica no tiene disponibilidad.

Y_{ik}= Capacidad del profesor k para dictar el curso i. Toma el valor de 1 si el curso i puede ser dictado por el profesor k.

5.3 Variables

$$X_{ijklm} = \begin{cases} 1 \text{ Si el curso } i \text{ se dicta en el salón } j \text{ por el profesor } k \text{ el día } l \text{ en el periodo } m \\ 0 \text{ de lo contrario} \end{cases}$$

5.4 Función objetivo

Minimizar la penalidad de asignar un profesor en una franja en la cual no está disponible.

$$\sum_{i=1}^{C}\sum_{j=1}^{S}\sum_{k=1}^{P}\sum_{l=1}^{3}\sum_{m=1}^{T} P_{klm} * X_{ijklm}$$

5.5 Restricciones

R1: Un profesor no puede dictar dos cursos al tiempo

$$\sum_{i=1}^{C}\sum_{j=1}^{S} X_{ijklm} \leq 1 \qquad \forall_k \quad \forall_l \quad \forall_m$$

R2: En un salón no se pueden dictar dos cursos al tiempo

$$\sum_{i=1}^{C}\sum_{k=1}^{P} X_{ijklm} \leq 1 \qquad \forall_j \quad \forall_l \quad \forall_m$$

R3: Para un profesor no programar lunes y miércoles a la misma hora (ya que lo que se programa el lunes se repite para el miércoles, las asignaciones del lunes no se pueden cruzar con las asignaciones del miércoles)

$$\sum_{i=1}^{C} X_{ijklm} + X_{ijk(l+2)m} \leq 1 \qquad \forall_k \quad \forall_j \quad \forall_{l=1} \quad \forall_m$$

R4: Se deben programar todos los cursos

$$\sum_{j=1}^{S}\sum_{k=1}^{P}\sum_{l=1}^{3}\sum_{m=1}^{T} X_{ijklm} = 1 \qquad \forall_i$$

R5: Un profesor debe dictar un curso para el cual esté capacitado

$$\sum_{j=1}^{S}\sum_{l=1}^{3}\sum_{m=1}^{T} X_{ijklm} - Y_{ik} = 0 \qquad \forall_i \quad \forall_k$$

6. Construcción del algoritmo

El algoritmo se construyó en la herramienta Matlab. Se inici´0 definiendo la cantidad de cursos, salones, profesores, los días y los periodos, el tamaño de la población y el número de generaciones.

En el modelo, se implementó un algoritmo memético, para hacer esto, se debe respetar los principios de los algoritmos genéticos en general. La idea de un algoritmo genético es generar una población inicial y de alguna manera tratar de simular lo que pasaría en un proceso natural, mediante selección natural. Definir condiciones y calificar las soluciones de esa manera para que, a través de las generaciones, las mejores soluciones tengan más opción de reproducirse, las peores soluciones vayan eliminándose y al final quede una solución buena.

Al comenzar se genera una población inicial dados los parámetros de cantidad de cursos, salones, profesores, días, periodos y el tamaño de la población. Una vez se genera la población y se corre la programación se obtienen unas medias y unos máximos.

Medias y máximos = La media es el promedio de los puntajes de esa solución. El máximo, es el máximo puntaje de esa solución.

En la parte del memetismo, lo que se hace es recorrer todas las soluciones generadas y mediante el uso de los vecindarios tratar de optimizar las restricciones fuertes, la población se genera de manera aleatoria, como es una matriz muy grande y los valores que interesan son muy pocos, el resultado va a ser mucho más lento, al correr el programa la idea es generar el mejor puntaje posible y que aumente el valor a medida de que va pasando de generación en generación, al final se toma el máximo como la solución al problema.

Como se muestra en la figura 7, se genera la población inicial siguiendo la primera restricción (R1: Un profesor no puede dictar dos cursos al tiempo).

Figura 7

Población inicial

```
1  function [population] = GET_INIT_POPULATION(C, S, P, D, T, pop_size)
2  % Genera la población inicial que cumple con la condicion
3  % R1
4
5  % Se inicializa la población en ceros
6  population = zeros(pop_size, C, S, P, D, T);
7
8  % Se generan valores aleatorios tal que se ajusten a R1
9  for n = 1:pop_size
10     for k = 1:P
11         for l = 1:D
12             for m = 1:T
13                 % Se generan indices aleatorios por C y S
14                 idx_C = randi(C);
15                 idx_S = randi(S);
16                 population(n, idx_C, idx_S, k, l, m) = 1;
17             end
```

Al final de la ejecución, se toma la mejor solución y se mira si cumple o no, si es una solución aceptable o se deben realizar cambios al algoritmo.

Después de generar la población, como se puede observar en la figura 8, se mejora haciendo uso de vecindarios, lo cual consiste en variar un solo parámetro buscando que la

solución sea mejor, si es mejor se reemplaza por ese, y así se va mejorando de a poco, después de que se pasan por esas optimizaciones para cada restricción fuerte que en este caso vendrían siendo N1, N2, N3, N4, N5, se toman las mejores soluciones y a partir de estas generar una nueva solución. Todo esto se hace a través de una función de selección que básicamente las organiza según el puntaje. La idea es ir depurando poco a poco el espacio de las soluciones solamente con las mejores, es decir, las que mejor se adapten al problema, esto se logra realizando una función de ajuste, que es básicamente que tan bueno es el desempeño de la solución.

Figura 8

Simulador del Algoritmo

```
1  function [sol, max_values, means] = SIMULADOR(C, S, P, D, T, pop_size, Yik, Pklm, n_gen)
2
3  % Se genera una población aleatoria de tamaño pop_size
4  population = GET_INIT_POPULATION(C, S, P, D, T, pop_size);
5
6  % Se almacenan los maximos por generacion
7  max_values = zeros(1, n_gen - 1);
8
9  % Se almacena la media de score por generacion
10 means = zeros(1, n_gen - 1);
11
12 % Se itera sobre el numero de iteraciones
13 for n_generation = 1:n_gen
14
15     % Se almacenan los scores por generacion
16     scores = zeros(1, pop_size);
```

6.1 Algoritmos genéticos

Existe dos maneras de optimizar las soluciones, la parte del memetismo que es cuando se realizan estas variaciones y la generación de nuevos individuos.

Para cada individuo generado, hay una probabilidad de mutación, esta mutación se realiza con el objetivo de generar variabilidad, lo que busca hacer la variación es ampliar el dominio de

las soluciones modificando los genes que ya de por si están triunfando, esto no necesariamente significa una mejora en la solución, el objetivo es agregar variedad a las soluciones porque puede que en la generación inmediata se genere incluso una reducción en el puntaje de esa solución, se podría ver como un empeoramiento. Sin embargo, con esta variación agregada la idea es que con el paso de las siguientes generaciones se exploren más ramas, más posibles soluciones, todo esto para evitar una convergencia demasiada rápida o un posible estancamiento en mínimos locales, es decir, que la solución no se vaya solo por un lado especifico, sino que exista posibilidad de que se creen soluciones diferentes que cumplan con los requerimientos, y que incluso puedan ser mejores a las que se tienen.

Para esto, lo que se hace es generar valores de 1 al azar, y ponerlos en la matriz de forma aleatoria, esto puede mejorar o empeorar el puntaje de ese hijo, allí es donde va la variedad, tal como se muestra en la figura 9.

Figura 9

Operador de mutación

```
1  function [result] = MUTATE(sol)
2  % Toma una solución y la altera aplicando el algoritmo de bitflip mutation,
3  % solo muta los indices para los valores que contienen unos.
4  
5  % Cantidad de bits a ser cambiados
6  MUTATION_SIZE = 100;
7  
8  % Se obtienen las dimensiones de la matriz
9  [C, S, P, D, T] = size(sol);
10 
11 % Se generan indices de forma aleatoria
12 idxs = randi(C*S*P*D*T, 1, MUTATION_SIZE);
13 
14 % Se crea una copia de la matriz original
15 result = sol;
16 
17 % Se cambia el digito
```

6.2 Restricciones

6.2.1 R1: Un profesor no puede dictar dos cursos al tiempo

Para cada profesor, por cada día y por cada periodo, la sumatoria entre cursos y salones debe ser mayor que 1 como se muestra en la figura 10. Iterar por cada uno de los profesores, por cada uno de los días y por cada uno de los periodos, para recorrer cada uno de forma individual y comprobar si efectivamente la suma sobre los cursos y los salones efectivamente da mayor o menor que 0.

Por cada profesor, por cada día y por cada periodo, se suma todos los unos que tenga en estas dos dimensiones (cursos y salones) y si la suma es mayor que uno, es decir, si esta condición no se cumple, se aplica una penalización.

Como tal, lo que hacen las funciones con las restricciones es contar las veces que la restricción es violada, y después se penalizan de forma muy severa en la función de ajuste para cumplir con esas restricciones a parte del ajuste por vecindarios.

En este caso se cuenta cuantas veces pasa esto, cuantas veces la suma da mayor que uno, lo que indica esta función R1 es cuantas veces se infringe esa condición por cada profesor, por cada día y por cada periodo.

Figura 10

Restricción 1

```
1  function [value] = R1(sol)
2  % Verifica que la solución cumpla con la restriccion R1
3  % de no cumplir cuenta el numero de incosistencias y las
4  % regresa en value
5
6  % Se obtienen las dimensiones de la matriz
7  [C, S, P, D, T] = size(sol);
8  value=0;
9
10 for k = 1:P
11     for l = 1:D
12         for m = 1:T
13             % Cuenta la cantidad de cursos asignados
14             % a cada salon, dado un profesor, un día y
15             % un periodo determinado.
16             suma = sum(sol(:, :, k, l, m), 'all');
17             if (suma > 1)
```

6.2.1.1 Vecindario N1 optimiza a R1. Al recorrer cada profesor, en cada día, en cada periodo y a raíz de los resultados, se crea un vecindario para poder optimizar la R1, es por esto que este vecindario N1 contiene un k_aux, que pasa a una función que se llama índice de p, el cual es una permutación aleatoria de los índices de p (profesores, días y periodos) como se observa en la figura 11.

El orden de los valores es aleatorio, porque al explorar los vecindarios, como en ellos se van a hacer cambios a la solución, se decide vectorizar para no dar preferencia a los primeros valores de los índices más bajos, es decir, si se recorren en orden se empezaría desde el índice 1 hasta el índice final y siempre seria así, sin embargo, por cuestiones de optimización no se exploran todos los vecindarios posibles para cada solución. Por eso se hacen solo unos cambios,

para no darle preferencia a las soluciones que se encuentran antes en la matriz se crea una aleatorización de los índices, con el objetivo de no crear un sesgo, más que todo por prevención. Aun así, en caso de ser necesario se recorren todos y cada uno de los índices, cada una de las combinaciones, y se accede a esas de forma aleatoria.

Se aplica la restricción 1, se mira si la suma es mayor que uno, se guarda la matriz original y luego se saca el puntaje, se analiza condición a condición y se hace un recuento de los errores y se aplican ciertas penalizaciones basados en unos puntajes que están dados de momento de manera arbitraria.

Figura 11

Vecindario N1 Optimiza a R1

```
1  function [resp] = N1(sol, Yik, Pklm)
2  % Hace una busqueda local para los vecinos segun la restriccion
3  % R1 y selecciona la mejor
4
5  % Se obtienen las dimensiones de la matriz
6  [C, S, P, D, T] = size(sol);
7  resp = sol;
8
9  % Se randomiza el acceso a los indices de la matriz
10 idxs_P = randperm(P);
11 idxs_D = randperm(D);
12 idxs_T = randperm(T);
13 for k_aux = 1:P
14     k = idxs_P(k_aux);
15     for l_aux = 1:D
16         l = idxs_D(l_aux);
17         for m_aux = 1:T
```

6.2.2 R2: En un salón no se pueden dictar dos cursos al tiempo

Para cada salón, por cada día y por cada periodo, la sumatoria de todos los cursos y todos los profesores, debe ser menor que 1, si la condición se infringe se suma un valor a la restricción

y el resultado es cuantas veces infringe esa condición por cada salón, por cada día y por cada periodo como se muestra en la figura 12.

Figura 12

Restricción 2

```
1   function [value] = R2(sol)
2   % Verifica que la solución cumpla con la restriccion R2
3   % de no cumplir cuenta el numero de incosistencias y las
4   % regresa en value
5
6   % Se obtienen las dimensiones de la matriz
7   [C, S, P, D, T] = size(sol);
8   value=0;
9
10  for j = 1:S
11      for l = 1:D
12          for m = 1:T
13              % Cuenta la cantidad de cursos asignados
14              % a cada profesor dado un salon, un día y
15              % un periodo determinado.
16              suma = sum(sol(:, j, :, l, m), 'all');
17              if (suma > 1)
```

6.2.2.1 Vecindario N2 optimiza a R2. Lo que se hace en el vecindario número dos es similar a lo que se hace en el vecindario número uno. Se accede de manera aleatoria a cada salón, a cada día y a cada periodo, tener en cuenta que siempre se realiza de manera aleatoria no en orden.

Es muy similar a lo que se hizo en el vecindario 1, como se indica en la figura 13, se verifica que el salón, el día y el periodo infrinja esta restricción y si la infringe, va a ser un proceso muy similar que es comenzar guardando la solución original, sacando el puntaje y después variando uno a uno todas las posibles combinaciones en el vecindario de soluciones y tomando la mejor solución.

Figura 13

Vecindario N2 Optimiza a R2

```
1   function [resp] = N2(sol, Yik, Pklm)
2   % Hace una busqueda local para los vecinos segun la restriccion
3   % R2 y selecciona la mejor
4
5   % Se obtienen las dimensiones de la matriz
6   [C, S, P, D, T] = size(sol);
7   resp = sol;
8
9   % Se randomiza el acceso a los indices de la matriz
10  idxs_S = randperm(S);
11  idxs_D = randperm(D);
12  idxs_T = randperm(T);
13  for j_aux = 1:S
14      j = idxs_S(j_aux);
15      for l_aux = 1:D
16          l = idxs_D(l_aux);
17          for m_aux = 1:T
```

6.2.3 R3: Para un profesor no programar lunes y miércoles a la misma hora

Se comienza de manera similar como se indica en la figura 14, para cada salón, por cada profesor y por cada periodo, se asigna el lunes como el índice 1 del día y el miércoles como el lunes + 2 días, esto se hace por si en algún momento varia, sea fácil hacer esa variación. Esa condición se debe cumplir para cada profesor, para cada salón y para cada periodo.

Figura 14

Restricción 3

```
function [value] = R3(sol)
% Verifica que la solución cumpla con la restriccion R3
% de no cumplir cuenta el numero de incosistencias y las
% regresa en value

% Se obtienen las dimensiones de la matriz
[C, S, P, D, T] = size(sol);
value=0;
% Indices de los días usados en la verificación
lunes = 1;
miercoles = lunes + 2;

for j = 1:S
    for k = 1:P
        for m = 1:T
            % Cuenta la cantidad de cursos asignados
            % de igual forma tanto el lunes como el
```

6.2.3.1 Vecindario N3 optimiza a R3. El vecindario 3 busca optimizar la condición 3, para cada profesor, para cada salón y para cada periodo como se muestra en la figura 15. El día 1 en este caso es el lunes y para todos los periodos en los que se cumpla, se analiza uno a uno salón, profesor y periodos y por cada uno se analiza esa condición.

Se suman todos los X_{ijklm} y todos los $X_{ijk(l+2)m}$, si esto da mayor que uno se infringe esta condición y de lo contrario no se cuenta.

En el vecindario 3, se hace lo mismo que la permutación para acceder a los índices de manera aleatoria, se toman los lunes y los miércoles, buscando si se incumple la restricción y así pasar a optimizarla.

Se saca el puntaje del horario original y se pasa a recorrer cada uno de los índices, se varía los unos para el miércoles, se va eliminando uno a uno aquellos valores que causan interferencia, así es como se arma el vecindario y así se toma la mejor solución del vecindario.

Se toma la solución original, se cambia el i-ésimo uno por un cero, y esto se le asigna a la respuesta para el valor del miércoles, se pasa a calcular el puntaje general de esa respuesta, si es mejor que la anterior se reemplaza de lo contrario no se reemplaza, y al final se toma la mejor de todas.

Figura 15

Vecindario N3 Optimiza a R3

```
1  function [resp] = N3(sol, Yik, Pklm)
2  % Hace una busqueda local para los vecinos segun la restriccion
3  % R3 y selecciona la mejor
4  % NOTA: No necesariamente elimina un indice que causa interferencia tipo R3
5  % ya que al comparar los vecindarios se toman todos los indices.
6  
7  % Se obtienen las dimensiones de la matriz
8  [C, S, P, D, T] = size(sol);
9  resp = sol;
10 
11 % Se randomiza el acceso a los indices de la matriz
12 idxs_S = randperm(S);
13 idxs_P = randperm(P);
14 idxs_T = randperm(T);
15 
16 lunes = 1;
17 miercoles = lunes + 2;
```

6.2.4 R4: Se deben programar todos los cursos

Como se puede observar en la figura 16, por cada curso se suma todos los valores a través del resto de los ejes y si la sumatoria de alguna manera es diferente de 1 bien sea que de 0 o de mayor que 1 se cuenta como un error.

Figura 16

Restricción 4

```
1  function [value] = R4(sol)
2  % Verifica que la solución cumpla con la restriccion R4
3  % de no cumplir, cuenta el numero de incosistencias y las
4  % regresa en value
5  
6  % Se obtienen las dimensiones de la matriz
7  [C, S, P, D, T] = size(sol);
8  value=0;
9  
10 for i = 1:C
11     suma = sum(sol(i, :, :, :, :), 'all');
12     if (suma ~= 1)
13         % Se toma como errados el exceso o la falta respecto al
14         % valor ideal (1)
15         errados = abs(suma - 1);
16         value = value + errados;
17     end
```

6.2.4.1 Vecindario N4 optimiza a R4. Se toma el puntaje para la función original sin cambios, se toma la matriz original, y se mira dos posibles cambios: el primero, si la suma da cero, se tendría que cambiar uno de los valores, para un índice de 0 a 1 y la otra si la suma da mayor que uno, se tendría que cambiar los índices de 1 a 0 como se muestra en la figura 17.

Si la suma da menor que 1 se cambia el dígito, en vez de un 0 pasa a 1, pero si la suma llega a ser mayor que uno, reemplazo el 1 por un 0, al final lo que va a decir es el dígito que se quiere cambiar.

Una vez se tienen los índices que serán evaluados, se realiza el procedimiento estándar que se venía haciendo antes, variable auxiliar, se saca el puntaje y se reemplaza. Para ese índice, se le asigna el índice de reemplazo seleccionado, dependiendo de que si lo que se quiere es poner un 1 este será un 1 pero si se quiere poner un 0 se define como 0, para los índices.

Figura 17

Vecindario N4 Optimiza a R4

```
1   function [resp] = N4(sol, Yik, Pklm)
2   % Hace una busqueda local para los vecinos segun la restriccion
3   % R4 y selecciona la mejor
4
5   % Se obtienen las dimensiones de la matriz
6   [C, S, P, D, T] = size(sol);
7   resp = sol;
8
9   % Cantidad de indices a evaluar para el vecindario en caso de que la suma
10  % sobre los ejes para R4 sea 0. Este valor debe ser menor a S*P*D*T
11  IDXS_SUM_ZERO = 10;
12
13  % Se randomiza el acceso a los indices de la matriz
14  idxs_C = randperm(C);
15
16  for i_aux = 1:C
17      i = idxs_C(i_aux);
```

6.2.5 R5: Un profesor debe dictar un curso para el cual esté capacitado

Como se muestra en la figura 18, para cada curso y para cada profesor, la sumatoria a través de todos los salones, de todos los días y de todos los periodos, debe ser 0 para estas dos matrices.

Si esta diferencia llega a ser diferente de 0, se incumple esta condición, porque la condición solo se cumple cuando esa diferencia es igual a 0, cuando se incumple la condición se aumenta la cantidad de veces que se infringió esa condición.

Figura 18

Restricción 5

```
function [value] = R5(sol, Yik)
% Verifica que la solución cumpla con la restriccion R5
% de no cumplir, cuenta el numero de incosistencias y las
% regresa en value

% Se obtienen las dimensiones de la matriz
[C, S, P, D, T] = size(sol);
value=0;

for i = 1:C
    for k = 1:P
        % Si el profesor imparte el curso, este valor debe estar en 1
        suma = sum(sol(i, :, k, :, :), 'all');

        % Verifica si el profesor esta capacitado para dictar el curso
        % este valor debe estar en 1
        max_cur = Yik(i, k);
```

6.2.5.1 Vecindario N5 optimiza a R5. Se toman índices de forma aleatoria como se indica en la figura 19, se analiza la condición, si se infringe esa condición hay que optimizarla, se guarda el puntaje inicial, se toma la matriz original, asumiendo que es la mejor, si la diferencia da negativo se tiene que hacer un procedimiento diferente a si la diferencia diera positivo, si la diferencia da negativo, es decir, si el profesor imparte menos cursos de los que puede, se toman índices para posiciones distintas de cero y se escoge la mejor opción para el turno, la idea es asignar un curso a ese profesor y dentro de ese vecindario solo se van a analizar un máximo de índices, como está guardado en esta variable.

Se obtienen índices para posiciones aleatorias dados la cantidad de salones, la cantidad de días y la cantidad de periodos, se tiene que asegurar de que ese índice en efecto corresponda a un cero. Si, por otro lado, esta diferencia da mayor que cero, significa que el profesor está impartiendo más cursos de los que debe, en este caso, se necesita encontrar los unos y cambiarlos

por un cero, se cambia un índice por el índice de reemplazo, si es un uno, el índice de reemplazo va a ser un cero, si es cero el índice de reemplazo va a ser un no, luego se calcula el puntaje, si es mejor se reemplaza sino no se reemplaza y al final se queda con el mejor de todos.

Figura 19

Vecindario N5 Optimiza a R5

```
1  function [resp] = N5(sol, Yik, Pklm)
2  % Hace una busqueda local para los vecinos segun la restriccion
3  % R5 y selecciona la mejor
4
5  % Se obtienen las dimensiones de la matriz
6  [C, S, P, D, T] = size(sol);
7  resp = sol;
8
9  % Se randomiza el acceso a los indices de la matriz
10 idxs_C = randperm(C);
11 idxs_P = randperm(P);
12
13 % Numero de de indices que componen el vecindario N5 en caso de que sea
14 % necesario agregar un 1
15 N_INDEXES = 10;
16
17 for i_aux = 1:C
```

6.3 Función de ajuste o de costo

Esta función, evalúa la solución respecto a todas las restricciones, indica que tan bien o que tan mal trabaja esta función como se puede observar en la figura 20.

Primero se toma una solución a la vez, para esto es necesario tener la matriz que indica que profesor está capacitado para dictar que curso y la matriz que dice la disponibilidad de cada profesor, estos son los pesos que se le dan a cada restricción, esto se hace con el fin de que en el momento de hacer las pruebas, si hay una restricción que no se está cumpliendo se pueda tratar de cambiar estos pesos para darle más importancia a esta restricción y de esa manera darle

preferencia a las soluciones que optimicen esa restricción en particular. Lo mismo se realiza para las restricciones suaves.

Se ve la cantidad de errores respecto a cada uno de los problemas. Una vez se cuenta cuantas veces esa solución en específico infringe cada una de las restricciones hay que ver que tal lo hace respecto a la función objetivo.

Figura 20

Función de ajuste o de costo

```
1  function [value] = FITNESS(sol, Yik, Pklm)
2  % Evalua que tan bien se adapta la solución dada a las restricciones
3  % del problema
4  % PARAMETROS
5  %   sol: La solucion a ser evaluada
6  %   Yik: Matriz con la capacidad de un porfesor k de impartir el curso i
7  %   Pklm: Matriz con la disponibilidad horaria de cada profesor k al dia l
8  %   y periodo m
9
10 % Pesos de cada restriccion
11 p1 = 1;
12 p2 = 1;
13 p3 = 1;
14 p4 = 1;
15 p5 = 1;
16 pSoft = 1;
17
```

6.4 Función objetivo

Se recorren todos los valores, para cada curso y para cada salón, después se suma a través de todos los índices, para cada profesor, para cada día y para cada periodo. Esto se hace por que la función no varía respecto a cursos y salón, sino que solamente varia respecto a profesor, día y periodo, debe cumplirse para todos los cursos y para todos los salones.

Se hace de esta manera para que se analice cada curso y cada salón de manera independiente y así se garantice que se cumple para todos los cursos y para todos los salones. Una vez hecho esto, se toma la solución de cada curso y de cada salón.

Como se muestra en la figura 21, la función objetivo va a contar básicamente esta sumatoria, es decir, cuantas veces se cumple esta condición, entre menos asignaciones al profesor se hagan en este horario de disponibilidad esta función va a ser menor, y entre más se le haga esta función va a ser mayor.

Figura 21

Función objetivo

```
1  function [value] = FUNCION_OBJETIVO(sol, Pklm)
2  % Funcion a minimizar. Relaciona el horario asignado a la disponibilidad de
3  % dicho profesor para dictar clase.
4  
5  [C, S, P, D, T] = size(sol);
6  value = 0;
7  for i = 1:C
8      for j = 1:S
9          temp = squeeze(sol(i, j, :, :, :));
10         value = value + sum(temp(Pklm == 1), 'all');
11     end
12 end
```

6.5 Penalización

Se toma un valor arbitrario debido a que todas las posibles soluciones antes de ser evaluadas sobre las restricciones y aplicar las correspondientes penalizaciones, siempre tienen un valor base implícito, ya sea 0, 1 o el que sea y sobre ese valor se va a sumar, restar, dividir o multiplicar las penalizaciones o los incentivos que se elijan sobre ese valor. Es por esto, que se toma un valor arbitrario porque esta función de costo (FITNESS) se puede hacer orientada de dos maneras:

Una función que diga que el mayor puntaje significa una mejor solución y otro que indique que el menor puntaje significa una mejor solución. La diferencia es el tipo de problema que se está buscando optimizar, si se busca maximizar o minimizar, la manera en la que se va a penalizar cada uno de estos valores será de la siguiente forma: el valor a retomar de esta función de ajuste va a ser: En el numerador se va a tener este valor arbitrario que va a ser un valor muy grande y se restará esta función objetivo multiplicado por el peso de la función objetivo, esto debido a la manera en que se trató de abordar las restricciones fuertes el cual fue penalizándolas mucho más al hacer que la suma de estas fueran un cociente del valor inicial de la solución.

Si esta función objetivo es mayor significa que esta resta va a dar un menor resultado y, por lo tanto, el puntaje general de fitness va a dar un menor valor.

6.6 Two points crossover

Tomar dos soluciones que van a servir a modo de padres y a partir de ellos, sacar soluciones hijo como se observa en la figura 22. La idea es que estas soluciones hijo se conviertan en buenas soluciones al compartir los mejores genes de los padres.

Se generan hijos y las peores soluciones van siendo reemplazadas, pero esto va a terminar hasta que llegue a un límite, este límite está dado tal que, no llegue a reemplazar todas las soluciones porque se quiere conservar las mejores soluciones de la generación pasada. Esto en caso de que los hijos no generen soluciones tan buenas, como los padres. La idea es no reemplazar todos los valores malos por las nuevas generaciones, sino dejar los mejores que continúen sobreviviendo (ELITISMO).

Figura 22

Crossover

```
1  function [new_born1, new_born2] = DOUBLE_POINT_CROSSOVER(parent1, parent2)
2  % Toma 2 puntos entre los padres y los mezcla para dar lugar a 2 crias
3
4  % Se extraen las dimensiones de ambos padres (son las mismas)
5  [C, S, P, D, T] = size(parent1);
6
7  % El factor para el corte de los elementos de los padres
8  CUT_FACTOR = 0.3;
9
10 % Se calcula el numero de elementos de los padres
11 max_length = C*S*P*D*T;
12
13 % Se calculan los indices para la particion
14 cuts = [round(max_length * CUT_FACTOR) round(max_length * 2 * CUT_FACTOR)];
15
16 % Se inicializa la nueva generacion
17 new_born1 = zeros(C, S, P, D, T);
```

Se toma un par de padres y a partir de esto se saca un par de hijos.

Primero se obtienen los padres, mediante una selección basada en probabilidad como se observa en la figura 23, de tal manera que los padres con mayor puntaje tengan mayor probabilidad de aparearse que los individuos con peor puntaje, la idea es ir seleccionando las mejores soluciones.

Lo primero que hay que hacer, es una lista ordenada de todas las soluciones, clasificarlos de mejor a peor, eso es lo que se hace en esa función de selección, se comienza a obtener todas las dimensiones de la matriz original, sumado a la cantidad de individuos de la población, después de eso se va recorriendo individuo a individuo, cabe destacar que se va guardando el índice de cada individuo y su puntaje.

Se toma la solución para el índice, se saca el puntaje para la solución y se almacena. Una vez están guardados todos los índices de estas soluciones se genera un vector con base en el eje 1, a la columna uno y eso básicamente los va a ordenar basado en su puntaje.

Figura 23

Función de selección

```
1  function [ordered_vector] = SELECTION_FUNCTION(population, Yik, Pklm)
2  % Recive un vector de posibles soluciones y las ordena en orden descendente
3  % segun su puntaje dado por la función de ajuste (FITNESS)
4  % Regresa un vector columna con los indices y sus respectivos puntajes
5  % en orden descendente.
6
7  % Se obtienen las dimensiones de la matriz original
8  [N, C, S, P, D, T] = size(population);
9
10 % Se inicializa el vector con el tamaño de la población
11 vec = zeros(N, 2);
12
13 for idx = 1:N
14
15     % Se obtiene un elemento de la poblacion
16     matrix = squeeze(population(idx, :, :, :, :, :));
```

Una vez se tenga la lista de índices y puntajes ordenados se obtienen los porcentajes asociados a cada individuo como se indica en la figura 24.

Primero se suman todos los puntajes para obtener un total, para mirar que tanto puntaje tiene una solución respecto al total de los puntajes, si una puntuación es muy grande respecto a las otras va a tener una fracción mayor, va a tener una probabilidad mayor, básicamente se quiere asignar una probabilidad basada en el puntaje de esa solución, comparándola con el total de puntajes.

Figura 24

Función para obtener los porcentajes

```
function [prob_list] = GET_PERCENTAGES(fit_list)
% Obtiene una matriz (Nx2) con los puntajes y los indices de la poblacion y
% regresa la matriz pero con porcentajes en vez de puntajes.

totals = sum(fit_list, 1);
total_score = totals(1);

% Se obtienen los valores de probabilidad para cada elemento
prob_list = fit_list(:, 1)./total_score;
end
```

Una vez se tiene la lista de probabilidades, se selecciona un índice basado en la probabilidad de cada uno como se muestra en la figura 25. Esto se hace de la siguiente manera:

Inicialmente, se hace una función de probabilidad acumulada y se genera un número aleatorio entre 0 y 1, a partir de eso, se obtiene un índice, esta cambia la lista de probabilidades para guardar en cada índice la suma acumulada que equivale el índice.

Figura 25

Operador de Selección de padres

```
function [p_idx1, p_idx2] = GET_PARENTS_IDX(population, Yik, Pklm)
% Selecciona el indice de los siguientes padres basado en la probabilidad
% de reproduccion tomada de la selección de los miembros mas aptos de la
% población.

% Obtiene una lista ordenada de forma descendente de los elementos de la
% población segun puntaje e indice en la población original.
fit_list = SELECTION_FUNCTION(population, Yik, Pklm);

% Se obtiene la lista con las probabilidades
prob_list = GET_PERCENTAGES(fit_list);

% Se selecciona un indice basado en la probabilidad
idx1 = sum(rand >= cumsum(prob_list)) + 1;
p_idx1 = fit_list(idx1, 2);
```

6.7 Mutación

Como se puede observar en la figura 26, para una cantidad de índices se genera índices aleatorios y para todos esos índices que sean iguales a 0 se cambia a 1 y para todos los índices que sean 1 se cambia a 0. La idea es girar los posibles valores de manera aleatoria con el fin de generar variedad en las soluciones.

Se considera una cantidad de bits significativa, la idea de ese parámetro es básicamente cambiar de 0, la cantidad de bits que cambian su valor ya sea de 0 a 1 o de 1 a 0, debido a la naturaleza del problema y como están formuladas las restricciones, si se cambian muchos 0 a 1 de repente, esa solución va a ser penalizada porque las soluciones o donde convergen son soluciones donde hay pocos 1 en su mayoría hay 0, entonces no se puede cambiar de repente muchos valores de 0 a 1. Aun así, el valor seleccionado de los bits, se considera que generan un cambio pequeño pero significativo respecto a la cantidad de 1 por cada solución, o a las soluciones que convergen después del análisis por vecindarios y demás.

Luego se ordenan las soluciones, se toma el último índice de las soluciones, este se toma por el tamaño de la población, puesto que están ubicados los valores de manera descendente, es decir, de mayor a menor, y del anterior al último índice, como se tienen dos hijos estos van a reemplazar estas dos soluciones. Al final se devuelve la mejor solución obtenida para todo el algoritmo.

Figura 26

Operador de mutación

```
1  function [result] = MUTATE(sol)
2  % Toma una solución y la altera aplicando el algoritmo de bitflip mutation,
3  % solo muta los indices para los valores que contienen unos.
4
5  % Cantidad de bits a ser cambiados
6  MUTATION_SIZE = 100;
7
8  % Se obtienen las dimensiones de la matriz
9  [C, S, P, D, T] = size(sol);
10
11 % Se generan indices de forma aleatoria
12 idxs = randi(C*S*P*D*T, 1, MUTATION_SIZE);
13
14 % Se crea una copia de la matriz original
15 result = sol;
16
17 % Se cambia el digito
```

7. Validación del algoritmo

Para la validación del algoritmo se realizaron 2 corridas de este con datos aleatorios en el software MATLABR22a, los datos utilizados se encuentran en el apéndice B. Se realizó un diseño experimental para determinar el valor de los factores en la implementación del algoritmo en la programación de horarios de la Escuela de Estudios Industriales y Empresariales, las variables respuesta que se analizaron fueron: el valor de la función de aptitud de la solución que en el algoritmo se denomina máximo y el tiempo de ejecución.

7.1 Diseño factorial

Se seleccionó un diseño factorial 2^k, con 3 factores k y dos niveles para cada factor, en la tabla 2 se muestran los factores y niveles que se tuvieron en cuenta.

Tabla 2

Factores y niveles del diseño experimental

Factores	Nivel bajo (-)	Nivel alto (+)
Tamaño de la población (A)	20	40
Número de generaciones (B)	10	30
Porcentaje de mutación (C)	10%	50%

En la tabla 3 se muestra el diseño del experimento obtenido de la herramienta Minitab.

Tabla 3

Matriz del diseño del experimento

Corrida	A	B	C
1	-	-	-
2	-	+	-
3	+	-	-
4	-	-	+
5	+	+	+
6	-	+	+
7	+	+	-
8	+	-	+

7.1.1 Análisis de varianza primera instancia

El análisis de varianza para el máximo (valor de la función objetivo) de la primera instancia se muestra en la tabla 4.

Tabla 4

Análisis de varianza para el máximo la primera instancia

Fuente	GL	SC Ajust.	MC Ajust.	Valor F	Valor p
Modelo	7	0.020000	0.002857	*	*
Lineal	3	0.005000	0.001667	*	*
Tamaño de la población	1	0.000000	0.000000	*	*
Número de generaciones	1	0.005000	0.005000	*	*
Porcentaje de mutación	1	0.000000	0.000000	*	*
Interacciones de 2 términos	3	0.010000	0.003333	*	*
Tamaño de la población*Número de generaciones	1	0.005000	0.005000	*	*
Tamaño de la población*Porcentaje de mutación	1	0.000000	0.000000	*	*
Número de generaciones*Porcentaje de mutación	1	0.005000	0.005000	*	*
Interacciones de 3 términos	1	0.005000	0.005000	*	*
Tamaño de la población*Número de generaciones*Porcentaje de mutación	1	0.005000	0.005000	*	*
Error	0	*	*		
Total	7	0.020000			

Figura 27

Diagrama de Pareto de los efectos para el máximo primera instancia

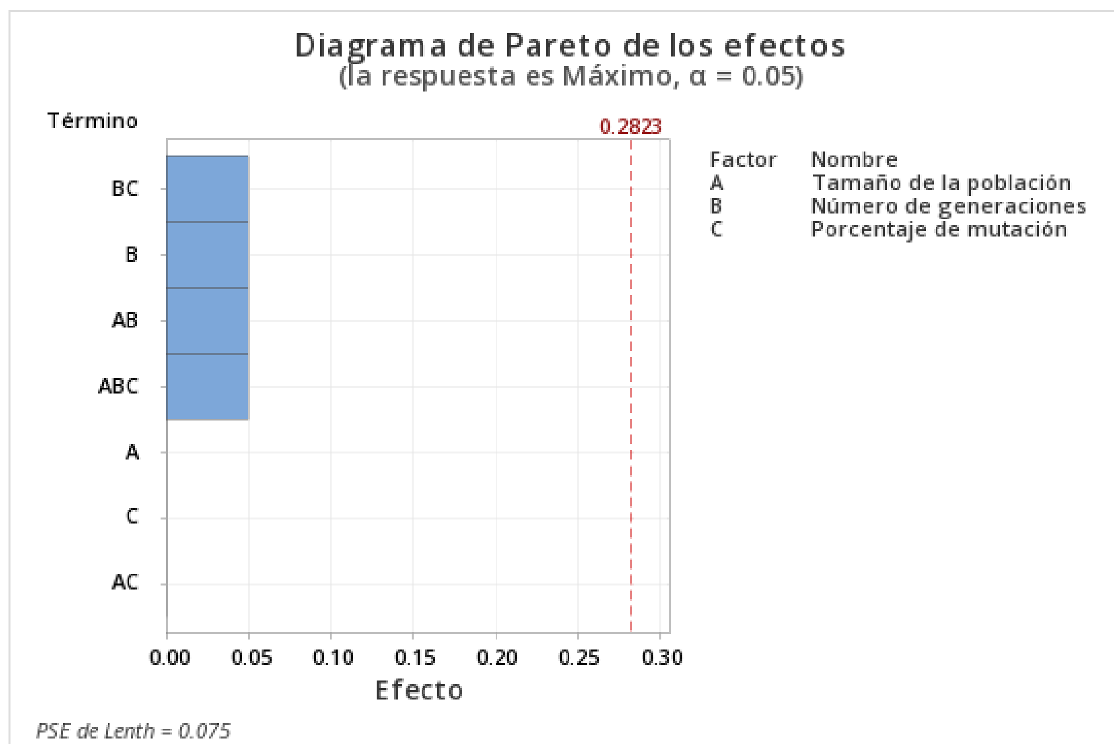
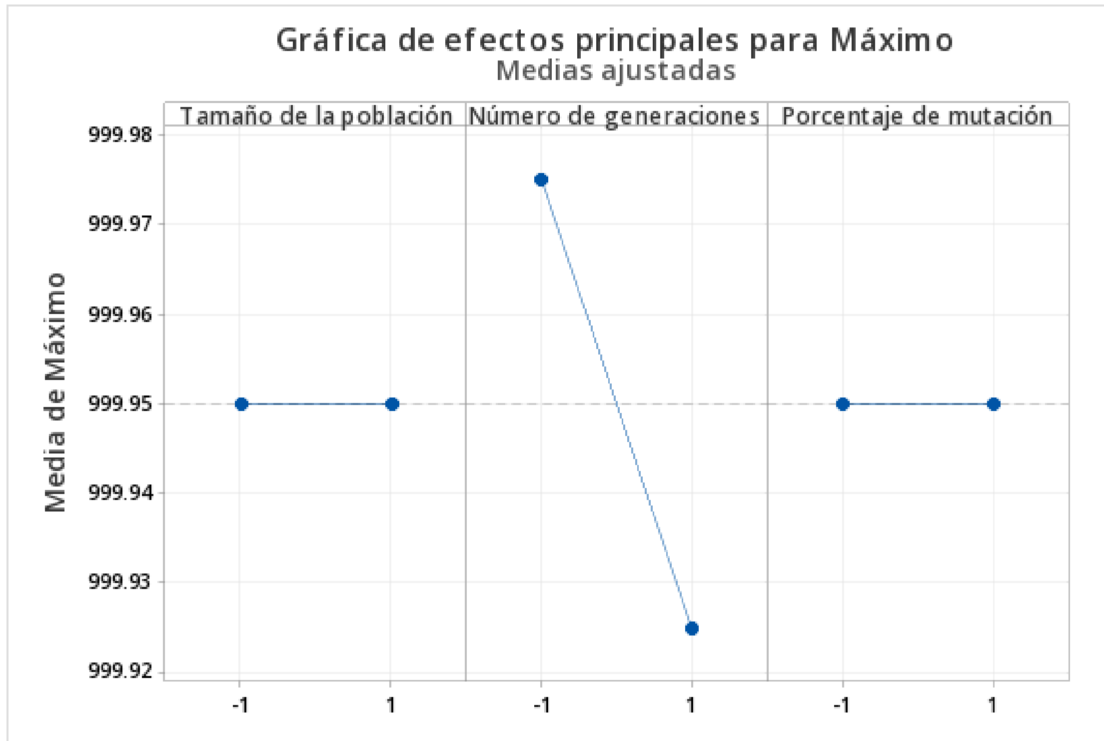


Figura 28

Diagrama de efectos principales para el máximo primera instancia



En las figuras 27 y 28 se puede observar que los factores no tienen un efecto significativo en la función de aptitud de la solución del problema. Ya que para la primera instancia se usa una pequeña cantidad de datos, se puede concluir que al tener menos datos es indiferente el valor que se le asigne el tamaño de la población, el número de generaciones y el porcentaje de mutación.

El análisis de varianza para el tiempo de ejecución del algoritmo de la primera instancia se muestra a continuación en la tabla 5.

Tabla 5

Análisis de varianza de la para el tiempo de ejecución primera instancia

Fuente	GL	SC Ajust.	MC Ajust.	Valor F	Valor p
Modelo	7	23136.8	3305.3	*	*
Lineal	3	21138.5	7046.2	*	*
Tamaño de la población	1	9193.7	9193.7	*	*
Número de generaciones	1	11944.6	11944.6	*	*
Porcentaje de mutación	1	0.2	0.2	*	*
Interacciones de 2 términos	3	1964.3	654.8	*	*
Tamaño de la población*Número de generaciones	1	1935.4	1935.4	*	*
Tamaño de la población*Porcentaje de mutación	1	2.6	2.6	*	*
Número de generaciones*Porcentaje de mutación	1	26.4	26.4	*	*
Interacciones de 3 términos	1	34.0	34.0	*	*
Tamaño de la población*Número de generaciones*Porcentaje de mutación	1	34.0	34.0	*	*
Error	0	*	*		
Total	7	23136.8			

Figura 29

Diagrama de Pareto de los efectos para el tiempo de ejecución primera instancia

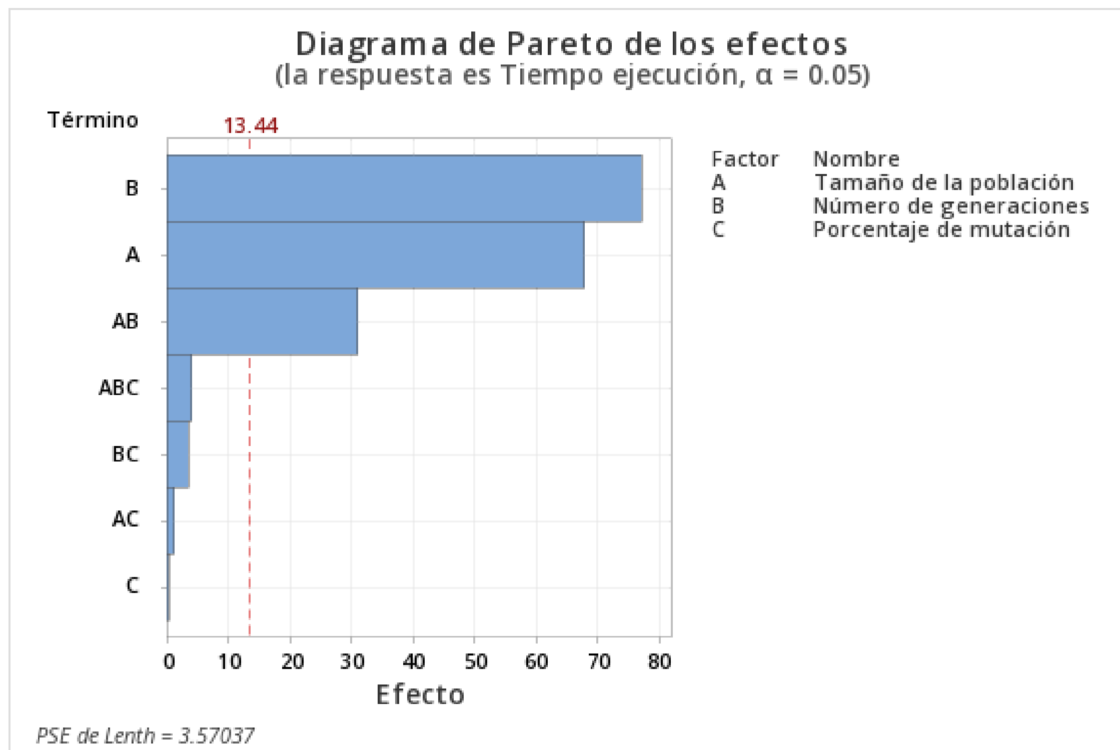
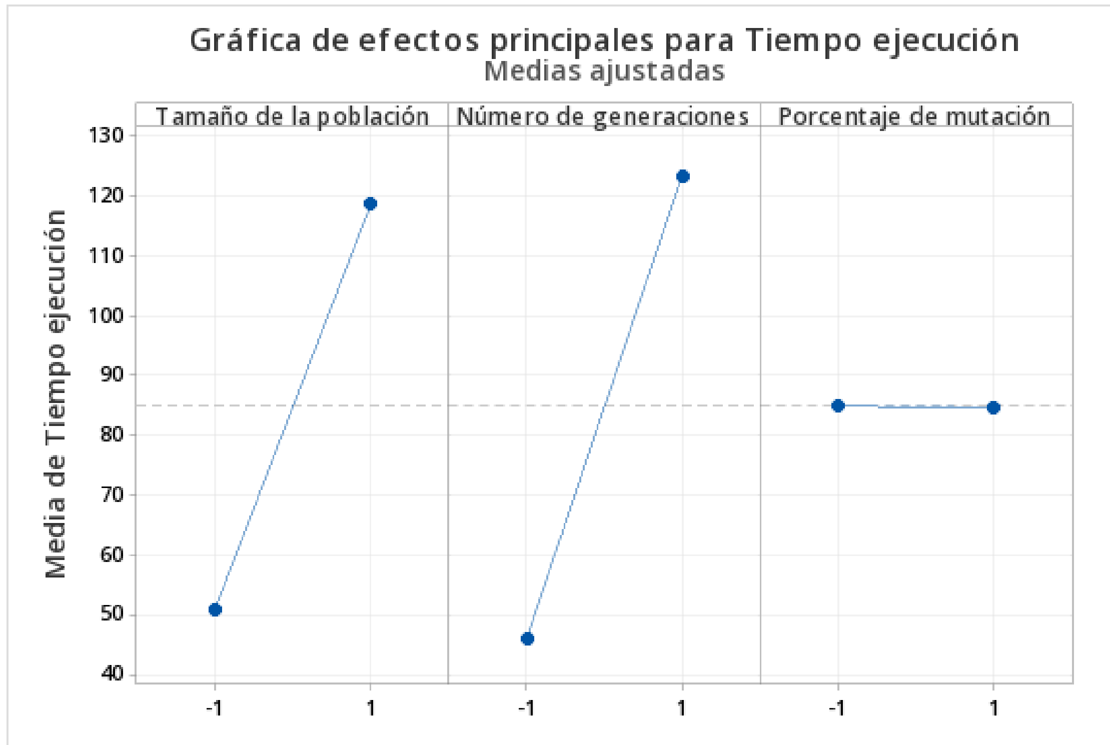


Figura 30

Diagrama de efectos principales para el tiempo de ejecución primera instancia



De las figuras 29 y 30 se puede concluir que los factores que tienen efecto significativo en el tiempo de ejecución del algoritmo son el número de generaciones, el tamaño de la población y el tamaño de la población*número de generaciones, al aumentar dichos factores se aumenta el tiempo de ejecución del algoritmo, además se observa que no hay un efecto significativo entre el porcentaje de mutación y el tiempo de ejecución.

7.1.2 Análisis de varianza segunda instancia

En la tabla 6 se presenta el análisis de varianza para el máximo (valor de la función objetivo) de la segunda instancia.

Tabla 6

Análisis de varianza de la para el máximo segunda instancia

Fuente	GL	SC Ajust.	MC Ajust.	Valor F	Valor p
Modelo	7	164.470	23.496	*	*
Lineal	3	154.417	51.472	*	*
Tamaño de la población	1	1.735	1.735	*	*
Número de generaciones	1	147.588	147.588	*	*
Porcentaje de mutación	1	5.094	5.094	*	*
Interacciones de 2 términos	3	8.364	2.788	*	*
Tamaño de la población*Número de generaciones	1	1.689	1.689	*	*
Tamaño de la población*Porcentaje de mutación	1	1.735	1.735	*	*
Número de generaciones*Porcentaje de mutación	1	4.940	4.940	*	*
Interacciones de 3 términos	1	1.689	1.689	*	*
Tamaño de la población*Número de generaciones*Porcentaje de mutación	1	1.689	1.689	*	*
Error	0	*	*		
Total	7	164.470			

Figura 31

Diagrama de Pareto de los efectos para el máximo segunda instancia

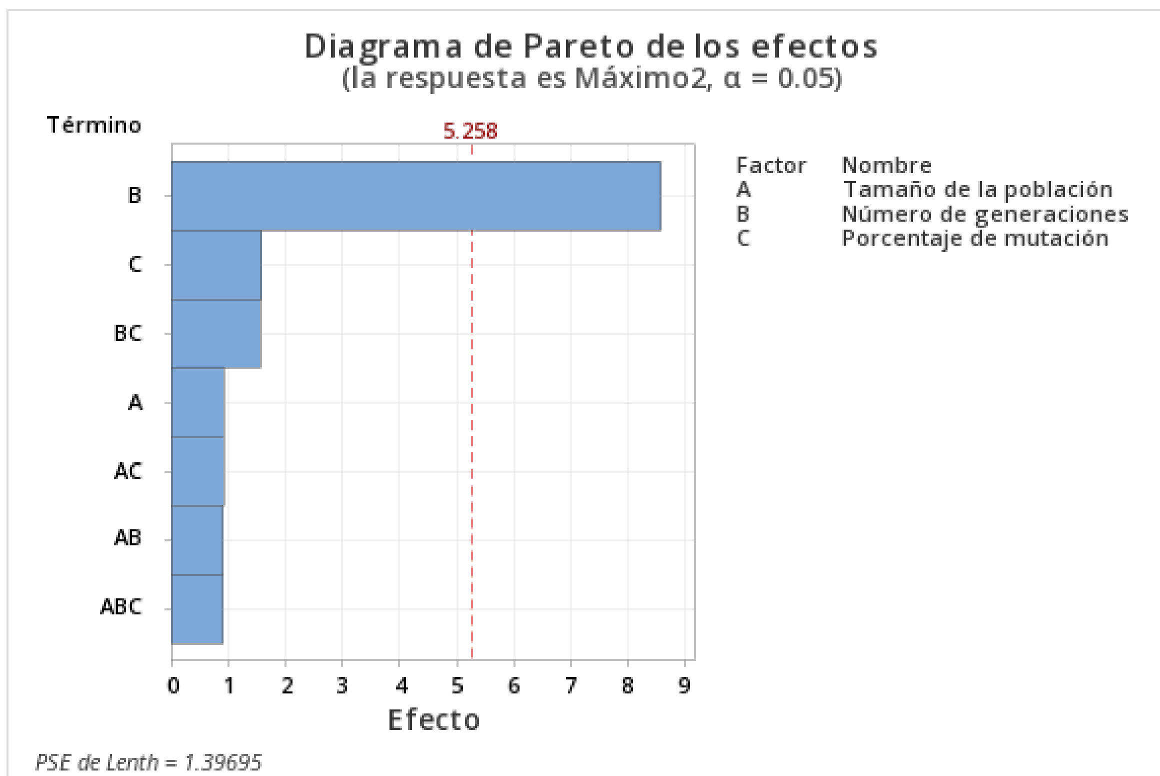
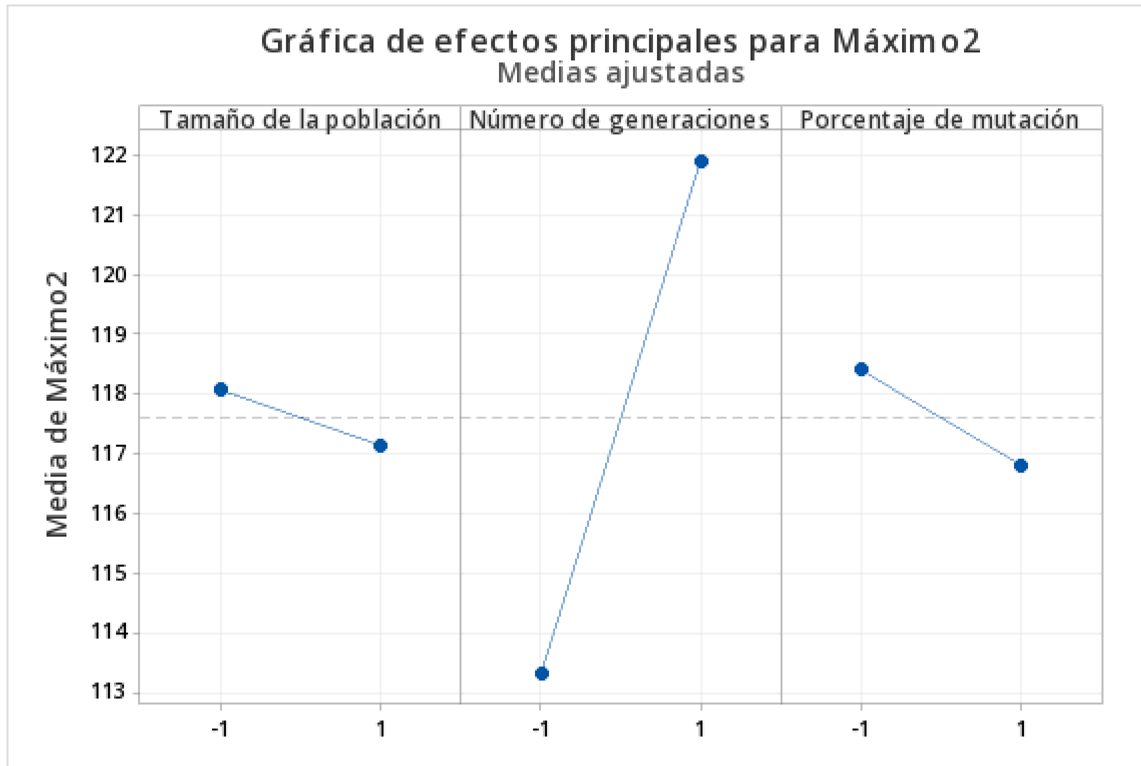


Figura 32

Diagrama de efectos principales para el máximo segunda instancia



De las figuras 31 y 32 se concluye que el número de generaciones tiene un efecto significativo en el valor de la función de aptitud de la solución, al utilizar un número mayor de generaciones se obtiene un mejor valor en la función de aptitud.

En la tabla 7 se presenta el análisis de varianza para el tiempo de ejecución de la segunda instancia.

Tabla 7

Análisis de varianza de la para el tiempo de ejecución segunda instancia

Fuente	GL	SC Ajust.	MC Ajust.	Valor F	Valor p
Modelo	7	284225883	40603698	*	*
Lineal	3	264379184	88126395	*	*
Tamaño de la población	1	145253860	145253860	*	*
Número de generaciones	1	116733333	116733333	*	*
Porcentaje de mutación	1	2391992	2391992	*	*
Interacciones de 2 términos	3	17339178	5779726	*	*
Tamaño de la población*Número de generaciones	1	9533689	9533689	*	*
Tamaño de la población*Porcentaje de mutación	1	3404116	3404116	*	*
Número de generaciones*Porcentaje de mutación	1	4401372	4401372	*	*
Interacciones de 3 términos	1	2507521	2507521	*	*
Tamaño de la población*Número de generaciones*Porcentaje de mutación	1	2507521	2507521	*	*
Error	0	*	*		
Total	7	284225883			

Figura 33

Diagrama de Pareto de los efectos para el tiempo de ejecución segunda instancia

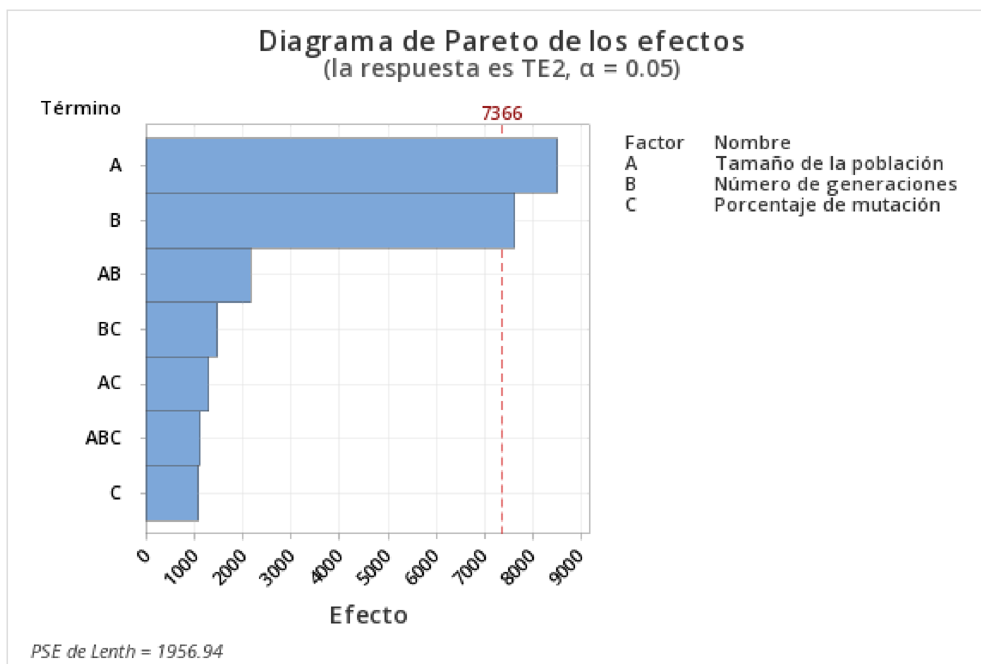
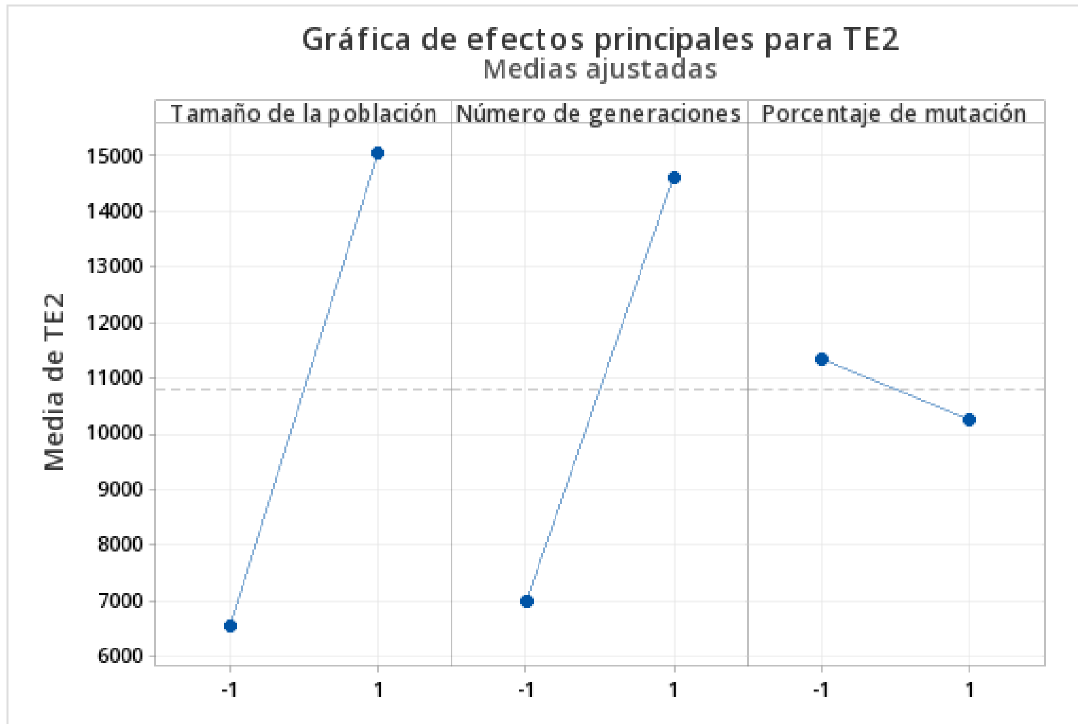


Figura 34

Diagrama de efectos principales para el tiempo de ejecución segunda instancia



Como se puede observar en las figuras 33 y 34, el tamaño de la población y el número de generaciones tienen un efecto significativo en el tiempo de ejecución, al aumentar su valor se aumenta de manera significativa el tiempo de ejecución.

Al analizar los resultados del diseño factorial y teniendo en cuenta que el tiempo de ejecución es importante a la hora de realizar una programación de horarios los factores para la implementación del algoritmo para el caso de estudio se definen de la siguiente manera: tamaño de la población igual a 20 individuos, número de generaciones igual a 30 y porcentaje de mutación del 50%.

8. Implementación del algoritmo

Al validar el algoritmo se pudo observar que al aumentar la cantidad de datos se aumenta de manera significativa el tiempo de ejecución por lo cual para la implementación del algoritmo en la Escuela de Estudios Industriales y Empresariales se decidió no correr el programa con los datos completos sino dividirlos en 4 subproblemas ya que al intentar correr con los datos completos el algoritmo tardaría días en ejecución se mantuvo el algoritmo corriendo durante tres días pero en este tiempo no llegó a finalizar su ejecución.

Los datos completos se pueden observar en el apéndice D y los datos divididos en los 4 subproblemas se encuentran en el apéndice E, lo que se hizo fue partir de los datos completos y dividir el número de cursos, al tener menos cursos disminuye el número de profesores, los salones disponibles se dividieron entre los 4 subproblemas y los días y periodos se mantienen con el mismo valor. En el apéndice F se encuentra la solución obtenida para cada uno de los 4 subproblemas y la programación completa al unir las soluciones.

Para el caso puntual de la escuela se tuvieron en cuenta las siguientes consideraciones:

- La asignatura seguridad y salud en el trabajo tiene una intensidad horaria de 3 horas que son programadas de manera continua, la hora adicional debe ser ajustada de manera manual. Teniendo en cuenta la matriz de disponibilidad del profesor y la disponibilidad de salones para evitar conflictos.
- Ya que los periodos corresponden a dos horas y comienzan en horas pares al tener profesores que tienen disponibilidad horaria que comienza en horas impares se decide asignar el

valor de 1 a los dos periodos de tiempos que contienen la hora disponible del profesor. Ejemplo: si la disponibilidad del profesor es desde las 17:00 a las 19:00, en la matriz de disponibilidad asigno el valor 1 al periodo 6 y 7.

- En el algoritmo lo que se programa el lunes se repite el miércoles, lo programado el martes el jueves y los cursos del miércoles se repiten el viernes, pero hay profesores que no tienen disponibilidad de esta manera sino en días diferentes, en este caso se realiza una excepción y la repetición del curso el siguiente día de la semana se realiza de manera manual teniendo en cuenta las restricciones.

8.1 Evaluación de los resultados

La tabla 8 recopila los resultados obtenidos para cada subproblema, se tiene el valor de la función de aptitud obtenido y el tiempo de ejecución.

Tabla 8

Resultados de los subproblemas

Subproblema	Valor de la función de aptitud	Tiempo de ejecución (s)
1	76,3359	5755,792
2	172,3793	4943,447
3	50,4989	8657,621
4	178,5536	2891,370

En la tabla 9 se realiza un análisis del porcentaje de cumplimiento en el número de cursos asignados a cada profesor para el primer subproblema, teniendo en cuenta que hay un ideal en el

número de cursos a asignar, el ideal es 100% si el porcentaje es mayor de 100 indica que se le asignan más cursos de los ideales y si es menor es porque se le asigna una cantidad menor.

Tabla 9

Cumplimiento en asignación de cursos por profesor subproblema 1

Profesor	#Cursos ideal	#Cursos programados	% Cumplimiento
1	3	4	133%
2	2	3	150%
3	2	1	50%
4	2	2	100%
5	2	1	50%
6	1	2	200%
7	4	4	100%
8	3	4	133%
9	3	5	167%
10	2	2	100%
11	2	1	50%
12	4	3	75%
13	3	3	100%
14	4	5	125%
15	4	1	25%
		Promedio	104%

En la tabla 10 se analiza el cumplimiento en la disponibilidad de los profesores en cuanto a días para el primer subproblema. Lo que se hace es comparar los días que tienen disponibles

contra los días en los que el algoritmo les asigna cursos y se asigna un porcentaje de acuerdo a esto.

Tabla 10

Cumplimiento en los días programados por profesor subproblema 1

Profesor	Días ideal	Días programados	% cumplimiento
1	2	1,2,3	33%
2	1,2,3	1,2,3	100%
3	1,2	2	50%
4	2	1,2	50%
5	1	2	0%
6	2	2,3	50%
7	1,2,3	1,2,3	100%
8	1,2	1,2,3	67%
9	1,2	2,3	50%
10	1,2	1	50%
11	1,2	2	50%
12	1,2	2,3	50%
13	1,2	1,3	50%
14	2	1,3	0%
15	1,2	2	50%
		Promedio	50%

Para el siguiente análisis se tienen en cuanta solo los profesores cuyo porcentaje de cumplimiento en los días programados es mayor a 0 ya que a estos profesores son a los que se les respeta al menos un día su disponibilidad. En la tabla 11 se compara el periodo en el cual el

profesor está disponible contra el periodo en el cual le fue asignado curso por el algoritmo y se asigna un porcentaje de cumplimiento.

Tabla 11

Cumplimiento en los periodos por profesor subproblema 1

Profesor	Periodos ideal	Periodos programados	% cumplimiento
1	1,2,3,4	6	0%
2	5 4 2	1 1 1	0%
3	5	1	0%
4	2,3	7	0%
6	2	6	0%
7	5,6 7 2,4	4 2,3 7	0%
8	1 1	2 5	0%
9	2,3	4	0%
10	1	2,6	0%
11	1	4	0%
12	3,5,6	2	0%
13	5,6	2,4	0%
15	1,4	7	0%
		Promedio	0%

En las tablas 12, 13 y 14 que se muestran a continuación, se presentan el cumplimiento en asignación de cursos, de días y de periodos programados por profesor para el subproblema 2.

Tabla 12

Cumplimiento en asignación de cursos por profesor subproblema 2

Profesor	#cursos ideal	#cursos programados	% cumplimiento
1	2	3	150%
2	1	2	200%
3	2	2	100%
4	1	1	100%
5	3	3	100%
6	1	1	100%
7	2	2	100%
8	4	5	125%
9	2	2	100%
10	5	4	80%
11	1	1	100%
12	2	1	50%
13	1	1	100%
14	1	1	100%
15	3	3	100%
16	1	1	100%
17	2	2	100%
18	2	1	50%
19	1	1	100%
20	1	1	100%
		Promedio	103%

Tabla 13

Cumplimiento en los días programados por profesor subproblema 2

Profesor	Días ideal	Días programados	% cumplimiento
1	1	1,2,3	33%
2	2	1,2	50%
3	2	1,2	50%
4	3	2	0%
5	2	3	0%
6	1	1	100%
7	2	1,3	0%
8	1,2	1,2,3	67%
9	1	2	0%
10	1,2	1,2,3	67%
11	1	2	0%
12	3	1	0%
13	1	3	0%
14	1	3	0%
15	1,2	1,2,3	67%
16	1	3	0%
17	2	1,3	0%
18	1,2	1	50%
19	1	3	0%
20	2	2	100%
		Promedio	29%

Tabla 14

Cumplimiento en los periodos por profesor subproblema 2

Profesor	Periodos ideal	Periodos programados	% cumplimiento
1	5,6	1	0%
2	2	3	0%
3	3,7	2	0%
6	1	1	100%
8	5,6 3,5	7 1,7	0%
10	1,2,3 1	4,5 4	0%
15	5,6 7	4 5	0%
18	3	1	0%
20	5	6	0%
		Promedio	11%

Las tablas 15, 16 y 17 muestran el cumplimiento en asignación de cursos, días y periodos programados por profesor para el subproblema 3.

Tabla 15

Cumplimiento en asignación de cursos por profesor subproblema 3

Profesor	#cursos ideal	#cursos programados	% cumplimiento
1	3	3	100%
2	2	3	150%
3	3	3	100%
4	1	1	100%
5	4	3	75%

Continuación tabla 15

6	1	1	100%
7	3	2	67%
8	2	3	150%
9	3	3	100%
10	2	2	100%
11	2	3	150%
12	1	3	300%
13	1	1	100%
14	2	1	50%
15	2	2	100%
16	2	2	100%
17	2	2	100%
18	2	2	100%
		Promedio	113%

Tabla 16

Cumplimiento en los días programados por profesor subproblema 3

Profesor	Días ideal	Días programados	% cumplimiento
1	1,3	1,2	50%
2	1,2,3	1,2,3	100%
3	1,2	2,3	50%
4	2	2	100%
5	1,2	1,3	50%
6	3	3	100%
7	2,3	1,2	50%

Continuación tabla 16

8	2	1,2	50%
9	1,2	1,2	100%
10	1,2	2,3	50%
11	2	1,3	0%
12	2	1,2,3	33%
13	1	1	100%
14	2	3	0%
15	1,2	2,3	50%
16	2,3	2	50%
17	1,2	3	0%
18	1,2	2	50%
		Promedio	55%

Tabla 17

Cumplimiento en los periodos por profesor subproblema 3

Profesor	Periodos ideal	Periodos programado	% cumplimiento
1	7	4	0%
2	4,7 4,5 4	1 2 6	0%
3	2,3	4,6	0%
4	6,7	7	100%
5	2	4	0%
6	1	2	0%
7	2,7	3	0%
8	2,3	5,6,7	0%

Continuación tabla 17

9	3 2,3	2,7 5,6	0%
10	2	1	0%
12	7	3	0%
13	7	5	0%
15	6	5	0%
16	4	2,3	0%
18	1	7	0%
		Promedio	7%

En las tablas 18, 19 y 20 se presenta el cumplimiento en asignación de cursos, días y periodos programados por profesor para el subproblema 4.

Tabla 18

Cumplimiento en asignación de cursos por profesor subproblema 4

Profesor	#cursos ideal	#cursos programados	% cumplimiento
1	2	1	50%
2	1	2	200%
3	1	2	200%
4	1	1	100%
5	3	3	100%
6	1	1	100%
7	2	2	100%
8	1	1	100%
9	1	2	200%
10	5	4	80%

Continuación tabla 18

11	1	1	100%
12	1	2	200%
13	2	2	100%
14	2	2	100%
15	2	1	50%
16	3	4	133%
17	2	1	50%
18	2	2	100%
19	2	1	50%
20	1	1	100%
		Promedio	111%

Tabla 19

Cumplimiento en los días programados por profesor subproblema 4

Profesor	Días ideal	Días programados	% cumplimiento
1	1,2	3	0%
2	3	2	0%
3	2	1,2	50%
4	2	2	100%
5	1,2	1,2,3	67%
6	2	1	0%
7	1,2	2,3	50%
8	1	2	0%
9	1	1,2	50%
10	1,2	1,3	50%

Continuación tabla 19

11	1	3	0%
12	2	3	0%
13	1	1	100%
14	1,2	1	50%
15	2	1	0%
16	1,2	1,3	50%
17	2	2	100%
18	2	2	100%
19	1,2	1	50%
20	2	3	0%
		Promedio	41%

Tabla 20

Cumplimiento en los periodos por profesor subproblema 4

Profesor	Periodos ideal	Periodos programado	% cumplimiento
3	1	7	0%
4	3	2	0%
5	4 7	7 5,6	0%
7	2	5	0%
9	7	2	0%
10	4,5	6	0%
13	1,2	3,5	0%
14	7	4,5	0%
16	4	2,7	0%
17	1	4	0%

Continuación tabla 20

18	2,3	7	0%
19	2	7	0%
		Promedio	0%

De las tablas anteriores se puede concluir que el algoritmo respeta en buena medida la cantidad de cursos que pueden ser dictados por un profesor, el promedio de asignación de cursos es de 107,68%, se puede observar que al 23,29% de los profesores le son asignados más cursos de los que le corresponden mientras que, al 52,05% se le asigna exactamente la cantidad de cursos que se estipula pueden dictar y al 24,66% se le asignan menos cursos de los ideales.

Se deduce que en cuanto a programar cursos los días en los que los profesores tienen disponibilidad en promedio se respeta en un 43,66%. A un 17,81% de los profesores se les programan cursos exactamente los días que son estipulados en la matriz de disponibilidad, a un 49,32% de los profesores se les asignan cursos al menos en uno de los días estipulados mientras que al 32,88% no se le asignan los cursos en los días disponibles.

Por último, en cuanto a los periodos se analizó solo el de los días en los que a un profesor se le respeta al menos un día de su disponibilidad, es decir el 67,12% de los profesores y se observa que en promedio se respeta la disponibilidad de periodos en un 4,44%.

8.2 Observaciones

- Al asignar dos periodos en la matriz de disponibilidad para un profesor que tiene disponibilidad en horas impares se corre el riesgo de que a un profesor que solo puede dictar un curso le sean asignados dos.
- Al asignar los días restantes repitiendo la solución del algoritmo puede haber interferencia por lo cual hay que verificar siempre y de haber conflicto asignar manualmente respetando las restricciones.
- Cuando una asignatura tiene gran cantidad de cursos el algoritmo tiende a programar ese curso varias veces por lo cual hay que seleccionar el mejor de forma manual.
- En dos subproblemas no fue asignado ningún curso para un profesor, por lo cual se debió asignar manualmente.
- El tiempo total de ejecución es de 22248,23 s lo cual corresponde a 6,18 horas, siendo un tiempo de ejecución considerable.

9. Conclusiones

El problema de coloración de vértice es de gran utilidad a la hora de resolver problemas de asignación de recursos, en el presente trabajo se aplicó para resolver un problema de programación de horarios a través de un algoritmo memético. Los resultados obtenidos indican que el tamaño de la población influye en la capacidad de la ejecución del algoritmo propuesto.

Los objetivos propuestos se cumplieron en su totalidad, el algoritmo memético propuesto a pesar de que es un poco demorado en el momento de la ejecución al final de las corridas y a pesar de la complejidad del algoritmo arroja soluciones aceptables.

Se pudo observar que al usar una gran cantidad de datos el tiempo de ejecución es alto puesto que los datos de las matrices son binarios, esto hace que su tamaño sea muy grande y por ende ocasiona que el programa tenga que analizar una gran cantidad de información que al final se descartan porque son 0, ya que, lo que se necesita analizar son unos, lo anterior no representa un inconveniente cuando se usan menos datos ya que en estos casos se obtienen buenas soluciones en un tiempo aceptable.

10. Recomendaciones

El algoritmo cumple en cuanto a programar el número de cursos y a programar los días que tiene disponibilidad el profesor, así como también les asigna a los profesores los cursos que se le debe asignar, pero al momento de correr el programa la disponibilidad horaria de los profesores se omite un poco, si se respeta los días, pero las franjas no. Por eso es recomendable, realizar mejoras en el código, que busquen disminuir el tiempo de corrida y que respete las condiciones establecidas.

Al usar pocos datos el algoritmo corre rápido y la función de aptitud es mayor, se sugiere que al tener problemas grandes dividirlos en Sub problemas más pequeños para mejorar el tiempo de ejecución.

Considerar otras alternativas para realizar el cruce y mutación para el algoritmo genético, que permita obtener mejores soluciones.

En investigaciones futuras, se podría considerar el uso de matrices con números enteros, para que de esta manera se pueda reducir la cantidad de ceros en las matrices y así el algoritmo podría llegar a correr más rápido.

Referencias Bibliográficas

Adegbindin, M., Hertz, A., & Bellaïche, M. (2016). A new efficient rlf-like algorithm for the vertex coloring problem. *Yugoslav Journal of Operations Research*, *26*(4), 441–456.

Aguilar Imitola, K. J., & Perez Diaz, Y. T. (2013). *Un algoritmo memético para la minimización del makespan en el problema del Job Shop Scheduling*. 223.

Akbari Torkestani, J., & Meybodi, M. R. (2011). A cellular learning automata-based algorithm for solving the vertex coloring problem. *Expert Systems with Applications*, *38*(8), 9237–9247.

Caicedo, A., Wagner, G. y Méndez, R. (2010). *Introducción a la teoría de grafos*, Armenia, Colombia: Ediciones Elizcom.

Costa, D., & Hertz, A. (1997). Ants can colour graphs. *Journal of the Operational Research Society*, *48*(3), 295–305.

De Antonio Suárez, O. (2011). Una aproximación a la heurística y metaheurísticas. *Inge@UAN*, *1*(1), 44–51.

Fleurent, C., & Ferland, J. A. (1996). Genetic and hybrid algorithms for graph coloring. *Annals of Operations Research*, *63*, 437–461.

Furini, F., Gabrel, V., & Ternier, I. C. (2016). Lower Bounding Techniques for DSATUR-based Branch and Bound. *Electronic Notes in Discrete Mathematics*, *52*, 149–156.

Galán, S. F. (2017). Simple decentralized graph coloring. *Computational Optimization and Applications*, *66*(1), 163–185.

Galinier, P., & Hertz, A. (2006). A survey of local search methods for graph coloring. *Computers and Operations Research*, *33*(9), 2547–2562.

Guerrero, L. (2010). *Propuesta de un algoritmo Branch and Cut para resolver el problema de distribución de planta bidemensional con áreas desiguales.* (Tesis de maestría). Universidad Nacional Autómata de México.

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

Malaguti, E., & Toth, P. (2010). A survey on vertex coloring problems. *International Transactions in Operational Research*, *17*(1), 1–34.

Méndez-Díaz, I., & Zabala, P. (2006). A Branch-and-Cut algorithm for graph coloring. *Discrete Applied Mathematics*, *154*(5 SPEC. ISS.), 826–847.

Mitchell, M. (1998). *Elements of Generic Algorithms - An Introduction to Generic Algorithms*. 158.

Moalic, L., & Gondran, A. (2018). Variations on memetic algorithms for graph coloring problems. *Journal of Heuristics*, *24*(1), 1–24.

Moscato, P., & Cotta Porras, C. (2003). An Introduction to Memetic Algorithms. *Inteligencia Artificial*, *7*(19).

Myszkowski, P. B. (2008). Solving scheduling problems by evolutionary algorithms for graph coloring problem. *Studies in Computational Intelligence*, *128*(2008), 145–167.

Pahlavani, A., & Eshghi, K. (2011). A hybrid algorithm of simulated annealing and tabu search for graph colouring problem. *International Journal of Operational Research*, *11*(2), 136–159.

Pena, S. (2017). *El Problema de Coloración de Grafos.* (Tesis de maestría). Universidad de Santiago de Compostela, Galicia, España.

Rezapoor Mirsaleh, M., & Meybodi, M. R. (2016). A new memetic algorithm based on cellular learning automata for solving the vertex coloring problem. *Memetic Computing*, *8*(3), 211–222.

San Segundo, P. (2012). A new DSATUR-based algorithm for exact vertex coloring. *Computers and Operations Research*, *39*(7), 1724–1733.

Sianturi, E. T. (2019). Un modelo estocástico de ruteo para la recolección de escombros post-desastre en la ciudad de Bucaramanga. *8*(5), 55.

Tapias Isaza, C. J., Galeano Ossa, A. A., & Hincapie Isaza, R. A. (2011). Planeación de sistemas secundarios de distribución usando el algoritmo Branch and Bound. *Ingeniería y Ciencia - Ing.Cienc., 7*(13), 47–64.

Torkestani, J. A., & Meybodi, M. R. (2010). A new vertex coloring algorithm based on variable action-set learning automata. *Computing and Informatics, 29,* 447–466.