Universidad Autónoma Metropolitana Unidad Azcapotzalco

División de Ciencias Básicas e Ingeniería

Licenciatura en Ingeniería en Computación

Proyecto terminal

Un enfoque multi-objetivo basado en recocido simulado para resolver el problema de la coloración robusta

Rubén González Camacho 207202558

Trimestre 2014 Invierno

25 de marzo de 2014

Reporte Final

Asesor: Antonin Sébastien Ponsich

Profesor del Departamento de Sistemas, categoría Asociado

Yo, Dr. Antonin Sébastien Ponsich, declaro que aprobé el contenido del presente Reporte de Proyecto de Integración y doy mi autorización para su publicación en la Biblioteca Digital, así como en el Repositorio Institucional de UAM Azcapotzalco.

Firma del asesor

Yo, Rubén González Camacho, doy mi autorización a la Coordinación de Servicios de Información de la Universidad Autónoma Metropolitana, Unidad Azcapotzalco, para publicar el presente documento en la Biblioteca Digital, así como en el Repositorio Institucional de UAM Azcapotzalco.

Firma del alumno

Resumen.

El Problema de Coloración Robusta (PCR) es una variante del problema clásico de coloración de gráficas. En el PCR, el conjunto de colores a asignar es constante y el objetivo es encontrar una coloración que sea robusta en el sentido de que permanezca válida al introducir ciertos cambios a la estructura inicial del grafo (añadir aristas).

El PCR es un problema *NP*-duro, por lo que sólo instancias pequeñas del problema pueden resolverse mediante métodos exactos. Para instancias más grandes se han aplicado métodos heurísticos, de los cuales los métodos de búsqueda local han probado ser más eficientes.

Para este proyecto, se propone utilizar, como herramienta de resolución, la estrategia de multi-objetivización, que consiste en agregar objetivos a optimizar simultáneamente, en adición al inicial. Al buscar soluciones no-dominadas para los diferentes objetivos, esta técnica permite extraer el proceso de búsqueda de posibles óptimos locales. La elección o el diseño de los objetivos adicionales (helper objectives) se basó en una descomposición de los dos aspectos importantes en el PCR: minimizar la rigidez y tener una coloración válida (factible) mediante la minimización del número de violaciones de restricciones (conflictos).

Para resolver el problema de optimización multi-objetivo resultante, y debido al carácter combinatorio del problema tratado en este proyecto, se eligió aplicar una extensión de un algoritmo basado en técnicas de búsqueda local, el recocido simulado. Esta extensión, el AMOSA (Achived Multi-Objective Simulated Annealing) considera el tratamiento de varios objetivos gracias a un mecanismo de archivado de las soluciones no dominadas. Además, incorpora un nuevo concepto de cantidad de dominancia para determinar la probabilidad de aceptación de una nueva solución.

Después de ajustar los parámetros de funcionamiento, experimentos numéricos efectuados sobre un banco de problemas clásicamente utilizado en la literatura especializada han permitido concluir sobre el desempeño del algoritmo AMOSA: soluciones de buena calidad pueden ser obtenidas sobre el PCR en un tiempo de cómputo razonable con respecto a otros algoritmos del estado de arte, tales como la búsqueda dispersa o el Recocido Simulado mono-objetivo. Este estudio establece, finalmente, las bases necesarias para resolver el problema de PCR de acuerdo a la estrategia de multi-objetivización.

Contenido

1. Introducción.

El Problema de Coloración de Gráficas es un problema de optimización en el que se busca pintar los vértices de una gráfica con el número mínimo de colores posible, bajo la restricción de que dos vértices conectados por una arista no pueden compartir el mismo color. Este problema forma entonces parte de la clase de los problemas de partición, ya que se desea finalmente particionar un conjunto de elementos (los vértices) bajo ciertas restricciones, desde el siglo XIX, aplicaciones prácticas del mundo real han despertado interés por investigar y formalizar este tipo de problemas.

Existen muchas aplicaciones vinculadas al problema de coloración gráficas, tales como la asignación de horarios, la asignación de registros en compiladores, el diseño de sistemas de manufactura, etc. Si el PCG ha sido exhaustivamente estudiado en las últimas décadas, una variante reciente, el Problema de Coloración Robusta (PCR), ha recibido mucho menos atención. Propuesto en 2003 en [1], esta versión propone considerar posibles cambios en la estructura inicial de la gráfica. Desde este punto de vista, la mejor solución no es la que menos colores usa, sino la que mantiene su calidad a pesar de estos cambios de estructura. Por ejemplo, la mejor asignación de horarios para los cursos de una universidad no es la que minimiza el horario total necesario, sino la que se mantiene válida sin importar que los grupos de alumnos cambien.

La complejidad computacional asociada a este tipo de problema ha sugerido el uso de métodos de optimización heurísticos: no garantizan la optimalidad de la solución pero son insensibles a las propiedades matemáticas del modelo considerado y se pueden adaptar a cualquier problema, proporcionando una solución en cualquier instante de la búsqueda. Tanto para el PCG como para el PCR, la particular eficiencia de los métodos de búsqueda local ha sido destacada en varios trabajos.

En este proyecto se propone la aplicación de la estrategia de multi-objetivización. Esta estrategia, consiste en agregar, a un problema mono-objetivo, objetivos a optimizar simultáneamente, en adición al inicial. Esto puede permitir extraer el proceso de búsqueda de posibles óptimos locales. Para la aplicación de dicha estrategia, se decidió adaptar, para resolver el PCR, el algoritmo AMOSA, el cual considera el tratamiento de varios objetivos gracias a un mecanismo de archivado de las soluciones no dominadas. Además, incorpora un nuevo concepto de cantidad de dominancia para determinar la probabilidad de aceptación de una nueva solución.

El presente reporte se organiza de la forma siguiente. En la sección 3, se presentan el planteamiento del problema y los trabajos relacionados con el tema de estudio. La sección 4 introduce la metodología propuesta, desde la adaptación del algoritmo AMOSA al problema tratado hasta la definición de los objetivos propuestos en el estudio. La sección 5 expone los experimentos numéricos, realizados con los objetivos definidos en la sección 4. Las soluciones obtenidas se comparan mediante otras técnicas de

optimización. Finalmente, algunas conclusiones y perspectivas a este trabajo se proporcionan en la sección 6.

2. Objetivos.

2.1 Objetivo general

Adaptar la estrategia de multi-objetivización al problema de coloración robusta de gráficas e implementar el algoritmo AMOSA (*Archived Multi-Objective Simulated Annealing*) para la resolución del problema multi-objetivo resultante.

2.2 Objetivos específicos

- Implementar el algoritmo AMOSA y probar su desempeño sobre algunos problemas clásicos de optimización combinatoria multi-objetivo.
- Diseñar la estrategia de multi-objetivización, es decir la forma en que se van a elegir los objetivos adicionales, para el problema de coloración robusta.
- Calibrar el algoritmo sobre un conjunto reducido de instancias de prueba.
- Resolver las instancias de un banco de prueba propuesto en la literatura especializada (ver sección 5.1 Banco de instancias), para probar la eficiencia del algoritmo propuesto.
- Analizar los resultados y comparar el desempeño de la técnica propuesta con el de algoritmos del estado del arte.

3. Antecedentes.

El interés por los problemas de coloración de gráficas nace con los trabajos de F. Guthrie, quien estuvo trabajando sobre problemas de coloración de los mapas en Inglaterra, en el siglo XIX. La idea de Guthrie era colorear los diferentes condados del mapa, de forma tal que dos condados adyacentes no compartan el mismo color. Sus trabajo derivaron en la "conjetura de los cuatro colores", enunciando que cuatro colores son suficientes para cumplir tal objetivo. Cabe mencionar que este resultado fue refutado por Heawood en 1890, concluyendo que el número de colores necesarios es cinco.

A pesar de algunos trabajos de investigación en la primera mitad del siglo XX, no fue hasta los 70s que el problema de coloración fue considerado desde un punto de vista algorítmico. Desde entonces, una multitud de estudios, tratando los diferentes aspectos del problema, han sido desarrollados por investigadores de las áreas de Computación e Investigación de Operaciones.

3.1 Definiciones Generales:

Dada una gráfica $G = (V, E)$, donde V es un conjunto de vértices y E un conjunto de aristas $\{i, j\}$ tales que $i, j \epsilon V$, una c-coloración es una función $c: V \rightarrow \{1, 2, ..., c\}$.

Todos los vértices con el mismo color asignado forman una clase. Dos vértices x y y son adyacentes si existe en E una arista $\{x, y\}$. Si dos vértices adyacentes pertenecen a la misma clase, se dice que x y y son vértices conflictivos y que $\{x, y\}$ es una arista conflictiva. Una coloración es propia si no genera conflictos, i.e. si, para cualquier pareja de vértices adyacentes, cada uno tiene distinto color.

G es c-coloreable si existe una c-coloración propia para G. En la **Figura 1** se observan algunas coloraciones propias sobre una gráfica de ejemplo, se puede observar que para 3 ó 4 colores (denotados como R, A, V y N), es fácil encontrar distintas coloraciones propias para la misma gráfica. En cambio, no es posible encontrar ninguna coloración propia para dos colores. Al número mínimo de colores con el cual G es c-coloreable se le llama el número cromático de G, denotado como $\chi(G)$.

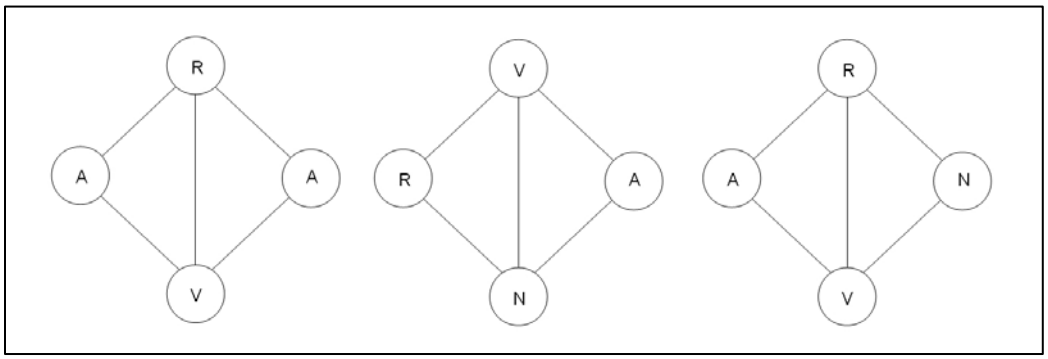

Figura 1. Coloraciones propias de 3 y 4 colores para una gráfica.

3.2 El Problema de Coloración de Gráficas:

Aunque estudiado de forma implícita desde el siglo XIX, el problema clásico de coloración de gráficas (PCG) fue introducido y formalizado en 1972 por R. Karp [2], con otros 20 problemas de optimización combinatoria considerados como muy difíciles de resolver de forma exacta. El problema consiste en determinar el número cromático de una gráfica dada, es decir la cantidad mínima de colores con la que se pueda colorear sin generar conflictos entre sus vértices. Lo anterior se formula como un problema de optimización, en el que se busca el valor mínimo c^* de tal forma que exista una c^*-coloración propia de la gráfica considerada.

En este contexto, un color es básicamente una etiqueta, por lo cual, dependiendo del problema, podría representar un recurso, un intervalo de tiempo (horario), etc. El PCG cubre por lo tanto un amplio rango de aplicaciones, en problemas relacionados con la asignación de frecuencias en comunicaciones inalámbricas, programación de horarios, distribución de recursos, diseño y operación de sistemas productivos [3], enrutamiento en redes computacionales [4], etc.

Se ha demostrado en [1] que el PCG es un problema *NP*-duro, por lo que sólo instancias pequeñas (en cuanto al número de vértices) pueden resolverse mediante métodos exactos. Para instancias más grandes, el tiempo computacional para encontrar una solución óptima se vuelve prohibitivo, por lo que los métodos heurísticos representan una opción más viable. Hertz y Zufferey [3], por ejemplo, proponen una solución al PCG por medio de un algoritmo de hormigas. El mejor algoritmo desarrollado para resolver el problema de coloración clásico, propuesto por Galinier y Hao [5], es un híbrido entre un algoritmo evolutivo y una búsqueda tabú.

3.3 El Problema de Coloración Robusta:

Como ya se mencionó, muchos problemas de asignación de horarios, recursos, etc., pueden modelarse como problemas de coloración, sin embargo, para algunos de ellos el PCG resulta ser un esquema

restrictivo. Efectivamente, en algunos casos, podría considerarse que la estructura de la gráfica (particularmente las aristas, asociadas a tareas compartiendo recursos) no es fija sino que puede ser sujeta a variaciones; mientras que, al contrario, el número de colores a utilizar (i.e., recursos disponibles) sea un parámetro constante.

Tal es el caso del Problema de Coloración Robusta (PCR), el cual es una variante del problema clásico de coloración de gráficas. Según este formalismo, el conjunto de colores a asignar es constante (lógicamente mayor o igual al número cromático de la gráfica) y el objetivo es encontrar una coloración que sea robusta, en el sentido de que permanezca válida al introducir ciertos cambios en la estructura inicial de la gráfica (por ejemplo, la adición de aristas nuevas).

En el PCR, se toman en cuenta las aristas complementarias del grafo (es decir las aristas que no existen en la estructura inicial del grafo) de forma tal que, si los vértices situados en sus extremos tienen el mismo color, se incrementa el valor de la función objetivo. Esta penalización representa típicamente la probabilidad de que dichas aristas aparezcan y puedan, por lo tanto, generar nuevos conflictos.

Este formalismo nuevo se puede ilustrar mediante el problema de asignación de horarios en una universidad, como es presentado en [1]. Con el objetivo de asignar horarios a los cursos impartidos, se puede definir una gráfica en la cual los vértices representen los cursos disponibles y se coloque una arista entre dos vértices cada que al menos un alumno esté inscrito en ambos cursos. De esta forma, los colores con los que se pretende colorear la gráfica representan los horarios en los que se pueden impartir los cursos. A primera vista, este problema podría resolverse encontrando una *c*-coloración propia de la gráfica generada, donde *c* es el número de horarios disponibles para agendar dichos cursos. Sin embargo, es posible que, bajo cierta probabilidad, un alumno ya inscrito en un curso, inscriba otro curso después de la asignación de horarios. Esto induciría la aparición de una nueva arista y la coloración inicial podría resultar inválida (i.e., los cursos asociados a la arista nueva ahora no podrían compartir el mismo horario). Conociendo la probabilidad de que esto ocurra, podría asignarse a cada arista complementaria una penalización proporcional a dicha probabilidad.

Formalmente el PCR puede finalmente definirse de la siguiente manera:

Dada una gráfica $G= (V, E)$ con $|V|= n$ y $|E| = m$ y sea c un número tal que $c>\chi(G)$. C es una c-coloración de G. Definimos la matriz de penalizaciones de G de la siguiente manera:

$$\{p_{ij} \geq 0, \{i,j\} \epsilon \bar{E}\}$$

Entonces, la *Rigidez* de la coloración C, denotada como $R(C)$, está definida como la suma de las penalizaciones de las aristas complementarias de G cuyos extremos inciden en dos vértices con el mismo color.

$$R(C) = \sum_{\{i,j\} \epsilon \bar{E}, C(i)=C(j)} p_{ij}$$

Esta Rigidez es la función objetivo a minimizar en el PCR, sujeta a la restricción de que C debe ser una

coloración propia. La formulación del problema de optimización es:

$$minR(C) = \sum_{\{i,j\}\epsilon\, \bar{E}, C(i)=C(j)} p_{ij}$$

$$s.a:$$

$$C(i) \neq C(j)\ \forall\ \{i,j\} \in E$$

Al igual que el PCG, ha sido probado que el PCR es un problema *NP*-duro [1,10], por lo que sólo instancias pequeñas del problema pueden resolverse mediante métodos exactos. Sin embargo, varias técnicas heurísticas han sido aplicadas para el tratamiento de instancias más grandes, demostrando particularmente la eficiencia de estos métodos de búsqueda. Una revisión del estado del arte en cuanto a estas estrategias se presenta en la sección siguiente.

3.4 Métodos de resolución para el PCR:

En la literatura especializada, se han encontrado diversos métodos para resolver el PCR, estos métodos van desde algoritmos exactos de programación entera [1] (útiles solo para instancias pequeñas), hasta algoritmos heurísticos, de búsqueda local o evolutivos.

A continuación se presentan brevemente algunos algoritmos heurísticos propuestos en Gutiérrez-Andrade et al. [6]. Todos estos algoritmos utilizan la siguiente función de aptitud:

$$\min f(S) = \sum_{\{i,j\}\epsilon\, \bar{E}, C(i)=C(j)} p_{ij} + \mu \sum_{i=1}^{k} E(V_i)$$

El primer término es la función objetivo de rigidez, presentada en la sección anterior. El segundo es un término de penalización de las soluciones infactibles: $|E(V_i)|$ es la cantidad de aristas conflictivas de G y μ un coeficiente numérico llamado factor de penalidad. El objetivo de esta formulación es penalizar la aptitud de las soluciones, en una forma proporcional al número de conflictos que generan. Al contrario, para una coloración propia ($|E(V_i)|$=0), la aptitud será evaluada únicamente en base a su Rigidez. En el estudio previamente citado, los autores usaron un valor de μ dependiendo del algoritmo de resolución utilizado (entre 3 –para un Recocido Simulado– a 100 –para una búsqueda tabú).

Las técnicas heurísticas propuestas en [6] son algoritmos de búsqueda local. Un algoritmo de búsqueda local parte generalmente de una solución inicial, generada ya sea aleatoriamente o por medio de otro algoritmo. A dicha solución se le practica una transformación para intentar mejorarla (con respecto de la función de aptitud del problema en cuestión), es decir, se busca una nueva solución en el entorno de la actual. Si la nueva solución es mejor, ésta sustituye a la actual. Se repite el proceso hasta que no sea posible obtener ninguna solución mejor. En ese momento se dice que se ha llegado a un óptimo local dentro del vecindario actual de solución.

Este modo de operación implica la definición del vecindario o entorno de una solución, necesariamente propio al problema tratado. La dificultad, al diseñar un vecindario específico, consiste en encontrar un balance entre eficacia y eficiencia: se desea poder generar cualquier solución del espacio de búsqueda (con tal de no perder el óptimo) pero, al contrario, no se quiere un vecindario de tamaño demasiado grande, para limitar el costo computacional asociado a su exploración.

Si una solución *S'* puede obtenerse a partir de una solución inicial *S*, introduciendo un cambio o una serie de cambios bien definidos en ella, se dice que *S'* es vecina de *S*. Al conjunto de soluciones vecinas de *S* se le llama el vecindario de *S* y se denota como *N(S)*.

En el marco de problemas de coloración de gráficas, una solución se representa como una partición del conjunto *V* en *c* clases $S = \{V_1, \ldots, V_c\}$. Entonces, una solución $S' = \{V_1', \ldots, V_c'\}$ se considera vecina de *S* si existen i, j y $x \in V_i$ tales que:

$$V'_i = V_i - \{x\};\ V'_j = V_j \cup \{x\}$$

$$V'_l = V_l \ \forall\ l \neq i, j$$

El proceso de formar *S'* a partir de *S*, es decir cambiar la clase (o color) del vértice *x*, se llama un movimiento. En [6], un movimiento se implementa seleccionando de manera aleatoria los índices *i* y *j*, así como el vértice *x*.

A continuación se explican brevemente los algoritmos utilizados en el estudio citado:

3.4.1 Búsqueda Tabú:

Es una técnica iterativa con la que se van buscando vecinos de *S* tratando de mejorar cada vez el valor de la función objetivo [7]. Este algoritmo registra una lista de los últimos movimientos realizados (los más recientes), llamada lista "tabú". Los movimientos en esta lista serán prohibidos o ilegales durante ciertas iteraciones del algoritmo y cada que se realiza un movimiento nuevo, la lista es actualizada. El objetivo es evitar movimientos cíclicos, de tipo $S \rightarrow S' \rightarrow S$. Cabe mencionar, sin embargo, que dichos movimientos "tabú" no son restricciones fuertes. De hecho, se pueden efectuar si y sólo si proporcionan una mejor solución que cualquier otro movimiento posible. El algoritmo termina después de un número fijo de iteraciones, o de un máximo de iteraciones sin obtener una mejora. En el **Algoritmo1** se presenta el pseudo-código del algoritmo de Búsqueda Tabú.

Algoritmo1. Algoritmo de Búsqueda Tabú:

Inicio
1 Generar una solución inicial
2 Para *i= 1* hasta *niter*:
3 Analizar soluciones vecinas: cambiar vértices entre 2 clases de color
4 Seleccionar la mejor solución

5 Seleccionar aleatoriamente un vértice y asignarle un color diferente de manera aleatoria
6 Fin para
Fin

**Cada que un vértice x cambia de la clase Vi a la clase Vj, una pareja {i, x} se añade a la lista tabú. Es decir, no se puede volver a colorear x con el color i en un número determinado de iteraciones.*

3.4.2 Recocido Simulado:

Es un algoritmo de búsqueda local cuyo modo de operación está basado en la analogía con el proceso de recocido físico [8]: después de calentarlo, un metal se enfría muy progresivamente, de forma que la estructura cristalina adoptada por los átomos alcanza un estado de energía mínimo. Similarmente, en el Recocido Simulado, se define un parámetro comúnmente asociado a la temperatura (*t*) del sistema. A diferencia de un algoritmo de búsqueda local simple, existe la posibilidad de aceptar una nueva solución *S'* a pesar de que ésta deteriore el valor de la función objetivo. La probabilidad de aceptar esta solución se calcula según el criterio de Metropolis [9]:

$$P = e^{\frac{f(S)-f(S\prime)}{t}}$$

La temperatura *t* tendrá su mayor valor cuando se inicie el ciclo principal del algoritmo e irá decreciendo conforme avancen las iteraciones. Así pues, cuando la temperatura *t* es alta, existe mayor probabilidad de aceptar una nueva solución a pesar de que empeore la solución actual. Cuando *t* es baja, es improbable aceptar soluciones que tengan un mayor valor en la función objetivo. Este procedimiento permite una exploración exhaustiva del espacio de búsqueda al inicio del algoritmo y, al contrario, una intensificación exclusivamente hacia mejores soluciones al final. El pseudo-código del algoritmo utilizado en [10] está presentado en el **Algoritmo2**.

Algoritmo2. Algoritmo de Recocido Simulado:

Inicio
1 Generar una coloración inicial de forma aleatoria
2 Evaluar el costo de la función objetivo
3 Hacer *t= √V(G) ; r= exp(d/t); B= 0.95; a= 1.05*
4 Mientras se acepte una nueva solución en un ciclo completo:
5 Para *i= 1* hasta *S*:
6 Generar una solución vecina *j*
7 Si *Δf= f(i) – f(j) > 0*, entonces *j* se vuelve la mejor solución *i*
8 Si no generar un número aleatorio *U(0,1)*
9 Si *exp(Δf/t)* es mayor a dicho número aleatorio, entonces *j* se vuelve la mejor solución *i*
10 Ajustar el parámetro: *r = a *r*

11 Fin Para
12 Ajustar el parámetro: *t = B * t*
13 Fin Mientras
Fin

3.4.3 GRASP:

Esta técnica, propuesta en [11], consta básicamente de 2 etapas. En la primera, llamada de construcción, se colorea la gráfica de manera glotona, es decir de acuerdo a una secuencia aleatoria generada de tal forma que cada color colocado genere una coloración factible (o propia). De no ser posible, se colorea el vértice en cuestión de cualquier color. La segunda etapa, dicha de mejora, consiste en un proceso de búsqueda local clásico, en el cual se varía el color de un vértice aleatoriamente y se acepta la nueva solución en base a la función objetivo. Este proceso termina cuando todos los colores han sido probados en todos los vértices y no se ha podido encontrar una mejor coloración (ver **Algoritmo 3**).

Algoritmo 3. Algoritmo GRASP:

Inicio
1 Para *m = 1* hasta *PoolSize*:
2 Generar una secuencia en la que los vértices se pintaran: *Sec(j)*
3 Para *j = 1* hasta *n*:
4 Generar una secuencia en la cual se intentará cada color: *SecCol(i)*
5 Para *i = 1* hasta *c*:
6 Si *Col(Sec(j)) = SeqCol(i)* es una coloración válida, pintar el vértice de ese color.
7 Fin Para *i*
8 Si *Col(j) = ϕ*, pintar el vértice de algún color aleatorio
9 Fin Para *j*
10 Mientras se obtenga un color que mejore la solución
11 Generar una secuencia para visitar los vértices *Sec(j)*
12 Para *j = 1* hasta *n*:
13 Generar una secuencia para intentar colores: *SecCol(i)*
14 Para *i = 1* hasta *c*:
15 Si *Col(Sec(j)) = SecCol(i)*, reduce el número de vértices mal coloreados o la rigidez, sin empeorar la coloración, entonces reemplazar la solución.
16 Fin Para *i*
17 Fin Para *j*
18 Fin Mientras
19 Fin Para *m*
Fin

3.4.4 Búsqueda Dispersa:

Este algoritmo trabaja sobre un banco de soluciones, inicializado con 100 soluciones generadas por el algoritmo GRASP. De dicho banco, se escoge un conjunto "de referencia" compuesto por las mejores soluciones, i.e. con rigidez baja. La mitad de éstas son soluciones factibles, mientras que la otra mitad son ligeramente infactibles (con 1 o 2 conflictos). Se lleva posteriormente a cabo un proceso iterativo en el cual se combinan y mejoran soluciones pasando por una etapa inicial de diversificación de soluciones que permite la actualización del conjunto de referencia. Este proceso se repite de forma iterativa, creando nuevos subconjuntos, combinando las soluciones obtenidas y actualizando el banco de soluciones en cada etapa, como se muestra en el **Algoritmo 4**.

Algoritmo4. Algoritmo de búsqueda dispersa:

Inicio: Un conjunto de 100 soluciones GRASP son generadas
1 Mientras nuevas soluciones sean agregadas a *RefSet*
2 Borrar del banco de soluciones las que sean redundantes
3 Escoger 5 soluciones válidas con la mejor rigidez
4 Escoger 5 soluciones con 1 o 2 conflictos y la mejor rigidez
5 Vaciar el banco de soluciones
6 Considerar todas las posibles combinaciones de conjuntos de 2 elementos
7 Considerar la unión de los 4 mejores elementos en conjuntos de 3 elementos
8 Para todos los subconjuntos con 2 elementos *a* y *b*:
9 Para *m = 1* hasta *particiones*:
10 Mezclar de manera aleatoria los elementos de la solución *Col(b)* con *Col(a)*
11 Usar el método de mejora en la nueva solución
12 Agregar la nueva solución al banco de soluciones
13 Fin Para
14 Fin "Para cada"
15 Para los 4 mejores elementos en *RefSet*:
16 Mezclar los 3 elementos en una proporción de *1/3* cada uno
17 Usar en las soluciones el método de mejora y agregarlas al banco de soluciones
18 Fin "Para los cuatro"
19 Fin Mientras
Fin

3.4.5 Un algoritmo evolutivo híbrido:

En Kong et al. [12], se introduce un algoritmo híbrido, combinando un algoritmo genético (AG) y un algoritmo de búsqueda local. El método comienza con una población inicial de coloraciones formadas aleatoriamente y consta de las siguientes etapas:

Selección: La población *m* se divide de manera aleatoria en *m/2* parejas. El operador de cruzamiento se aplica en cada pareja con una probabilidad de p_c.

Cruzamiento: Nuevas soluciones se obtienen utilizando el algoritmo de cruzamiento GPX [5] (Greedy

Partition Crossover), que genera un único hijo a partir de dos padres. El hijo hereda de la clase de mayor cardinalidad de cada padre sucesivamente, removiendo cada vez de ambos padres los vértices seleccionados. Al final, los vértices aún no asignados se reparten aleatoriamente entre las clases de la nueva solución.

Búsqueda Local: Se utiliza, en lugar del operador de mutación clásicamente usado en la versión canónica del AG, para mejorar a un individuo durante un número fijo de iteraciones. La originalidad del trabajo consiste en el diseño de un nuevo esquema de vecindades, que permite la construcción de mejores soluciones, con la desventaja de un costo computacional muy alto.

Actualización de la población: El individuo mejorado es insertado en la población, sustituyendo al individuo padre con peor valor en la función objetivo.

3.5 Un enfoque multi-objetivo.

Los orígenes de la optimización multi-objetivo suelen ser atribuidos a la obra de Adam Smith "La riqueza de las naciones", publicada en 1776, ya que es una parte inherente de la teoría de equilibrio económico. Sin embargo, fue el concepto de problema del vector máximo introducido por Harold W. Kuhn y Albert W. Tucker en 1951 el que permitió que la optimización multi-objetivo pudiera convertirse en una disciplina matemática propia.

3.5.1 Optimización mono-objetivo

En el mundo real, la mayoría de los problemas que se nos presentan, tienen múltiples objetivos (en ocasiones, estos objetivos, incluso están en conflicto entre sí) que se desean sean optimizados de manera simultánea. La estrategia generalmente usada para resolver este tipo de problemas, es tratándolos como un problema de optimización clásica (mono-objetivo). Para convertir dichos problemas a mono-objetivo, se elige un objetivo principal (el cuál será optimizado) y los demás se convierten en restricciones adicionales. Un problema de minimización mono-objetivo, se define de la siguiente manera:

$$Min\ f(X)$$

$$X = [X_1, \ldots, X_n] \in X$$

3.5.2 Multi-objetivización

En este proyecto se propone la aplicación de una estrategia poco utilizada (aunque no sea tan reciente), la de multi-objetivización. Esta metodología fue propuesta en Knowles et al. [13], donde los autores ilustran, por medio de pequeñas instancias del problema del agente viajero, que el "multi-objetivizar" un problema (es decir agregar objetivos a optimizar simultáneamente, en adición al inicial) puede permitir extraer el proceso de búsqueda de posibles óptimos locales. Efectivamente, en un proceso mono-objetivo, una solución candidata será rechazada si la solución actual es mejor que ella de acuerdo a un objetivo; pero el agregar objetivos adicionales aumenta la probabilidad de que las dos soluciones sean incomparables, por lo cual se podría aceptar la solución candidata: este procedimiento puede ser visto como una estrategia de escape de óptimos locales. Sin embargo, la elección o el diseño de objetivos adicionales es un punto crítico en la aplicación de esta estrategia.

Existen dos requisitos fundamentales para elegir objetivos adicionales al aplicar la estrategia de multi-objetivizaciòn. El primero, indica que el òptimo del problema original debe estar contenido en el conjunto de soluciones no-dominadas. El segundo requisito es que la consideraciòn d eobjetivos adicionales debe producir una eliminaciòn de los òptimos locales en el paisaje de aptitud. Aunado a los requisitos mencionados anteriormente, existen dos posibles enfoques para la correcta elecciòn de objetivos adicionales. El primer enfoque, por descomposición, se refiere a que el problema original se descompone en varios sub-problemas o sub-objetivos. En el segundo enfoque, por agregación, se diseñan objetivos completamente nuevos en el problema original.

3.5.3 Técnicas de optimizaciòn multi-objetivo

Un problema de optimización multi-objetivo se define de la siguiente manera:

$$Min\{f_1(X), f_2(X), \dots, f_k(X)\}$$

$$X = [X_1, \dots, X_n] \in X$$

Se considera sin perder generalidad que todos los objetivos son de minimizar, ya que un problema de maximización puede fácilmente transformarse en un problema de minimización.
Naturalmente, la consideración de varios objetivos implica tener herramientas adecuadas de optimización multi-objetivo. Esta área consiste en resolver problemas en los cuales se trata de optimizar simultáneamente dos o más funciones objetivo. A diferencia de cuando se trabaja con un único objetivo, existe generalmente no una sino un conjunto de soluciones óptimas, reformulando el concepto de optimalidad como "soluciones igualmente importantes", "incomparables entre sí" o "no-dominadas" [14].El concepto de dominancia se define de la siguiente manera: una solución X domina a X', si x es al menos tan bueno como X' para todos los objetivos:

$$\forall i \epsilon \{1, \dots, k\}, f_i(X) \leq f_i(X')$$

Y estrictamente mejor que *X'* para al menos un objetivo:

$$\exists j \epsilon \{1, \dots, k\}, f_j(X) < f_j(X')$$

Dos soluciones son no-dominadas entre sí, si una es mejor para algunos objetivos mientras que la otra es mejor para los demás.
Al conjunto de soluciones óptimas generadas se le llama "Conjunto de Pareto" y su representación en el espacio de los objetivos es denotada como "Frontera de Pareto".

Debido al interés nuevo generado por este tema, las dos últimas décadas han visto el desarrollo de muchos algoritmos que pretenden resolver eficaz y eficientemente problemas de optimización multi-objetivo. Particularmente, los Algoritmos Evolutivos han sido utilizados por su carácter poblacional que les permite aproximar, en una sola corrida, el conjunto de soluciones Pareto-óptimas. Algoritmos como el NSGA-II [15], SPEA2 [16] y PAES [17] son instancias representativas del estado del arte (cabe mencionar que existen algoritmos más recientes y más eficaces de acuerdo a ciertos criterios, pero con un costo computacional demasiado importante para ser considerados en el marco de este proyecto).

A parte de los Algoritmos Evolutivos, una línea de investigación de particular interés ha sido la extensión de otras técnicas metaheurísticas al tratamiento de problemas de optimización multi-objetivo. Entre estos trabajos destaca el algoritmo Archived Multi-Objetive Simulated Annealing (AMOSA) [18], particularmente adaptado para el tratamiento de problemas combinatorios y cuya descripción se proporciona en la sección siguiente.

3.5.4 AMOSA

El algoritmo AMOSA, propuesto por Bandyopadhyay et al. [18], es una extensión del algoritmo clásico de Recocido Simulado. Este método propuesto en Kirkpatrick [19] es una meta-heurística basada en la analogía que puede existir entre un proceso de optimización y un proceso termodinámico de alineación de cristales. De forma similar al proceso físico, a altas temperaturas se permite que los parámetros varíen con cierta facilidad pero conforme la temperatura baja, se reduce el margen de variación de los parámetros: disminuye la energía termodinámica del sistema, que desde el punto de vista algorítmico es la función objetivo.

A diferencia del recocido simulado multi-objetivo convencional, el recocido simulado multi-objetivo archivado (AMOSA) incorpora un nuevo concepto de cantidad de dominancia para determinar la probabilidad de aceptación de una nueva solución. Dicho concepto se ilustra gráficamente en la **Figura 2** para el caso de dos objetivos.

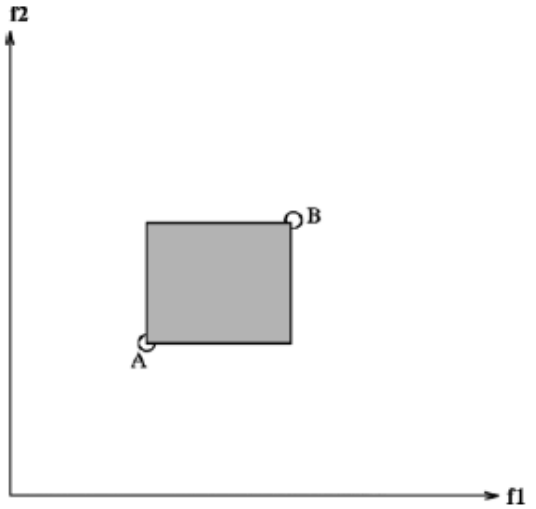

Figura 2. Cantidad total de dominancia entre dos soluciones A y B = área del rectángulo sombreado.

También incorpora un archivo externo en el cual se van almacenando las soluciones no-dominadas encontradas. El tamaño de este archivo es limitado ya que, finalmente, sólo se necesita un número limitado de soluciones eficientes, bien distribuidas a lo largo del frente de Pareto. El proceso principal de AMOSA comienza eligiendo un punto del archivo aleatoriamente (punto actual) como la solución inicial a temperatura máxima.

El punto actual es perturbado para generar una nueva solución (candidata), que eventualmente reemplazará la actual y será incluida en el archivo externo de soluciones no-dominadas. Este proceso de reemplazo constituye la originalidad del trabajo ya que está basado en el estado de dominancia de la solución candidata con respecto al punto actual y a las soluciones del archivo. Dependiendo de dicho estado de dominancia, [Bandyopadhyay et al.] identifican tres posibles casos:

- **Caso 1:** La solución nueva es dominada por la solución actual y por k (k > 0) soluciones del archivo. En este caso, la nueva solución reemplaza la actual con probabilidad P:

$$P = \frac{1}{1 + \exp(\Delta dom_{avg} \times temp)}$$

  Donde Δdom_{avg}es el promedio de cantidad de dominancia de la nueva solución por (k+1) puntos, a saber, la solución actual y k soluciones del archivo.
- **Caso 2:** Si las soluciones actual y nueva son no-dominadas entre sí, se pueden entonces presentar tres escenarios, dependiendo del estado de dominancia del nuevo punto y los puntos del archivo:
  - Si la solución nueva es dominada por k (k ≥ 1) soluciones del archivo, no se incluye en el archivo y sustituye la solución actual con la misma probabilidad P definida anteriormente.

- Si la solución nueva y las del archivo son no-dominadas entre sí, la solución nueva reemplaza la actual y se agrega al archivo. Si se supera la capacidad (finita) del archivo, se aplica un clásico proceso de agrupamiento para eliminar una solución del archivo, promoviendo una distribución uniforme de las soluciones a lo largo del frente de Pareto.

- Si la solución nueva domina k ($k \geq 1$) soluciones del archivo, se agrega al archivo y las k soluciones dominadas se eliminan del archivo.

- **Caso 3**: Si la solución nueva domina la actual, pueden presentarse tres escenarios, de acuerdo al estado de dominancia de la solución nueva y de las del archivo:

  - k (k ≥ 1) soluciones del archivo dominan a la candidata. Se obtiene el mínimo de la diferencia de cantidad de dominancia entre la solución nueva y las k que la dominan, denotada por Δdom_{min}. La solución del archivo que corresponde a la mínima diferencia es seleccionado como el punto actual con la probabilidad P' indicada a continuación, de lo contrario la solución nueva se vuelve la actual.

$$P = \frac{1}{1 + \exp(-\Delta dom_{min})}$$

  - Si la solución nueva y las del archivo son no-dominadas entre sí, exceptuando la solución actual (si ésta pertenece al archivo), sustituye a la solución actual y es almacenada en el archivo. Si el punto actual está en el archivo, se elimina.
  - Si la solución candidata también domina k ($k \geq 1$) soluciones del archivo, es seleccionada como solución actual y agregada al archivo. Los k puntos dominados del archivo son eliminados.

El resto del algoritmo, ilustrado en el diagrama de la **Figura 3**, es similar a la versión canónica de Recocido Simulado. El proceso antes descrito se repite un determinado número de iteraciones para cada temperatura, cuyo valor se reduce a una razón predefinida hasta alcanzar su mínimo. Se detiene entonces el proceso y el archivo externo contiene la aproximación final del frente de Pareto.

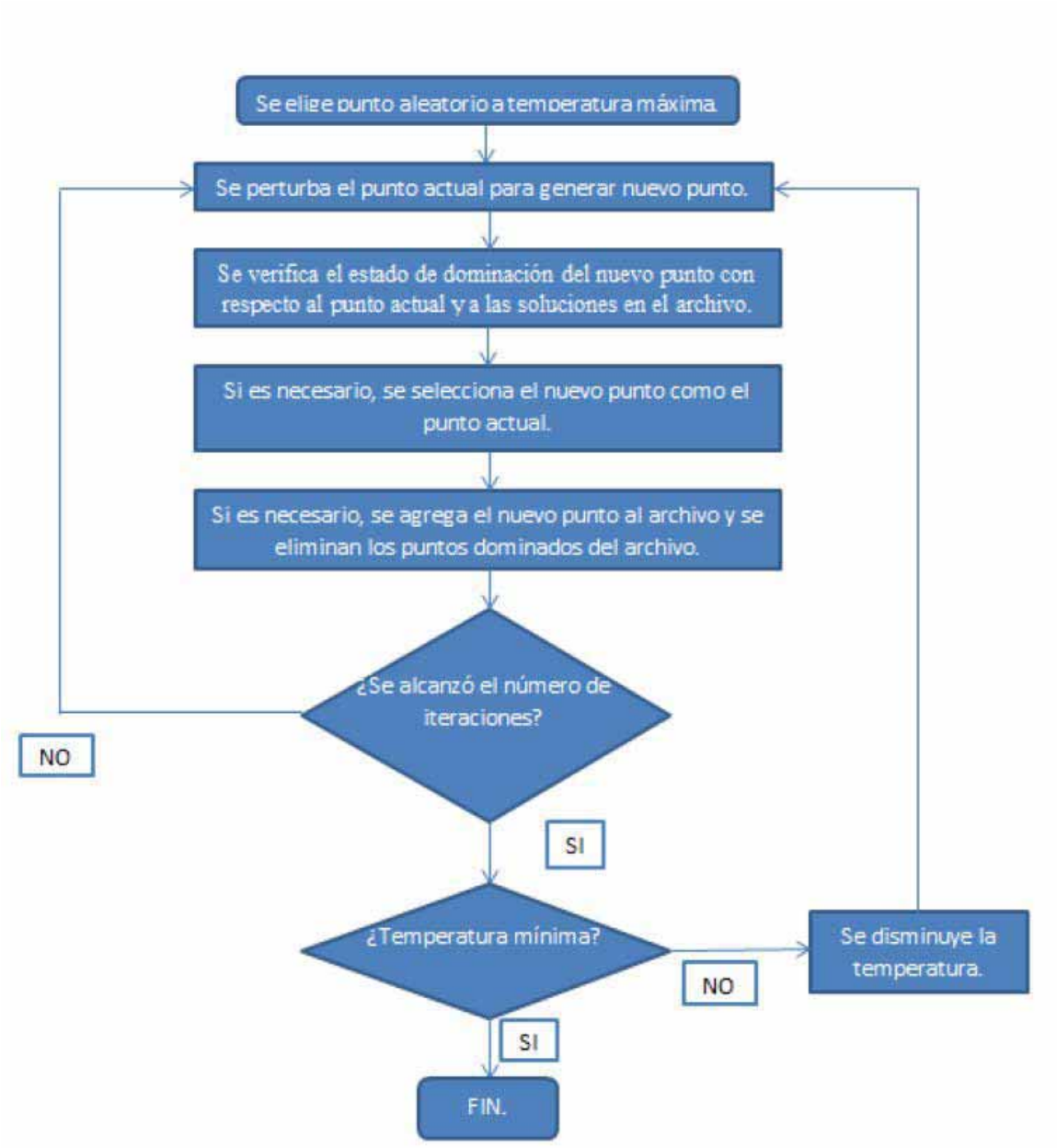

Figura 3. Proceso de AMOSA.

Cabe mencionar que este algoritmo ya ha tenido aplicaciones exitosas. Ha sido adaptado a sistemas de manufactura reconfigurables, para generar soluciones eficientes de acuerdo a tres criterios: tiempo de terminación de las operaciones, costo total y esfuerzo de reconfiguración (Bensmaine et al. [20]).

4. Metodología Propuesta

4.1 Descripción General del Proyecto:

En este proyecto se presenta una adaptación de la estrategia de multi-objetivización al PCR, definiendo un conjunto de pares de objetivos para implementar el algoritmo AMOSA y resolver el problema multi-objetivo resultante.

Constará de 3 módulos: el algoritmo AMOSA, un módulo de selección de objetivos adicionales y un banco de pruebas, como se muestra en la **Figura 4**.

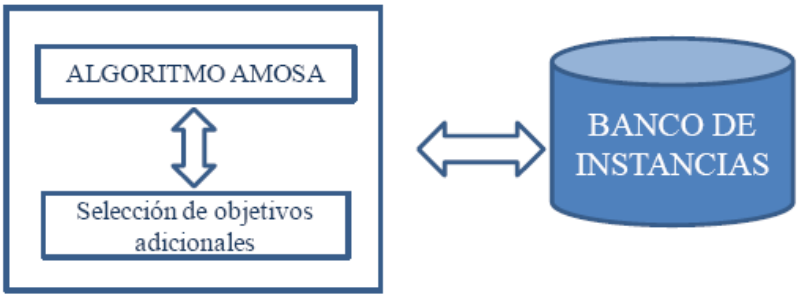

Figura 4. Descripción básica de los módulos del proyecto.

El banco de problemas está formado por un conjunto de archivos que contienen instancias de prueba, bien conocidas y ampliamente citadas en la literatura especializada. Su estudio puede resultar particularmente interesante por alguna propiedad como la estructura de la gráfica o la dificultad que presenta su resolución.

El módulo de selección de objetivos deberá determinar cuál es el objetivo o los objetivos adicionales a considerar en el proceso de optimización. Dado que estos objetivos son susceptibles de variar en el transcurso de una corrida, este módulo deberá "aprender" del proceso de búsqueda y su relación con el módulo de resolución (AMOSA).

Finalmente, el tercer módulo del proyecto corresponde al algoritmo AMOSA, descrito anteriormente.

4.2 Adaptación del algoritmo AMOSA para el PCR:

Para la realización de este proyecto se implementó, en un programa en lenguaje C++, el algoritmo AMOSA. Las entradas del programa son: Un entero N que representa el número de vértices de la gráfica, una gráfica representada por una matriz N x N que denota, en su matriz triangular inferior, la

adyacencia de los vértices y, en su matriz triangular superior, el conjunto de penalizaciones sobre las aristas complementarias y una constante k que representa el número fijo de colores a utilizar. Fue de particular interés en este proyecto observar cómo se comporta el algoritmo para entradas de diferentes tamaños. Asimismo, cabe recordar que la finalidad de utilizar métodos heurísticos es poder resolver instancias de tamaño mediano y grande (más de 20 nodos). Por lo que, tomando en cuenta lo anterior, cada instancia de prueba es una gráfica de N vértices, donde $20 \leq N \leq 100$.
El programa genera un conjunto de k-coloraciones (coloraciones válidas de k colores), procurando mejorar cada vez la solución. Las salidas del programa son la solución final (con la que se alcanzó el criterio de paro), la factibilidad y la rigidez de la solución (el valor de la función objetivo) y el tiempo en segundos que llevó resolver el problema.
Para la implementación del algoritmo AMOSA y su adaptación al PCR, se dividió el procedimiento en cuatro partes principales:

4.2.1 Inicialización del archivo

Como se mencionó anteriormente, el algoritmo AMOSA, incorpora el concepto de un archivo externo donde se almacenan las soluciones no dominadas visitadas a lo largo de la ejecución. Se ha decidido mantener el tamaño del archivo limitado ya que, finalmente, únicamente es necesario un cierto número de soluciones pareto-óptimas bien distribuidas. Para esto se utilizaron dos límites: Un límite estricto y un límite suave (denominados HL y SL respectivamente por sus siglas en inglés), los cuales deben ser inicializados *a priori*.

El archivo de inicializa con un número de $\gamma \times SL(\gamma > 1)$ soluciones. Cada una de dichas soluciones es refinada utilizando una técnica de *hill-climbing*, aceptando una nueva solución únicamente si esta domina a la anterior. Este proceso se realiza durante un número, previamente determinado (imax), de iteraciones. Para esto, se definieron dos estructuras (wActual y wNueva) que contienen, entre otros datos, un arreglo unidimensional de enteros que contiene la solución actual y la nueva, respectivamente y otro arreglo unidimensional que contiene los valores de los objetivos seleccionados. Se genera una posible solución aleatoria y, en cada iteración, se genera una nueva solución a partir de la actual, eligiendo aleatoriamente un vértice y pintándolo de otro color (elegido también aleatoriamente). Si la nueva solución domina a la actual, se actualiza la solución y el vector de objetivos de la solución actual.

Al alcanzar el criterio de paro, se obtienen las soluciones no dominadas y son almacenadas en el archivo, con un tamaño máximo igual a HL. En caso de que el número de soluciones no dominadas exceda HL, se aplica un algoritmo de clusterización (explicado en la siguiente sección) para reducir el tamaño hasta HL, esto significa que inicialmente, el archivo contiene un número máximo de HL soluciones.

Para obtener las soluciones no dominadas, se compara el estado de dominancia entre todas las soluciones y se conservan únicamente aquellas que no fueron dominadas por ninguna otra.

Para determinar el estado de dominancia entre dos soluciones, se utiliza la función *domina()*, cuyo

algoritmo es explicado en el anexo A1.2 del presente documento.

4.2.2 Clusterización del archivo

La clusterización del archivo es posiblemente utilizada al final del proceso de inicialización y durante el proceso principal de AMOSA. Esta técnica ayuda a reforzar explícitamente la diversidad de las soluciones no dominadas. En general, se permite al tamaño del archivo crecer hasta SL (> HL), después de esto, las soluciones son clusterizadas para agruparlas en un número de HL clusters. Al permitir que el archivo crezca hasta SL, no solo evita que la función de clusterización sea llamada excesivamente, sino que también permite la formación de clusters mejor distribuidos a lo largo del frente de pareto y por lo tanto, una mejor diversidad.

Para la función de clusterización, se utiliza el algoritmo de ligado simple [21]. En este, la distancia entre dos clusters corresponde a la longitud del vínculo más corto entre ellos. Una vez que un número de HL clusters es obtenido, el miembro de cada cluster con la menor distancia a los otros miembros de dicho cluster, es elegido como solución representativa. Para adaptar esta forma de clusterización al PCR, se empleó la distancia euclidiana en el espacio de los objetivos. Finalmente, los puntos representativos de cada uno de los HL clusters son almacenados en el archivo.

4.2.3 Algoritmo principal AMOSA

El algoritmo principal AMOSA se desarrolló de acuerdo a como fue mencionado en la sección 3.5.4 del presente documento.

Los parametros que fueron inicializados antes de éste procso son:

- HL: El tamaño máximo del archivo final. Éste debe ser inicializado como el número máximo de soluciones no dominadas requeridas por el usuario.
- SL: El tamaño máximo hasta el cual el archivo debe de ser llenado antes de aplicar la función de clusterización para reducir el tamaño hasta HL.
- Tmax: Temperatura máxima (inicial).
- Tmin: Temperatura mínima (final).
- Iter: Número de iteraciones en cada temperatura.
- Alfa: Razón de enfriamiento para el recocido simulado.

4.2.4 Cantidad de dominancia

Como se mencionó con anterioridad, AMOSA utiliza el concepto de cantidad de dominancia para calcular la probabilidad de aceptación de una nueva solución. Dadas dos soluciones *a* y *b*, la cantidad de dominancia se define como:

$$\prod_{i=1, f_i(a) \neq f_i(b)}^{M} (|f_i(a) - f_i(b)|/R_i)$$

Donde M = número de objetivos y R_i es el rango del i-ésimo objetivo. Nótese que en algunos casos, puede que no se conozca R_i *a priori*. En estos casos, se utilizan las soluciones en el archivo, junto con las soluciones nueva y actual para calular dicho rango.

4.3 Objetivos utilizados:

Para la realización de este proyecto se tomó en cuenta el enfoque de descomposición para llevar a cabo la multi-objetivización. La descomposición se efectuó en los conflictos y en la rigidez de las soluciones. Se tomó en cuenta también un valor llamado *Fitness*, el cual se calcula de la siguiente manera:

$$Fitness = Rigidez + k \times (Número\ de\ conflictos)^2$$

Donde *k* representa una constante determinada a partir de la necesidad de optimizar en mayor o menor magnitud el número de conflictos con respecto a la rigidez.

Se tomaron en cuenta cinco pares de objetivos (aunque el algoritmo está pensado para *n* número de obetivos) que se enlistan a continuación.

4.3.1 Conflictos y rigidez:

A pesar de que descomponer la función de aptitud de rigidez/conflictos no parece una estrategia prometedora, se decidió comenzar con este par de objetivos para ser aplicado como un estudio de sensibilidad.

En primer lugar, se buscó disminuir el número de conflictos, es decir, el número de vértices contiguos pintados del mismo color para lograr generar una solución factible. Como segundo objetivo, se eligió la

rigidez total de la solución (es decir, la suma de la rigidez individual de cada arista para los vértices pintados del mismo color).

Este par de objetivos fue utilizado como prueba preliminar para lograr ajustar de forma apropiada los diferentes parámetros de funcionamiento del algoritmo. Esta fase preliminar permitirá, en un segundo tiempo, efectuar corridas definitivas, esta vez sobre todas las instancias del banco, con el juego de parámetros más adecuado determinado en la etapa anterior.

4.3.2 *Fitness* y conflictos:

Una vez que se realizaron las pruebas preliminares con el objetivo de realizar un estudio de sensibilidad y ajustar los parámetros de funcionamiento del algoritmo, se decidió aplicar el valor denominado *fitness*. En este caso, fue necesario ajustar la constante *k* mencionada en la definición de dicha función en la sección 4.3 del presente documento.

Para las pruebas realizadas con este par de objetivos, no fue necesario hacer modificaciones considerables en la estructura del programa, únicamente se asignó el valor de *fitness* en los arreglos de objetivos de cada estructura de soluciones.

4.3.3 *Fitness* y número de conflictos del peor vértice

Como parte del enfoque de descomposición para obtener los nuevos objetivos a partir del problema mono-objetivo, se decidió descomponer los conflictos de acuerdo a los vértices. Para esto, se eligió como segundo objetivo (como primer objetivo se eligió el valor de *fitness*, definido anteriormente) el número de conflictos del peor vértice. El número de conflictos del peor vértice se determinó de la siguiente manera: Primero, se inicializó en cero la variable que contendrá dicho número (confPeor), también se inicializa en cero un vector unidimensional, de tamaño N (donde N = número de vértices) que contendrá el número de conflictos de cada vértice (confVert). Después, se recorre la matriz triangular inferior, y en caso de que dos vértices estén pintados del mismo color y sean contiguos, se incrementa (en el vector confVert) en uno el número de conflictos de los vértices en cuestión.

En caso de que el número de conflictos por vértice se haya incrementado, se verifica si el número de conflictos de cada uno de los vértices en cuestión es mayor al número de conflictos del peor vértice, de ser así, se toma como número de conflictos del peor vértice (confPeor).

La variable confPeor es actualizada para cada solución y es tomada como segundo objetivo a optimizar.

Por motivos de eficiencia, el proceso descrito anteriormente se lleva a cabo dentro de la función que calcula el número de vértices de cada solución (conf).

4.3.4 *Fitness* y rigidez del peor vértice

En este caso, la elección del objetivo adicional al del problema inicial (mono-objetivo) también se efectuó con base en la descomposición. La descomposición se llevó a cabo en la rigidez, de acuerdo a los vértices. Para esto, se eligió como segundo objetivo (como primer objetivo se eligió el valor de *fitness*, definido anteriormente) la rigidez del peor vértice. La rigidez del peor vértice se determinó de la siguiente manera: Primero, se inicializó en cero la variable que contendrá dicho número (rigPeor), también se inicializa en cero un vector unidimensional, de tamaño N (donde N = número de vértices) que contendrá la suma de las penalidades asociadas a las aristas complementarias que van de cada uno de los vértices hacia otro vértice pintado del mismo color (rigVert). Después, se recorre la matriz triangular superior, y en caso de que dos vértices estén pintados del mismo color y no sean contiguos, se incrementa (en el vector rigVert) el valor de la rigidez asociada a los vértices en cuestión.

En caso de que el valor de la rigidez asociada a los vértices se haya incrementado, se verifica si el valor de la rigidez asociada a los vértices en cuestión es mayor al valor de la rigidez asociada al peor vértice, de ser así, se toma como valor de la rigidez del peor vértice (rigPeor).

La variable rigPeor es actualizada para cada solución y es tomada como segundo objetivo a optimizar.

Por motivos de eficiencia, el proceso descrito anteriormente se lleva a cabo dentro de la función que calcula el número de vértices de cada solución (conf).

4.3.5 *Fitness* y rigidez del peor color

Para este último par de objetivos, se decidió también, aplicar el enfoque de descomposición al problema inicial (mono-objetivo) para la elección del objetivo adicional La descomposición se llevó a cabo, una vez más, en la rigidez, pero en esta ocasión de acuerdo a los colores. Para esto, se eligió como segundo objetivo (como primer objetivo se eligió el valor de *fitness*, definido anteriormente) la rigidez del peor color. La rigidez del peor color se determinó de la siguiente manera:

Primero, se inicializó en cero la variable que contendrá dicho número (rigPeor), también se inicializa en cero un vector unidimensional, de tamaño k (donde k = número de colores) que contendrá, para cada color, la suma de las penalidades asociadas a las aristas complementarias que van de cada uno de los

vértices hacia otro vértice pintado de dicho color (rigColor). Después, se recorre la matriz triangular superior, y en caso de que dos vértices estén pintados del color en cuestión y no sean contiguos, se incrementa (en el vector rigColor) el valor de la rigidez asociada a los vértices en cuestión.

En caso de que el valor de la rigidez asociada al color se haya incrementado, se verifica si el valor de la rigidez asociada al color en cuestión es mayor al valor de la rigidez asociada al peor color, de ser así, se toma como valor de la rigidez del peor color (rigPeor).

La variable rigPeor es actualizada para cada solución y es tomada como segundo objetivo a optimizar.

Por motivos de eficiencia, el proceso descrito anteriormente se lleva a cabo dentro de la función que calcula el número de vértices de cada solución (conf).

5. Experimentos Numéricos

Los experimentos numéricos realizados constan de dos principales etapas. La primera tiene como objetivo ajustar de forma apropiada los diferentes parámetros de funcionamiento del algoritmo, en base a su desempeño sobre dos ejemplos del banco de instancias. Esta fase preliminar permitirá, en un segundo tiempo, efectuar corridas definitivas, esta vez sobre todas las instancias del banco, con el juego de parámetros más adecuado determinado en la etapa anterior.

La primera fase de experimentos consiste en realizar un estudio de sensibilidad (véase la sección 4.3.1) para ajustar los parámetros mencionados en la sección 4.2.3, los cuales deben ser definidos *a priori*.

Por cuestiones prácticas, todas las pruebas de esta primera fase se hicieron sobre los tres ejemplos más sencillos considerados en el banco de instancias (involucrando 20, 30 y 40 vértices), ya que implican un menor tiempo de cómputo.

Finalmente, las conclusiones de la primera etapa se extrapolan a la totalidad del banco de instancias, utilizando los parámetros determinados para resolver todos los ejemplos. Estos experimentos se describen en una segunda fase y se comparan los resultados obtenidos con aquellos encontrados en [6]. Antes que nada, sin embargo, se presenta el banco de instancias utilizado en esta sección.

5.1 El banco de instancias

Las pruebas presentadas a continuación, se realizaron sobre un subconjunto de instancias de prueba obtenido del banco de problemas utilizado en [6]. Dicho banco de problemas consta de 22 instancias. Cada una, es una gráfica $G_{N,p}$ generada aleatoriamente , donde N, es el número de vértices de la gráfica, y p, la densidad de la misma, es decir, la probabilidad de que una arista exista entre cualesquiera 2 vértices.

Todas las instancias del banco de problemas tienen una densidad fija $p = 0.5$. El valor de N, varía entre 20 y 120 vértices. En este trabajo se estudian solo 6 de éstas instancias, presentadas en la **Tabla 2.** Donde la primera columna muestra el nombre de la instancia (*id*), la segunda, el número de vértices, la tercera el número de colores utilizados y la cuarta la rigidez de la mejor solución encontrada en [6].

Para las pruebas preliminares presentadas en el siguiente apartado, se utilizan las instancias *al20* y *al30*. Finalmente se evalúa el desempeño del algoritmo sobre el resto de las instancias en la sección de resultados.

Banco de problemas			
Id.	*N*	*c*	Mejor *Rg.*
al20	20	8	3.5934
al30	30	10	7.5749
al40	40	15	4.9947
al60	60	20	8.7568
al80	80	27	9.7132
al100	100	34	9.9658

Tabla2. Conjunto de casos a tratar.

5.2 Pruebas Preliminares

Como se mencionó con anterioridad, el primer conjunto de pruebas se utilizó para relizar un estudio de sensibilidad y así lograr ajustar los parámetros definidos en la sección 4.2.3 del presente documento.

Para estas pruebas se eligieron como objetivos el número de conflictos y la rigidez total de la solución. Las siguientes tablas comparativas, muestran los resultados obtenidos al realizar las pruebas para cada combinación de parámetros (definidos en la sección 4.2.3). Por cuestiones de espacio, se muestran únicamente los resultados más significativos. La mayor cantidad de pruebas se realizaron en la instancia *al20* con un nùmero fijo de 8 colores.

La **Tabla 3**, muestra un estudio estadístico con respecto al número de conflictos (mejor/promedio/desviación estándar) para cada juego de parámetros.

al20		0.9			0.95			0.99			alfa
		20	40	60	20	40	60	20	40	60	iter
Tmax	Tmin										
1000	0.001	2/7.46/3.66	2/7.78/3.74	2/7.59/3.70	2/7.26/3.56	2/7.30/3.92	2/7.82/3.88	2/7.52/3.50	2/7.32/3.68	2/7.41/3.66	
	0.0025	2/7.46/3.66	2/7.78/3.74	2/7.59/3.70	2/7.26/3.56	2/7.30/3.92	2/7.82/3.88	2/7.69/3.54	2/7.28/3.60	2/8.37/4.56	
	0.0005	2/7.46/3.66	2/7.78/3.74	2/7.59/3.70	2/7.26/3.56	2/7.30/3.92	2/7.73/3.91	2/7.40/3.52	2/7.26/3.63	2/7.81/3.88	
1500	0.001	2/7.78/3.77	2/7.67/3.73	2/7.62/3.64	2/7.24/3.77	2/7.42/3.70	2/7.61/3.73	2/7.40/3.69	2/7.50/3.58	2/7.41/3.52	
	0.0025	2/7.78/3.77	2/7.67/3.73	2/7.62/3.64	2/7.24/3.77	2/7.42/3.70	2/7.57/3.83	2/7.40/3.69	2/7.83/3.72	2/7.45/3.75	
	0.0005	2/7.78/3.77	2/7.67/3.73	2/7.58/3.64	2/7.24/3.77	2/7.42/3.70	2/7.61/3.73	2/7.40/3.69	2/8.17/3.85	2/7.80/3.66	
2000	0.001	2/7.75/3.74	2/7.77/3.78	2/7.35/3.73	2/7.70/3.72	2/7.65/3.83	2/7.46/3.62	2/7.24/3.64	2/7.24/3.56	2/7.47/3.52	
	0.0025	2/7.75/3.74	2/7.77/3.78	2/7.35/3.73	2/7.70/3.72	2/7.65/3.83	2/7.23/3.76	2/7.34/3.75	2/7.59/3.59	2/7.57/3.54	
	0.0005	2/7.71/3.74	2/7.77/3.78	2/7.35/3.73	2/7.73/3.72	2/7.65/3.83	2/7.61/3.69	2/7.24/3.64	2/7.53/3.51	2/7.32/3.43	

Tabla 3. Estudio estadístico para la instancia al20, usando 8 colores. Enfocado a conflictos.

En la **Tabla 4**, podemos observar el mismo análisis pero esta vez enfocado a rigidez:

al20		0.9			0.95			0.99			alfa
		20	40	60	20	40	60	20	40	60	iter
Tmax	Tmin										
1000	0.001	0.59/3.13/2.00	0.52/3.01/2.05	0.52/3.06/2.00	0.59/3.25/2.04	0.21/3.26/2.10	0.19/2.97/1.98	0.52/3.20/1.93	0.29/3.26/2.07	0.59/3.17/1.97	
	0.0025	0.59/3.13/2.00	0.52/3.01/2.05	0.52/3.06/2.00	0.59/3.25/2.04	0.21/3.26/2.10	0.19/2.97/1.98	0.52/3.12/1.93	0.52/3.39/2.48	0.59/2.98/1.98	
	0.0005	0.59/3.13/2.00	0.52/3.01/2.05	0.52/3.06/2.00	0.59/3.25/2.04	0.21/3.26/2.10	0.19/3.07/2.08	0.52/3.19/1.93	0.52/3.28/2.11	0.59/3.04/2.01	
1500	0.001	0.52/3.02/2.00	0.09/3.03/1.99	0.31/3.09/2.00	0.09/3.27/2.05	0.52/3.22/2.05	0.19/2.96/1.97	0.19/3.13/2.01	0.59/3.13/1.92	0.18/3.32/1.94	
	0.0025	0.52/3.02/2.00	0.09/3.03/1.99	0.31/3.09/2.00	0.09/3.27/2.05	0.52/3.22/2.05	0.19/3.09/2.04	0.19/3.13/2.01	0.59/3.07/1.92	0.18/3.20/1.97	
	0.0005	0.52/3.02/2.00	0.09/3.03/1.99	0.31/3.10/1.99	0.09/3.27/2.05	0.52/3.22/2.05	0.19/2.96/1.97	0.19/3.13/2.01	0.59/2.95/1.94	0.18/3.28/1.99	
2000	0.001	0.52/3.00/2.01	0.52/3.00/2.02	0.32/3.20/2.00	0.00/3.03/1.97	0.19/3.03/2.03	0.10/3.31/2.13	0.59/3.19/1.93	0.18/3.48/2.16	0.18/3.34/2.03	
	0.0025	0.52/3.00/2.01	0.52/3.00/2.02	0.32/3.20/2.00	0.00/3.03/1.97	0.19/3.03/2.03	0.10/3.46/2.33	0.59/3.17/1.93	0.18/3.34/2.05	0.18/3.27/1.92	
	0.0005	0.52/3.01/2.00	0.52/3.00/2.02	0.32/3.20/2.00	0.00/3.02/1.97	0.19/3.03/2.03	0.10/3.14/2.02	0.59/3.19/1.93	0.18/3.20/1.94	0.59/3.21/1.92	

Tabla 4. Estudio estadístico para la instancia al20, usando 8 colores. Enfocado a rigidez.

Al finalizar todas las pruebas con la instancia *al20*, se analizaron los resultados y, posteriormente, se realizaron pruebas con la instancia *al30* con un números fijo de 10 colores (**Tabla 5** y **Tabla 6**):

al30		0.9			0.95			0.99			alfa
		20	40	60	20	40	60	20	40	60	iter
Tmax	Tmin										
1000	0.001	6/14.31/4.93	6/14.59/4.99	6/14.71/5.06	6/14.68/4.87	6/14.70/4.61	6/14.57/4.78	6/15.20/5.11	6/14.84/5.03	6/13.97/5.21	
	0.0025	6/14.82/4.77	6/14.75/4.71	6/14.46/4.79	6/14.52/4.79	6/14.70/4.61	6/14.57/4.78	6/14.54/4.72	6/14.50/4.85	6/13.89/5.01	
	0.0005	6/14.37/4.85	6/14.59/4.99	6/14.46/4.79	6/14.27/4.94	6/14.70/4.61	6/14.54/4.65	6/14.00/4.81	6/14.86/4.98	6/14.59/4.91	
1500	0.001	6/14.82/4.85	6/14.10/4.83	6/14.65/4.80	6/14.41/4.91	6/14.54/4.82	6/14.47/4.90	6/14.62/4.73	6/15.00/4.78	6/14.61/4.73	
	0.0025	6/14.82/4.85	6/14.49/4.65	6/14.65/4.80	6/14.47/4.89	6/14.54/4.82	6/14.29/4.93	6/14.79/5.14	6/15.33/5.04	5/14.44/4.72	
	0.0005	6/14.82/4.85	6/14.53/4.77	6/14.72/4.82	6/14.41/4.91	6/14.54/4.82	6/14.47/4.90	6/14.89/4.89	6/14.25/4.79	5/15.08/5.06	
2000	0.001	6/14.78/4.92	6/14.34/4.76	6/14.83/5.10	6/14.08/4.87	6/14.56/4.89	6/14.22/5.07	6/13.96/4.95	6/14.14/5.16	6/14.09/4.91	
	0.0025	6/14.78/4.92	6/14.44/4.83	6/14.83/5.10	6/15.24/5.13	6/14.56/4.89	6/14.29/4.94	6/14.30/4.84	6/14.14/5.16	6/14.09/4.91	
	0.0005	6/14.78/4.92	6/14.44/4.83	6/14.78/5.00	6/14.11/4.80	6/14.56/4.89	6/14.35/4.94	6/13.96/4.95	6/14.14/5.16	6/13.58/5.10	

Tabla 5. Estudio estadístico para la instancia al30, usando 10 colores. Enfocado a conflictos.

al30		0.9			0.95			0.99			alfa
		20	40	60	20	40	60	20	40	60	iter
Tmax	Tmin										
1000	0.001	2.21/6.59/2.64	2.46/6.96/2.89	2.46/6.76/3.08	2.46/6.61/2.78	2.46/6.95/2.67	2.46/6.70/2.80	2.46/6.65/2.61	2.46/6.76/2.64	2.46/7.01/2.62	
	0.0025	2.21/6.67/2.77	2.46/6.96/2.89	2.46/6.65/3.13	2.46/6.61/2.78	2.15/6.71/2.73	2.46/6.77/2.74	2.46/6.65/2.61	2.46/6.73/2.80	2.46/6.94/2.81	
	0.0005	2.46/6.82/2.71	2.46/6.96/2.89	2.46/6.67/2.97	2.46/6.62/2.75	2.46/6.78/2.62	2.46/6.70/2.80	2.46/6.65/2.61	2.46/6.93/2.72	2.46/7.14/2.70	
1500	0.001	2.46/6.67/2.66	2.46/6.68/2.92	2.46/6.80/2.78	2.46/6.79/2.70	2.46/7.14/3.03	2.46/6.85/2.78	2.46/6.67/2.50	2.46/6.86/2.57	2.46/6.50/2.74	
	0.0025	2.46/6.67/2.66	2.44/6.43/2.70	2.46/6.66/2.62	2.46/6.42/2.68	2.46/6.84/2.61	2.46/6.50/2.74	2.46/7.17/3.12	2.46/7.09/2.66	2.46/6.90/2.91	
	0.0005	2.46/6.67/2.66	2.46/6.64/2.76	2.46/6.66/2.62	2.46/6.42/2.68	2.46/6.84/2.61	2.46/6.68/2.84	2.46/6.84/2.73	2.46/6.90/2.59	2.46/6.50/2.69	
2000	0.001	2.46/6.69/2.62	2.46/6.72/2.80	2.46/7.26/3.25	2.46/7.07/2.76	2.46/6.86/2.69	2.46/6.76/2.74	2.46/6.79/2.67	2.46/6.80/2.71	2.46/6.94/2.56	
	0.0025	2.46/6.69/2.62	2.46/6.72/2.80	2.46/6.97/2.84	2.46/7.07/2.76	2.46/6.74/2.74	2.46/6.89/2.78	2.46/7.01/2.62	2.46/6.73/2.55	2.46/7.07/2.77	
	0.0005	2.46/6.69/2.62	2.46/6.96/3.11	2.46/6.72/2.68	2.46/6.88/2.81	2.46/6.85/2.76	2.46/6.89/2.55	2.46/6.79/2.67	2.46/6.80/2.71	2.46/6.86/2.50	

Tabla 6. Estudio estadístico para la instancia al30, usando 10 colores. Enfocado a rigidez.

De acuerdo a lo observado en el estudio de sensibilidad, se eligió un conjunto de parámetros (**Tabla 7**) a ser utilizado para resolvber el PCR utilizando cada uno de los pares de objetivos mencionados anteriormente.

Parámetro	Valor
Tmax	1000
Tmin	0.001
alfa	0.9
iter	20

Tabla 7. Parámetros elegidos para resolver el PCR.

5.3 Resultados finales:

Una vez que se ajustaron los parámetros con ayuda del estudio de sensibilidad, se realizó una serie de pruebas con el resto de pares de objetivos presentados en la sección 4.3 del presente documento para cada una de las instancias mencionadas en la sección 5.1. Cada una de estas pruebas fue realizada con 10 semillas diferentes. A continuación (**Tabla 8**) se reporta el mejor resultado (la rigidez más baja con cero conflictos) obtenido con cada par de objetivos en cada una de las instancias, también se incluye el promedio y la desviación estándar obtenida al tomar en cuenta todas las soluciones factibles (es decir, las soluciones que no presentan conflictos). Para facilitar el análisis de la elección de objetivos adicionales al principal, los valores resaltados en negritas muestran, para cada instancia, la rigidez más baja entre los distintos pares de objetivos utilizados.

		Fitness vs conflictos			*Fitness* vs número de conflictos del peor vértice			*Fitness* vs rigidez del peor vértice			*Fitness* vs rigidez del peor color		
Instancia	Colores	Mejor	Prom.	Desv. Est.	Mejor	Prom.	Desv. Est.	Mejor	Prom.	Desv. Est.	Mejor	Prom.	Desv. Est.
al20	8	4.6934	4.8901	0.12	**4.6934**	4.7567	0.07	4.6934	4.7976	0.13	4.7109	4.8283	0.13
al30	10	8.374	11.2237	1.11	**7.5749**	8.5712	0.79	8.0879	9.0760	0.51	8.2617	8.8271	0.43
al40	15	7.1253	11.2329	1.76	6.7047	8.7194	0.74	**6.5406**	7.7094	0.62	7.0354	7.6927	0.54
al60	20	**12.9179**	14.9863	0	15.4452	21.3799	2.72	17.5907	18.1474	0.42	18.195	19.2472	0.92
al80	27	**18.6232**	17.4129	0	34.7862	35.7195	0.83	26.5639	26.7539	0.26	25.9583	25.9583	0
al100	34	**18.4108**	20.1248	0	29.4997	29.4997	0	33.0550	34.3118	1.12	35.4008	35.5806	0.31

Tabla 8. Resultados (mejor, promedio y desviación estándar) de cada par de objetivos y para cada instancia.

Se puede observar en la tabla anterior que para instancias pequeñas (20 y 30 vértices), los mejores resultados fueron otenidos utilizando como objetivos el valor de *fitness* y el número de conflictos del peor vértice, incluso se obtuvieron los mismos resultados que con las heurísticas presentadas en [6] (**Tabla 9**). Cabe mencionar que, para estas instancias, las pruebas realizadas con los demás pares de objetivos, arrojaron resultados considerablemente cercanos.

Para instancias más grandes, los mejores resultados fueron obtenidos utilizando como objetivos el *fitness* y el número de conflictos. En estos casos, los otros pares de objetivos arrojaron resultados considerablemente alejados.

Cabe mencionar que, como se explicó anteriormente, el programa realizado durante este proyecto, devuelve un conjunto de soluciones no dominadas, tomando en cuenta los objetivos seleccionados (ver Figura 5). Sin embargo, por practicidad, en este reporte únicamente se muestran los resultados de soluciones factibles.

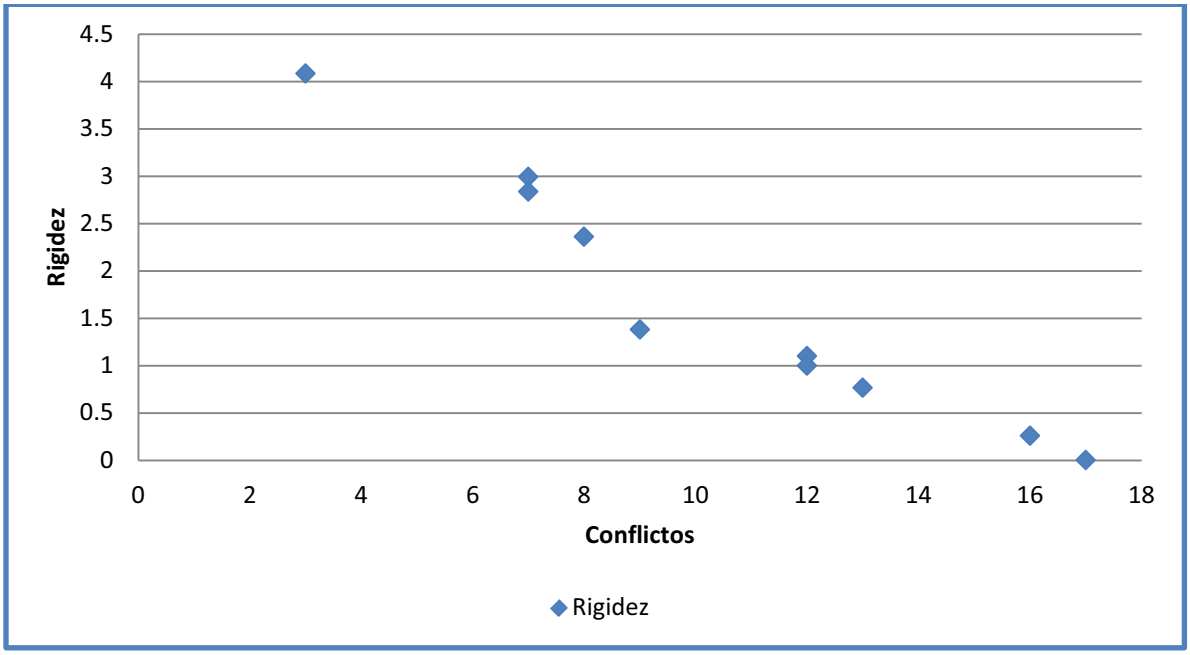

Figura 5. Gráfica de soluciones obtenidas para la instancia al20, utilizando los objetivos conflictos vs rigidez.

La **Tabla 9** muestra una comparación entre las mejores soluciones encontradas con el algoritmo propuesto en la sección 3.5.4 y las publicadas en [6]. En ella se muestran la solución encontrada con cada algoritmo y el tiempo de ejecución. Se puede observar que, aunque esta adaptación del algoritmo AMOSA genera buenas soluciones para instancias pequeñas, no es competitivo con los heurísticos

desarrollados en [6].

Estos resultados, invitan a pensar en alternativas para mejorar el desempeño del algoritmo, posiblemente eligiendo diferentes objetivos (ya sea mediante el enfoque de descomposición o el de agregación) o incrementando el número de objetivos.

		AMOSA		Búsqueda tabú		GRASP		Scatter Search	
Instancia	Colores	Mejor	Tiempo (s)	Mejor	Tiempo (s)	Mejor	Tiempo (s)	Mejor	Tiempo (s)
al20	8	4.6943	1.469	4.7710	---	4.6934	0.15	4.6934	1.3
al30	10	7.5749	11.531	8.0623	---	7.5749	0.44	7.5749	3.6
al40	15	6.5406	20.875	5.8173	14	6.3117	1.0	5.6708	7.2
al60	20	12.9179	1.25	9.8331	69	9.9687	3.3	8.8676	19
al80	27	18.6232	2.094	11.1946	218	11.7512	8.0	9.9835	52
al100	34	18.4108	3.14	12.1932	544	12.8675	15.8	10.0470	123

Tabla 9. Comparación de las mejores soluciones obtenidas con el algoritmo propuesto y otras heurísticas.

En la tabla anterior, se puede observar que para instancias pequeñas (en las que los mejores resultados fueron obtenidos utilizando los pares de objetivos *fitness* vs número de conflictos del peor vértice y *fitness* vs rigidez del peor vértice), el algoritmo propuesto en este proyecto, fue capaz de obtener resultados tan buenos como las heurísticas presentadas en [6]. Para instancias mayores como las de 60, 80 y 100 vértices (en las que los mejores resultados fueron obtenidos al elegir como objetivos el valor de *fitness* y el número de conflictos), los resultados no fueron tan satisfactorios a comparación de los presentados en [6], sin embargo, podemos observar que el tiempo de ejecución fue considerablemente mejor usando el algoritmo AMOSA, lo que podría indicar que al hacer un nuevo estudio de sensibilidad existe la posibilidad de obtener mejores resultados.

Cabe mencionar que las pruebas para la adaptación del algoritmo AMOSA, fueron realizadas con una computadora con procesador centrino duo a 1.83GHz, con 1 GB en RAM y sistema operativo Windows XP, lo que implica que los tiempos reportados en la tabla anterior, podrían variar.

6. Conclusiones

El objetivo primordial de este proyecto, consistiendo en adaptar la estrategia de multi-objetivización al PCR e implementar el algoritmo AMOSA para la resolución del problema multi-objetivo resultante, fue alcanzado mediante el cumplimiento de los objetivos específicos del mismo. Recapitulando las etapas sucesivas de la metodología:

1. Se implementó el algoritmo AMOSA y se probó su desempeño sobre algunos problemas clásicos de optimización combinatoria multi-objetivo.
2. Se diseñó la estrategia de multi-objetivización, es decir la forma en que se eligieron los objetivos adicionales, para el problema de coloración robusta. Para esto se seleccionó el enfoque de descomposición y se aplicó a los conflictos de acuerdo a los vértices, y a la rigidez tanto para los vértices como para los colores.
3. Se calibró el algoritmo sobre un conjunto reducido de instancias de prueba. Para esto, se eligieron como objetivos, el número de conflictos y la rigidez. Las instancias elegidas para dichas pruebas fueron las de 20, 30 y 40 vértices.
4. Se probó la eficiencia del algoritmo propuesto mediante la resolución de un banco de instancias propuesto en la literatura especializada.
5. Se hizo un análisis de los resultados, reportándolos en el presente documento por medio de tablas comparativas. Se compararon los resultados de cada conjunto de objetivos entre sí y finalmente, se realizó una comparación tomando en cuenta los mejores resultados de la técnica propuesta con el de algoritmos del estado del arte.

Si bien es cierto que el algoritmo presentado en este proyecto no obtuvo los resultados esperados con relación a los heurísticos presentados en [6], se han obtenido datos importantes del estudio realizado.

Uno de los resultados más importantes fue el hecho de que, al elegir objetivos tomando el enfoque de descomposición, que impliquen un manejo dinámico de los objetivos, como el número de conflictos del peor vértice, rigidez del peor vértice y rigidez del peor color, el tiempo de ejecución aumenta considerablemente debido a que para cada posible solución, se tiene que recalcular el objetivo a tomar en cuenta (en este caso se tuvo que recalcular el peor vértice tomando en cuenta los conflictos y la rigidez y el peor color tomando en cuenta la rigidez). Es posible que se pueda mejorar sustancialmente el desempeño de la técnica propuesta si se realizan algunas mejoras a la implementación. En este sentido, quedó también claro que procesos como la generación de números aleatorios y la actualización de las estructuras que componen a la solución ocupan una cantidad considerable de tiempo. La

optimización de dichos procesos, ayudaría en gran manera a subsanar los tiempos de cómputo excesivos.

Otro resultado importante se obtuvo al elegir como objetivos el valor de *Fitness* y la rigidez, ya que, para instancias mayores a 40 vértices, se obtuvieron los mejores resultados en comparación con los otros pares de objetivos, además de que el tiempo de ejecución era considerablemente mejor que el tiempo mostrado en [6]. Es posible que, al realizar un nuevo estudio de sensibilidad, tomando en cuenta únicamente estos dos objetivos, se pueda mejorar considerablemente el desempeño del algoritmo AMOSA, con un tiempo de ejecución sustancialmente bueno.

Todavía queda mucho por hacer en la búsqueda de un algoritmo eficiente para el PCR. Entre las mejoras que se pueden realizar como una posible extensión a este proyecto están las siguientes:

1. Realizar un estudio de sensibilidad para cada par de objetivos propuestos en este proyecto, ya que es posible que el ajuste de parámetros que se realizó, no sea el más efectivo para cada uno de los pares de objetivos tomados en cuenta. Posiblemente, esto mejoraría los resultados obtenidos en las instancias más grandes.

2. Realizar un estudio de nuevos pares de objetivos. Por ejemplo, se podría tomar en cuenta combinaciones de los objetivos propuestos en este proyecto o aplicar nuevos objetivos no tratados en este proyecto como el número de conflictos del peor color. También se podría realizar un estudio utilizando objetivos elegidos a partir del enfoque de agregación como la varianza del número de vértices en cada color.

3. Realizar un estudio en el que se optimicen no únicamente pares de objetivos, sino una mayor cantidad de objetivos, lo que podría ayudar a salir más fácilmente de óptimos locales. Para esto se podrían elegir, por ejemplo, tripletas de objetivos elegidas a partir de los objetivos seleccionados en este proyecto o elegidas aplicando nuevos objetivos como los propuestos en el punto anterior.

7. Bibliografía

[1] J. Yañéz y J. Ramírez, “The robust coloring problem”, *European Journal of Operational Research*, vol. 148, pp. 546-558, 2003.

[2] R.M. Karp. "Reducibility Among Combinatorial Problems". In R. E. Miller and J. W. Thatcher (editors). Complexity of Computer Computations. New York: Plenum, 1972, pp. 85–103.

[3] A. Hertz y N. Zufferey, "La coloration des sommets d’un graphe par colonies de fourmis" en *Fourmis Artificielles – Tome 1, Des bases algorithmiques aux réalisations avancées*, Francia: Hermès-Lavoisier, 2009, pp. 261- 279.

[4] H.A. Peelle, "Graph coloring in J: an introduction", en *Proc. APL*, pp.77-82, 2001.

[5] P. Galinier y J. Hao, “Hybrid Evolutionary Algorithms for Graph Coloring”, *Journal of Combinatorial Optimization*, vol. 3, pp. 379-397, 1999.

[6] M. A. Gutiérrez Andrade, P. Lara Velázquez, R. López Bracho y J. Ramírez Rodríguez, “Heuristics for the robust coloring problem”, *Revista Matemática: Teoría y Aplicaciones*, vol. 18, pp. 137-147, 2011.

[7] F. Glover y M. Laguna, *Tabu Search*, Kluwer academic Publishers, ISBN 0-7923-9965-X, 1998.

[8] S. Kirkpatrick, C.D. Gelatt y M.P. Vecchi, “Optimization by Simulated Annealing”, Science, vol. 220, No. 4598, pp. 671-680, 1983.

[9] M. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, E. Teller, “Equation of State Calculations by Fast Computing Machines", *The Journal of Chemical Physics*, vol. 21, No. 6, pp. 1087-1095, 1953.

[10] A.L. Laureano-Cruces, J. Ramírez-Rodríguez, D.E. Hernández-González, I.I. Méndez-Gurrola, “An ant colony algorithm for the robust graph coloring problem”, in: ICGST AIML Conference, Dubaï (EAU), Abril 2011.

[11] M.G.C. Resende, “Computing approximate solutions of the maximum covering problem using GRASP”, *Journal of Heuristics*, vol. 4, pp. 161-171, 1998.

[12] Y. Kong, F. Wang, A. Lim y S. Guo, “A New Hybrid Genetic Algorithm for the Robust Graph Coloring Problem”, *Lecture Notes in Computer Science* , vol. 2903, pp. 125-136, 2003.

[13] J. Knowles et al., “Reducing local optima in single-objective problems by multi-objectivization,” en Proceedings of the First International Conference on Evolutionary Multi-Criterion Optimization. Springer-Verlag, Berlin, Alemania, 2001.

[14] Rouge, Lausanne, “Pareto V.,” en Cours d’économie, Suiza, 1896.

[15] Deb K. et al., “A fast and elitist multiobjective genetic algorithm: NSGA–II,” en IEEE Transactions on Evolutionary Computation, vol. 6(2), pp. 182-197, 2002.

[16] E. Zitzler et al., “SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization,” en K.C. Giannakoglou and others, editors, Evolutionary Methods for Design, Optimisation and Control with Application to Industrial Problems (EUROGEN 2001), International Center for Numerical Methods in Engineering (CIMNE), pp. 95-100, 2002.

[17] Knowles J. y Corne D.W., “Approximating the Nondominated Front Using the Pareto Archived Evolution Strategy,” en Evolutionary Computation, vol. 8(2), pp. 149-172, 2000.

[18] Bandyopadhyay et al., “A simulated annealing-based multiobjective optimization algorithm: AMOSA,” en IEEE Transactions on evolutionary computation, 2007.

[19] S. Kirkpatrick et al., “Optimization by Simulated Annealing,” en Science 220, 1983, vol. 4598, pp. 671–680.

[20] Bensmaine A. et al., “Process Plan Generation in Reconfigurable Manufacturing Systems using AMOSA and TOPSIS,” en Proceedings of the 14th IFAC Symposium on Information Control Problems in Manufacturing, Romania, 2012.

[21] A. K. Jain and R. C. Dubes, Algorithms for Clustering Data. Englewood Cliffs, NJ: Prentice-Hall, 1988.

Anexo 1. Implementación

Con el objetivo de llevar a cabo las pruebas descritas en la sección 4, se desarrolló una implementación en del algoritmo presentado en este trabajo. El código fuente del programa se encuentra dividido en 13 funciones. Cada una de las funciones es necesaria para el funcionamiento global del programa.

A continuación, se presenta una descripción general de la función del programa, aquí se incluye también una explicación de la función pincipal. posteriormente se explica a detalle cada una de las funciones secundarias.

A1.1 Descripción General:

Representación de los Datos:

Como se mencionó en la sección 3, para el PCR, una gráfica se puede representar mediante una matriz de adyacencia-penalización. Durante la ejecución del programa, se mantiene en memoria una representación de dicha gráfica mediante un arreglo bidimensional de $N \times N$ entradas de punto flotante (m).

Se declara también una constante *kFit* que representa el valor que tendrá k para obtener el valor de *Fitness* (ver sección 4.3)

Para los casos en que se eligieron objetivos que implican un manejo dinámico, se declararon variables globales que contienen el número de conflictos o valor de la rigidez para el peor de los vértices, o el valor de la rigidez para el peor de los colores, según sea el caso (ver secciones 4.3.3, 4.3.4 y 4.3.5).

Una solución, en este contexto, es una estructura con los siguientes miembros:

obj: Es un arreglo unidimensional de números de punto flotante, de tamaño k (donde k = número de objetivos) donde se almacenan los valores de los objetivos de la| solución en cuestión.

sol: Es un arreglo unidimensional de números enteros, de tamaño N+1 (donde N = número de vértices), donde se almacena la solución. Índica de qué color está pintado cada color. Cada color está representado por un número x donde $0 \leq x \leq col$ (donde col = número de colores disponibles). Entonces, por ejemplo, wActual.sol[3] = 5; índica que el vértice 3 de la solución actual estará pintado con el color 5. En el último índice de este vector, se utiliza en el proceso de clusterización, para almacenar el número de cluster al que pertenece la solución en cuestión.

Con el objetivo de obtener la cantidad de dominancia, es necesario obtener el rango de los objetivos

(véase sección 4.2.4), para eso se utiliza una estructura llamada *r* que consta de un solo miembro llamado *o* que es un vector unidimensional de tamaño *k* (donde *k* = número de objetivos), de números de punto flotante, el cual contendrá los valores máximos, mínimos y rangos de los objetivos respectivamente.

Durante el proceso de clusterización, también se utiliza una estructura llamada *distancias* que consta de un único miembro que es un vector bidimensional de tamaño *numSols* (donde *numSols* = número de soluciones al momento de la inicialización del vector) llamado *dis,* el cual contendrá los valores de las distancias que existen ya sea entre las soluciones (*s*) o entre los clusters (*c*).

La función principal *main()* del código desarrollado consta de tres partes principales, descritas a continuación:

Inicialización:

En la fase de inicialización, se inicializan las siguientes tareas:

- Se inicializa el reloj para medir el tiempo de ejecución de la prueba.
- Se asigna el número de objetivos a utilizar en la prueba,
- Se determina el tamaño de los vectores de: objetivos de la solución actual y nueva, máximos, mínimos y rangos de objetivos; a partir del número de objetivos asignado en el punto anterior.
- Se leen los valores de *N* (número de vértices) y *col* (número de colores) del archivo de entrada.
- Se determina el tamaño de los vectores de: instancia obtenida del archivo de entrada, número de conflictos o valor de la rigidez para el peor de los vértices, o el valor de la rigidez para el peor de los colores (en los casos expuestos en las secciones 4.3.3, 4.3.4 y 4.3.5).
- Se llena el vector *m* a partir del archivo de entrada de la instancia a resolver.

AMOSA:

Durante la fase del algoritmo AMOSA, se inicializan los parámetros mencionados en la sección 4.2.3 del presente reporte. Además, se definen algunas variables que serán utilizadas más adelante como el número de iteraciones máximo (*imax*) para inicializar el archivo (ver sección 4.2.1), el valor de la semilla que será utilizada para los valores que se obtendrán aleatoriamente, el contador de iteraciones, el número de soluciones que se generarán para posteriormente obtener las soluciones no dominadas que serán almacenadas en el archivo; también se determinan el tamaño del vector de objetivos (se inicializa con un valor de -1) y de la solución mencionados en la representación de datos y se inicializa en cero

un vector (*noDom*) unidimensional de tamaño *numSols* que indicará si una solución es dominada por alguna otra (valor > 0) o es no dominada (valor = 0).

- Inicialización del archivo:

  Para la inicialización del archivo se siguen los siguientes pasos:

  - ✓ Se genera una solución inicial aleatoria y se copia en la solución actual (*wActual.sol*).
  - ✓ Se asignan los objetivos de la solución actual en el vector de objetivos (wActual.obj). Nótese que esta parte varía dependiendo del par de objetivos elegidos.
  - ✓ Para cada iteración (*imax*):
    - ➢ se genera una nueva solución a partir de la actual y se almacena en el vector correspondiente (*wNueva.sol*) junto con sus objetivos (*wNueva.obj*) utilizando la función *genera()* descrita en la sección A1.2.
    - ➢ Con ayuda de la función denominada *domina()*, la cual se describe en la sección A1.2, se obtiene el estado de dominancia entre ambas soluciones (*wActual* y *wNueva*).
    - ➢ Si la nueva solución domina a la actual, se actualiza la solución.
  - ✓ Se agrega la solución actual (*wActual*) al vector de estructuras *soluciones* junto con sus objetivos en la parte correspondiente (*soluciones.obj*).

  Este proceso se repite *numSols* veces.

- Obtención de soluciones no dominadas:

  Una vez que se tiene lleno el vector de estructuras *soluciones* (de tamaño *numSols*), se obtienen las soluciones no dominadas. Para esto, se hace una comparación de todos los objetivos de cada una de las soluciones. En esta parte se utiliza el vector llamado *noDom* mencionado anteriormente.

  Si al terminar una comparación de los objetivos de una solución *x* con los objetivos de otra solución *y*, se concluye que *x* domina a *y*, el vector *noDom* se incrementa en uno en la posición *y*, indicando que ésta no es una solución no dominada.

Posteriormente, para llevar un conteo de las soluciones no dominadas en el archivo, se cuentan las posiciones del vector *noDom* que tienen un valor de 0 y se pasan todas las soluciones no dominadas al principio del archivo (el vector de estructuras *soluciones*).

Finalmente, si el número de soluciones no dominadas es mayor a *HL*, se manda llamar la función *clusteriza()*, descrita en la siguiente sección.

- Proceso principal de AMOSA:

  Antes de inicializar el proceso principal de AMOSA, se manda llamar la función *obtieneRangos()*, descrita en la siguiente sección; con el propósito de conocer el rango actual de los objetivos.

  En esta fase, se sigue al pie de la letra, el algoritmo de AMOSA, descrito en la sección 3.5.4. Este proceso se apoya de las siguientes funciones: *obtieneRangos(), copia(), genera(), domina(), obtieneDelta(), aleat(), quitaSol(), clusteriza();* todas descritas en la sección A1.2.

Impresión de resultados

Finalmente, se calcula el tiempo de ejecución, y se almacena en la variable de punto flotante de doble precisión *time*. Se imprime dicha información en la consola y se genera (o sobrescribe, de ser necesario) el archivo de salida "solución.txt". Dicho archivo contendrá los valores de los parámetros definidos en la sección 4.2.3, la(s) solución(es) y el tiempo de ejecución.

A1.2 Funciones secundarias:

Como se mencionó en la sección anterior, las siguientes funciones son parte fundamental de programa. Cada una de ellas tiene un funcionamiento específico dentro del programa y fueron implementadas de tal forma que fueran re-utilizables.

int domina(int k, float *p, float *q);

Esta función sirve para determinar el estado de dominancia entre dos soluciones (p y q). Recibe como parámetros el número de objetivos a tratar y apuntadores a los vectores de objetivos de ambas soluciones. Devuelve un número entero que será: -1, si la solución p domina a la solución q; 0, si ambas soluciones son no dominadas entre sí o 1, si la solución q domina a la solución p. No utiliza ninguna otra función al ser ejecutada.

Para determinar el estado de dominancia, primero se determina *i1* (el primer índice de objetivo para el cual los objetivos son diferentes); si $i1 = k$ (donde k = número de objetivos), significa que todos los

objetivos son iguales para ambas soluciones, lo que implica que las soluciones son no dominadas entre sí. En caso contrario, si el objetivo *i1* de *p* es menor que el correspondiente en *q*, tomamos la hipótesis de que *p* domina a *q*, pero hace falta confirmarlo con los objetivos restantes, para lo cual se inicializa un segundo índice: *j = i1 + 1*, si *j=k*, significa que *i1* es el último objetivo, con lo cual se confirma la dominancia de *p*; en caso de que exista algún objetivo en *q* que sea menor a su correspondiente en *p,* no se confirma la dominancia de *p*, lo que implica que las soluciones son no dominadas entre sí. El proceso para confirmar la dominancia de *q* sobre *p* es análogo.

float rig(int *wp, int N);

Esta función calcula la rigidez total de una solución. Recibe como parámetros un apuntador al vector de la solución en cuestión y un número entero *N* que indica el número de vértices del problema tratado. No utiliza ninguna otra función durante su ejecución y devuelve un número de punto flotante que representa el valor de la rigidez de la solución.

int conf(int *wp, int N);

Esta función calcula el número de conflictos en una solución dada. Recibe como parámetros un apuntador al vector de la solución en cuestión y un número entero *N* que indica el número de vértices del problema tratado. No utiliza ninguna otra función durante su ejecución y devuelve un número entero que representa el número de conflictos de la solución.

void genera(int N, int col);

Esta función sirve para, a partir de una solución dada, generar una solución vecina (por medio de una perturbación) y asignarle los nuevos valores de los objetivos. Recibe como parámetros dos enteros: *N* y *col* que indican el número de vértices y colores del problema en cuestión, respectivamente. Para su funcionamiento, se elige un vértice aleatoriamente y se pinta de un color diferente al actual, elegido también aleatoriamente.. Finalmente, se calculan los objetivos de la nueva solución (lmacenada en el vector *wNueva.sol*) por medio de las funciones correspondientes (dependiendo de cuáles sean los objetivos analizados en el problema) y se almacenan en el vector de objetivos de dicha solución (*wNueva.obj*). No devuelve ningún valor.

float aleat();

Esta función devuelve un número de punto flotante, generado aleatoriamente entre cero y uno. No recibe parámetros ni manda llamar funciones definidas dentro del proyecto.

void obtieneDist(int n);

Esta función obtiene la distancia euclidiana entre las soluciones actuales del archivo basándose en el valor de los objetivos. Únicamente recibe como parámetro el número de soluciones que se encuentran actualmente en el archivo (el vector de estructuras *soluciones*). No utiliza ninguna función para su ejecución y no devuelve ningún valor ya que actualiza el valor de las distancias en el vector bidimensional llamado *dis* (mencionado en la parte de representación de datos de la sección anterior).

int clusteriza(int HL, int numSols, int N);

Esta es la función de clusterización, se basa en el procedimiento mencionado en la sección 4.2.2 del presente reporte. Recibe como parámetros: un número entero *HL* que indica el valor del límite estricto (el número de soluciones que deberá haber al finalizar el proceso de clusterización), un número entero *numSols* que indica el número de soluciones que hay en el archivo al momento de ser llamada la función y un número entero *N* que indica el número de vértices que tiene el problema en cuestión.Devuelve un número entero que representa el npúmero de soluciones al terminar el proceso de clusterización y utililiza la función *obtieneDist()* (explicada anteriormente) para obtener la distancia euclidiana que existe entre las soluciones del archivo.

void obtieneRangos(int solsArchivo, int k);

Como se mencionó anteriormente, estra función sirve para obtener los rangos de los objetivos actuales, en caso de que estos no puedan ser conocidos *a priori*. Es usada en la determinación de la cantidad de dominancia. Recibe como parámetros dos números enteros: *solsArchivo* que indica el número de soluciones en e archivo al momento de la llamada y *k* que indica el número de objetivos tratados en el problema en cuestión. No devuelve ningún valor ya que los rangos se actualizan en lo vectores *max*, *min* y *rango*; mencionados en la representación de los datos de la sección anterior.

float obtieneDelta(int k, float *a, float *b);

Esta función devuelve un número de punto flotante que representa la cantidad de dominancia que existe entre dos funciones *a* y *b*. Recibe como parámetros un número entero *k* que indica el número de

objetivos usado en el problema en cuestión, y dos apuntadores a vectores unidimensionales de números de punto flotante que contienen los valores de los objetivos de las soluciones *a* y *b*. Utiliza el vector que contiene la estructura *r* descrita en la representación de datos de la sección anterior para calcular la cantidad de dominancia.

int quitaSol(int x, int solsArchivo, int N, int k);

Esta función sirve para desechar una solución que se encuentra en el archivo. Se utiliza en algunos casos del proceso principal de AMOSA, descrito en la sección 3.5.4. Recibe como parámetros cuatro números enteros: *x* que indica el índice en el que se encuentra la solución a desechar dentro del archivo, *solsArchivo* que indica el número de soluciones en el archivo en el momento en que se llama la función y, finalmente, *N* y *k* que indican el número de vértices y objetivos del problema en cuestión, respectivamente. La actualización del archivo se realiza directamente en el vector de estructuras *soluciones*. Devuelve un número entero que indica el número de soluciones que hay en el archivo una vez que se eliminó la solución deseada. Utiliza para su ejecución, la función *copia()*, descrita a continuación.

void copia(struct individuo a, struct individuo b,int N, int k);

Esta función sirve para copiar una solución (y sus objetivos) *a* en otra solución *b* (actualizando también los objetivos). No devuelve ningún valor ya que la actualización se hace directamente en la estructura que recibe como segundo parámetro. Recibe también como parámetros una estructura *a* (fuente) y dos números enteros que indican el número de vértices y de objetivos utilizados en el problema en cuestión, respectivamente.

void imprime(struct individuo a, int N);

Esta función se utiliza en la fase de impresión de resultados, descrita en la sección anterior. Imprime, al final del archivo de salida *solucion.txt*, tanto la solución de la estructura que recibe como parámetro, como el valor de sus objetivos. También recibe como parámetro un número entero *N* que indica el número de vértices que contiene el problema en cuestión. No devuelve ningún valor.