Universidad Autónoma Metropolitana Unidad Azcapotzalco

División de Ciencias Básicas e Ingeniería

Licenciatura en Ingeniería en Computación

Heurísticas híbridas con búsqueda en vecindades variables para el problema de coloración robusta

Trimestre 13P

Reporte de Proyecto Terminal II

Junio de 2013

Elaborado por:

Georgina Cruz Gutiérrez 207201586

Cecilia Tapia Benítez 207200116

Asesores

Dr. Pedro Lara Velázquez Departamento de Sistemas

Dr. Roman Anselmo Mora Gutiérrez Departamento de Sistemas

Tabla de contenido

Resumen

1. Introducción

El problema de coloración robusta (PCR) es un problema combinatorio del tipo NP-Difícil [Yañez 2013], el cual permite obtener coloraciones donde no sólo es importante encontrar soluciones válidas, sino también que sean estables, es decir, que al hacer modificaciones al problema, la solución obtenida continúe siendo válida con una probabilidad cercana a uno [Ramírez 2003].

La técnica de búsqueda adaptativa aleatorizada y glotona (GRASP por sus siglas en inglés) fue desarrollada a finales de los años ochenta [Resende 2002] [Brownlee 2012], con el objetivo inicial de resolver problemas de cubrimiento de conjuntos. Es un procedimiento que consta de una fase de construcción que obtiene una solución factible aplicando un procedimiento voraz y una fase de búsqueda local que mejora la solución voraz.

El algoritmo de recocido simulado (RS) está basado en una analogía con el recocido de sólidos, un concepto utilizado en metalurgia, y la problemática de resolver problemas de optimización combinatoria de gran escala [Gutiérrez 1991]. El recocido metalúrgico es un proceso de recalentamiento de un sólido a una temperatura en la que sus granos deformados recristalizan para producir nuevos granos. Seguida a la fase de calentamiento, viene un proceso de enfriamiento en donde la temperatura se baja poco a poco. La simulación del proceso de recocido puede usarse para describir un proceso de generalización de soluciones de un problema de optimización combinatoria en donde se vayan obteniendo, conforme el proceso avanza, mejores soluciones al mismo.

La búsqueda de vecindades variables (VNS por sus siglas en inglés) es una técnica utilizada para mejorar una solución inicial para un problema de optimización que da mejores resultados que la búsqueda local simple. Es un proceso que en cada iteración busca una solución vecina de la solución actual que mejore el valor de la función objetivo, utilizando distintos vecindarios.

El propósito de este trabajo es diseñar dos algoritmos híbridos, un algoritmo GRASP/VNS y otro RS/VNS para resolver el problema de coloración robusta y evaluar su comportamiento respecto a instancias de prueba utilizadas en varios artículos de investigación, por ejemplo [Yañez 2003] [Gutiérrez 2012].

2. Objetivos

Objetivo general

Diseñar e implementar dos algoritmos híbridos metaheurísticos, un algoritmo GRASP/VNS y un algoritmo RS/VNS para resolver el Problema de Coloración Robusta (PCR).

Objetivos específicos

- Desarrollar un algoritmo GRASP para el PCR.
- Desarrollar un algoritmo de recocido simulado para el PCR.
- Implementar del algoritmo GRASP para el PCR.
- Implementar del algoritmo recocido simulado para el PCR.

3. Problema de coloración robusta

El PCR considera dos gráficas, una llamada original o G, la cual es idéntica a la utilizada en el problema de coloración mínima [Diestel 2000], y otra gráfica llamada complementaria o G', formada por todas las aristas no incluidas en E, que representan las incompatibilidades que podrían ocurrir posteriormente denotadas como $\bar{E}$. $\bar{E}$ es conocido como el conjunto de aristas complementarias. Las aristas complementarias se definen como:

$$\{i,j\} \in \bar{E} \leftrightarrow \{i,j\} \notin E$$

Entonces la gráfica complementaria está formada por:

$$G' = (V, \bar{E})$$

A las aristas de la gráfica complementaria les asociamos una penalización, p_{ij}, que indica la penalización que hay en la arista complementaria que conecta a los vértices i y j. Esta penalización puede indicar, por ejemplo, la probabilidad de que una arista forme parte de la gráfica original en un futuro cercano. Una coloración en particular que se realiza con k colores se le denota como C^k. Al color asignado a un vértice i se le denota como C (i). Una propiedad del PCR es la función de rigidez R (C^k), la cual se define como la suma de las penalizaciones de las aristas complementarias cuyos vértices extremos tienen el mismo color:

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

Otra propiedad del PCR es que no necesariamente se debe trabajar con la cantidad mínima de colores, ya que el objetivo es optimizar la rigidez y no minimizar el número de colores, en este problema la cantidad máxima de colores que se utilizan, se fija de antemano.

4. GRASP, Recocido Simulado y Búsqueda en Vecindades Variables

La técnica GRASP fue introducida por Feo y Resende en 1995 como una nueva técnica meta-heurística para uso general que obtiene buenas soluciones en tiempos cortos. GRASP es un procedimiento multi-arranque donde se genera un conjunto de soluciones (por lo general un centenar de ellas. Cada solución tiene una fase de construcción y una de mejora. En la fase de construcción se utiliza un procedimiento heurístico, por lo general se utiliza un algoritmo glotón aleatorizado para obtener una buena solución inicial y esta solución se mejora en la segunda fase a través de un algoritmo de búsqueda local. La mejor solución factible en el conjunto de soluciones es el que los rendimientos de algoritmos. Una breve descripción de cada fase del algoritmo de GRASP se muestra a continuación:

- *Algoritmo glotón aleatorizado.* Genera soluciones añadiendo una entrada de una en una, cada elemento está seleccionado para la inclusión de la solución entre un conjunto de las mejores soluciones elegibles para su inclusión. El proceso finaliza cuando se genera una solución completa.
- *Búsqueda local.* Una vecindad es el conjunto de todas las soluciones que se pueden alcanzar a partir de una solución por medio de "un movimiento" que puede ser, por ejemplo, el intercambio entre los dos elementos en la solución inicial. La característica fundamental para el éxito en una búsqueda local es su estructura y tamaño. En cuanto al tamaño de la vecindad más grande mayor será la calidad de las soluciones. Una vecindad de gran tamaño produce una heurística más eficaz, aunque será lento para explorar.

El algoritmo de recocido simulado se basa en una analogía entre el recocido simulado de sólidos y la solución de problemas de optimización combinatoria de gran escala. El recocido en metalurgia denota el tratamiento a alta temperatura hecho en metal o vidrio, donde se calienta el material, por lo general hasta que brilla intensamente, y se deja enfriar gradualmente. A temperaturas muy altas los granos se deforman y producen nuevos granos recristalizados. Este proceso cambia las propiedades físicas del material, como su dureza y ductilidad. La temperatura de recocido depende del tipo de material y su uso en el futuro. Después de la fase de calentamiento, el proceso de enfriamiento es donde la temperatura desciende gradualmente. Sin embargo, si el proceso de enfriamiento es demasiado rápido en cada etapa ha alcanzado el equilibrio térmico, el sólido enfriado estará en un estado cuya estructura será amorfa en vez de cristalina. La estructura amorfa se caracteriza por las dislocaciones, que significa, una imperfecta cristalización del sólido. En la analogía, con nuestro problema, las soluciones quedan atrapadas en un óptimo local con un valor relativamente alto de rigidez. De lo contrario, si dejamos enfriar el material muy lentamente, las partículas se pueden reorganizar en estados de energía más bajos hasta que

se obtiene un sólido con átomos dispuestos en una estructura cristalina. En nuestro problema, podemos llegar a un valor mucho más bajo de rigidez.

El equilibrio térmico se caracteriza por la distribución de Boltzmann [Toda 1983]. De acuerdo con esta distribución, la probabilidad de que el sólido está en un estado i con energía *Ei* a la temperatura *T*, viene dada por

$$\mathrm{P_T\{X} = i\} = \frac{1}{Z(T)} \exp\ (\frac{-\mathrm{E_i}}{K_B T})$$

donde *X* es una variable aleatoria que indica el estado actual del sólido. *Z(T)* es una constante de normalización denominado la función de partición, que es una función de densidad de probabilidad, que se define como:

$$Z(T) = \sum_j exp\ (\frac{-\mathrm{E_j}}{K_B T})$$

donde la suma se extiende a todos los estados posibles y k_B es una constante física conocida como la constante de Boltzmann. El factor $exp(-E_j/K_B T)$ que se llama el factor de Boltzmann. Se puede ver en esta ecuación que cuando el valor de T disminuye, la distribución de Boltzmann se concentra en los estados de energía más bajos, mientras que si la temperatura es cercana a cero, sólo los estados de energía mínimas tienen una probabilidad distinta de cero de ocurrencia.

Como se ha dicho antes de que el proceso de recocido consiste en dos pasos básicos:

• El aumento de la temperatura del baño termal a un valor máximo.
• Disminuir la temperatura del baño térmico cuidadosamente hasta que las partículas se reorganizan en un estado de mínima energía del estado fundamental del sólido.

Cuando la temperatura del sólido se eleva a la temperatura de recocido, todas las partículas se reorganizan al azar. En el estado óptimo, las partículas están dispuestas en un sistema de red altamente estructurada y la energía es mínima, y no hay ningún otro tipo de dislocaciones. El estado óptimo sólo puede alcanzarse si la temperatura máxima es lo suficientemente alta y el proceso de enfriamiento es suficientemente lento.

El proceso físico de recocido puede ser modelado utilizando métodos de simulación [Metropolis 1953], este algoritmo se basa en las técnicas de Monte Carlo y genera una secuencia de estados sólidos como sigue: dado un estado *i* del sólido con energía *Ei*, un estado posterior *j* es generado por la aplicación de un mecanismo de perturbación que transforma el estado actual al estado siguiente por una pequeña distorsión. La energía en el siguiente estado es *Ej*. Si la diferencia de energía, *Ej - Ei*, es menor que o igual a cero, el

estado *j* es aceptado como el estado actual. Si la diferencia de energía es mayor que cero, el estado *j* se acepta con una probabilidad dada por

$$\exp\left(\frac{E_i - \mathrm{E_j}}{K_B T}\right)$$

donde T indica la temperatura del baño térmico y k_B es la constante de Boltzmann. La regla de decisión se ha descrito anteriormente se conoce como el criterio de Metropolis y el algoritmo se llama el algoritmo de Metropolis.

Si la temperatura se reduce gradualmente, el sólido puede alcanzar el equilibrio térmico en cada cambio de la temperatura. En el algoritmo de Metropolis esto se logra mediante la generación de un gran número de transiciones para un valor dado de la temperatura.

La técnica de Búsqueda en Vecindades Variables (VNS por sus siglas en inglés) [Hansen 1997] implica la exploración iterativa de diferentes estructuras de vecindades con el objetivo de salir de óptimos locales. La estrategia es dirigida por tres principios:

1. Una estructura de vecindad mínima local puede no ser una estructura mínima local para un vecindario diferente.
2. Un mínimo global es el más pequeño de todos los posibles valores mínimos en las estructuras locales.
3. En la mayoría de los casos, los mínimos locales están relativamente cerca del mínimo global.

5. Algoritmos Propuestos

Este proyecto consta del diseño y la implementación de dos algoritmos híbridos para resolver el problema de coloración robusta. De los dos algoritmos propuestos, el primero de ellos consiste de un algoritmo GRASP en donde cada iteración incluye una fase de construcción de una solución y otra de mejora usando VNS de la solución generada en la primera fase. Este algoritmo en su implementación simple dió buenos resultados en tiempos de ejecución del orden de minutos [Gutiérrez 2012, Lara 2005]. El segundo algoritmo, llamado recocido simulado, es un algoritmo de búsqueda metaheurística cuyo objetivo general es encontrar una buena aproximación al valor óptimo de una función en un espacio de búsqueda grande, este algoritmo ha dado los mejores resultados en calidad de soluciones aunque sus tiempos de ejecución son mucho mayores [Lara 2005].

Algoritmo Greedy aleatorizado. Como se puede ver en es pseudocódigo a continuación, una solución inicial se construye seleccionando un elemento para su inclusión en la solución de entre un conjunto de buenas soluciones elegibles para su inclusión hasta tener una solución completa. En este problema en particular, una solución inicial se construye seleccionando al

azar un vértice, y los colores son evaluados hasta encontrar la solución que minimice el número de incompatibilidades con mínima rigidez logrado hasta el momento.

En la fase de VNS, en cada iteración se busca una solución vecino de la solución actual que aumenta el valor de la función objetivo. El algoritmo se detiene después de haber mejorado la solución 100 veces.

Para la fase de búsqueda local, VNS se utiliza porque da mejores soluciones que una simple búsqueda local, ya que puede escapar de óptimos locales fácilmente por el cambio sistemático de los vecindarios, lo que permite mejorar la calidad de la solución viene dada por el algoritmo GRASP.

```
Algoritmo híbrido GRASP/VNS para el Problema de Coloración Robusta

1. i := 0
2. Solución glotona v
3. j := 0
4. v ' := vecino(v)
5. Si f(v) ≤ f(v') entonces v := v'
6.   j := j+1
7. Si j ≤ 200 entonces ir al paso 4
8. Si f(v) < f(mejor)
9. mejor:= v
10. i:= i+1
11. Si i < 100 entonces ir al paso 2
12. Imprime mejor
```

El siguiente pseudocódigo muestra la estructura del algoritmo RS/VNS.

```
Algoritmo híbrido RS/VNS para el Problema de Coloración Robusta

1. Solución inicial aleatoria v, temperatura inicial T
2. v ' := vecino (v)
3. Si f(v') ≤ f(v) entonces v := v'
4. Si no
5. Si exp((f(v)-f(v'))/T) > No.Aleatorio(0,1) entonces v := v'
6. Si Número de actualizaciones< 200 entonces T := 0.9 T;
   regresa al paso 2
7. Imprime v
```

Recocido simulado. Una solución inicial se construye paso a paso seleccionando un elemento para su inclusión en la solución entre un conjunto de buenas soluciones candidatas para su inclusión. En este problema particular, una solución inicial se construye seleccionando al azar un vértice, y los colores son evaluados hasta que la solución que minimice el número de incompatibilidades más la rigidez lograda hasta ahora.

Variable Neighborhood Search. Proceso iterativo, donde en cada ciclo se busca una solución vecina de la solución actual que aumenta el valor de la función objetivo, este vecindario cambia aleatoriamente con una probabilidad determinada a priori.

6. Resultados

Se utilizaron las mismas instancias de prueba de las referencias [Ramírez 2003, Gutiérrez 2012, Frausto 2009, Lara 2005], con seis tipos diferentes de vecindades: 1, 2 y 3 cambios (es decir, se alteraban uno, dos o tres colores al azar en la solución para general tipos diferentes de vecindarios) así como 1, 2 y 3 permutaciones (realizando permutaciones en vértices con diferentes colores). Para cada vecindario se realizaron 10 réplicas en instancias de 20 y 30 vértices. Cabe mencionar que el comportamiento de los vecindarios en este tipo de algoritmos es similar sin importar el tamaño de instancia.

6.1 Optimización de parámetros para GRASP/VNS

A continuación se presentan los datos originales para el algoritmo GRASP donde cada columna está compuesta por 10 réplicas Cada columna fue obtenida con distintos porcentajes de cada tipo de vecindario. En la primera columna, alpha, se utilizó un vecindario con 1/6 de los vecindarios explorados con un cambio de colores y 1/6 con una permutación, un 1/6 con 2 cambios en los colores, 1/6 con 2 permutaciones, 1/6 con 3 cambios en los colores y 1/6 con 3 permutaciones.

En la segunda columna, beta, se muestran 10 réplicas las cuales tienen las variaciones de 1/3 de los vecindarios explorados con un cambio de colores, 1/3 de vecindarios explorados con una permutación, 1/12 de vecindarios explorados con 2 cambios de colores, 1/12 con 2 permutaciones; 1/12 de vecindarios explorados con 3 cambios en los colores y 1/12 con 3 permutaciones en dichos vecindarios.

En la tercer columna, gama, se vuelve a observar 10 réplicas pero en este caso existe la variación de ¼ de vecindarios explorados con un cambio de color y ¼ con una permutación, ¼ de vecindarios explorados con 2 cambios de color y ¼ con 2 permutaciones.

En la cuarta columna, delta, se muestran 10 réplicas, estas se obtuvieron utilizando ½ de los vecindarios explorados con un cambio de color y ½ con una permutación.

En la quinta columna, épsilon, se obtuvieron los datos utilizando solamente un cambio de color en todos los vecindarios explorados y en la última columna, zeta, se muestran los datos obtenidos con la variante de una permutación.

La siguiente tabla muestra los resultados obtenidos para 20 vértices y 7 colores.

10-20-30	20-25-30	15-30	30	1c	1p
Alpha	**Beta**	**Gamma**	**Delta**	**Épsilon**	**Zeta**
4.6934	4.7109	4.6934	4.6934	4.6934	4.6934
4.6934	4.6934	4.6934	4.6934	4.6934	4.6934
4.6934	4.7305	4.6934	4.6934	4.6934	4.6934
4.6934	4.7109	4.6934	4.6934	4.6934	4.6934
4.6934	4.7392	4.6934	4.6934	4.7392	4.6934
4.6934	4.6934	4.6934	4.6934	4.7185	4.771
4.6934	4.7305	4.6934	4.6934	4.6934	4.9128
4.6934	4.6934	4.6934	4.6934	4.6934	4.771
4.6934	4.7109	4.6934	4.6934	4.7109	4.6934
4.6934	4.6934	4.6934	4.6934	4.6934	4.7109

Tabla 1. Resultados obtenidos para el algoritmo GRASP/VNS para 20 vértices y 7 colores.

A continuación se observa los datos obtenidos en la ejecución del algoritmo *GRASP* para 30 vértices utilizando 10 colores.

10-20-30	20-25-30	15-30	30	1c	1p
Alpha	**Beta**	**Gamma**	**Delta**	**Épsilon**	**Zeta**
9.0296	8.3558	8.0879	7.5749	8.2959	8.1858
8.0243	8.2648	8.5058	7.5749	7.5748	8.2324
8.4065	8.7573	8.5448	7.5749	7.5748	8.4852
8.2324	8.526	8.169	7.5749	7.5748	8.8225
8.3912	8.6945	8.169	7.5749	7.5748	8.2324
8.2453	8.697	8.169	7.5749	7.5748	8.5655
8.2618	8.1986	8.169	7.5749	8.2715	7.8538
8.3715	8.2648	8.3634	7.5749	8.1858	8.3138
8.8433	9.0839	8.0623	7.5749	8.3012	8.4587
8.355	8.9124	7.8716	7.5749	8.3477	8.1985

Tabla 2.Resultados obtenidos para el algoritmo GRASP/VNS para 30 vértices y 10 colores.

Para los datos anteriores en el caso de 20 vértices los resultados son muy homogéneos (la instancia es fácil de optimizar debido a que es pequeña) por lo que se realizó una ANOVA

monomodal en la instancia de 30, que presenta mayores diferencias, para determinar si todos los tipos de vecindarios son iguales o si hay diferencia, obteniéndose las siguiente tabla:

Anova: un factor
Resumen

Grupos	*Cuenta*	*Suma*	*Promedio*	*Varianza*
Alpha	10	84.1609	8.4161	0.0895
Beta	10	85.7551	8.5755	0.0911
Gamma	10	82.1388	8.2139	0.0422
Delta	10	76.0279	7.6028	0.0077
Épsilon	10	79.2761	7.9276	0.1399
Zeta	10	83.3486	8.3347	0.0692

Análisis de varianza

Origen de variación	*Suma de cuadrados*	*Cuadrado medio*	*Valor P*	*F crítico*		
Entre grupos	6.3415	5	1.2683	17.3109	3.4216E-10	3.3769
Dentro de grupos	3.9564	54	0.0733			
Total	10.2979	59				

Tabla 3.ANOVA monomodal para la instancia de 30 resuelta por GRASP/VNS.

Dado que el valor de F_0 es mayor a la $F_{crítica}$ se concluye que si hay diferencias en las poblaciones, por otro lado, se graficó por tonos de color las tablas de datos para determinar visualmente cual es la mejor mezcla de vecindarios, obteniéndose para 30 vértices la siguiente gráfica (valores más obscuros son menores):

10-20-30	20-25-30	15-30	30	1c	1p
Alpha	Beta	Gamma	Delta	Épsilon	Zeta
9.02960	8.35580	8.08790	7.57490	8.29590	8.18580
8.02430	8.26480	8.50580	7.57490	7.57480	8.23240
8.40650	8.75730	8.54480	7.57490	7.57480	8.48520
8.23240	8.52600	8.16900	7.57490	7.57480	8.82250
8.39120	8.69450	8.16900	7.85380	7.57480	8.23240
8.24530	8.69700	8.16900	7.57490	7.57480	8.56550
8.26180	8.19860	8.19600	7.57490	8.27150	7.85380
8.37150	8.26480	8.36340	7.57490	8.18580	8.31380
8.84330		8.06230	7.57490	8.30120	8.45870
8.35500	8.91240	7.87160	7.57490	8.34770	8.19850

Tabla 4. Datos para 30 vértices con GRASP/VNS El fondo negro es el valor menor y se aclara a medida que crece.

De la tabla 4 se puede ver que los mejores resultados se obtuvieron con la combinación "Delta", que implica 50% de soluciones de un cambio y 50% de soluciones de una permutación. Por esta razón, para el resto de las instancias se utilizó la combinación "delta".

6.2 Optimización de parámetros para RS/VNS

Por otro lado, se muestran los datos originales para el algoritmo RS que al igual que en el algoritmo de *GRASP* cada columna está compuesta por 10 réplicas. Cada columna fue obtenida con distintos porcentajes de cada tipo de vecindario. En la primera columna, alpha, se utilizó un vecindario con 1/6 de los vecindarios explorados con un cambio de colores y 1/6 con una permutación, un 1/6 con 2 cambios en los colores, 1/6 con 2 permutaciones, 1/6 con 3 cambios en los colores y 1/6 con 3 permutaciones.

En la segunda columna, beta, se muestran 10 réplicas las cuales tienen las variaciones de 1/3 de los vecindarios explorados con un cambio de colores, 1/3 de vecindarios explorados con una permutación, 1/12 de vecindarios explorados con 2 cambios de colores, 1/12 con 2 permutaciones; 1/12 de vecindarios explorados con 3 cambios en los colores y 1/12 con 3 permutaciones en dichos vecindarios.

En la tercer columna, gama, se vuelve a observar 10 réplicas pero en este caso existe la variación de ¼ de vecindarios explorados con un cambio de color y ¼ con una permutación, ¼ de vecindarios explorados con 2 cambios de color y ¼ con 2 permutaciones.

En la cuarta columna, delta, se muestran 10 réplicas, estas se obtuvieron utilizando ½ de los vecindarios explorados con un cambio de color y ½ con una permutación.

En la quinta columna, épsilon, se obtuvieron los datos utilizando solamente un cambio de color en todos los vecindarios explorados y en la última columna, zeta, se muestran los datos obtenidos con la variante de una permutación.

La siguiente tabla muestra los resultados obtenidos para la instancia con 20 vértices y 7 colores.

10-20-30	20-25-30	15-30	30	1c	1p
Alpha	Beta	Gamma	Delta	Épsilon	Zeta
4.6934	4.9132	4.8932	4.6934	4.8391	8.9719
4.6934	4.9396	4.9752	4.7109	4.7109	5.7436
4.9624	4.7109	4.9633	4.6934	4.6934	4.9745
4.7392	4.7305	4.9265	4.6934	4.8373	12.1571
4.8391	4.8932	4.8373	4.7109	4.7109	5.8361
4.8391	4.7109	4.9265	4.6934	4.958	10.0452
4.8391	4.771	5.4729	4.7109	4.8791	7.1647
4.8373	4.7392	4.9752	4.7109	4.8373	7.0039
4.7185	4.9802	4.7109	4.6934	4.7109	9.2168
4.7392	4.8492	4.8373	4.7109	4.8492	12.8071

Tabla 5.Resultados obtenidos para el algoritmo RS/VNS para una instancia de 20.

A continuación se observa los datos obtenidos en la ejecución del algoritmo GRASP para la instancia con 30 vértices utilizando 10 colores.

10-20-30	20-25-30	15-30	30	1c	1p
Alpha	Beta	Gamma	Delta	Épsilon	Zeta
8.7367	8.7864	8.8226	8.2453	8.3862	12.801
7.5749	9.7104	8.6578	7.5749	7.5749	10.5187
8.8348	9.0631	9.434	7.9746	9.3472	12.8449
9.0663	8.6695	9.3761	7.8538	9.2327	15.969
9.0787	8.7726	10.3103	7.5749	8.1816	12.1742
8.8003	8.8186	8.9245	8.0879	8.2026	16.7998
8.3779	9.0735	9.0947	8.3372	7.5749	12.8701
7.9177	8.9192	8.7382	8.2618	7.8538	11.4287
8.9668	8.2422	9.3111	7.9746	8.6188	12.1301
8.8508	8.7384	8.8924	7.5749	8.7615	11.0767

Tabla 6.Resultados obtenidos para el algoritmo RS/VNS para una instancia de 30.

Para los datos obtenidos en el caso de 20 vértices los resultados son muy homogéneos al igual que ocurre con el algoritmo de GRASP (nuevamente, la instancia es fácil de optimizar debido a que es pequeña) por lo que se realizó una ANOVA monomodal en la instancia de 30, que presenta mayores diferencias, para determinar si todos los tipos de vecindarios son iguales o si hay diferencia, obteniéndose las siguiente tabla:

Anova: un factor

Resumen

Grupos	*Cuenta*	*Suma*	*Promedio*	*Varianza*
Alpha	10	86.2049	8.6205	0.2577
Beta	10	88.7939	8.8794	0.1397
Gamma	10	91.5617	9.1562	0.2381
Delta	10	79.4599	7.9460	0.0873
Épsilon	10	83.7342	8.3734	0.3899
Zeta	10	128.6132	12.8613	4.1006

Análisis de varianza

Origen de variación	*Suma de cuadrados*	*Cuadrado medio*	*Valor P*	*F crítico*		
Entre grupos	160.3400	5	32.0680	36.9073	2.9722E-16	3.3769
Dentro de grupos	46.9195	54	0.8689			
Total	207.2595	59				

Tabla 7. ANOVA monomodal para la instancia de 30.

Dado que el valor de f0 es mayor a la F crítica se concluye que si hay diferencias en las poblaciones como en el caso del algoritmo GRASP, y de igual forma, se graficó por tonos de color las tablas de datos para determinar visualmente cual es la mejor mezcla de vecindarios, obteniéndose para 30 vértices la siguiente gráfica (valores más obscuros son menores):

10-20-30	20-25-30	15-30	30	1c	1p
Alpha	Beta	Gamma	Delta	Épsilon	Zeta
8.73670	8.78640	8.82260	8.24530	8.38620	12.80100
7.57490	9.71040	8.65780	7.57490	7.57490	10.51870
8.83480	9.06310	9.43400	7.97460	9.34720	12.84490
9.06630	8.66950	9.37610	7.85380	9.23270	15.96900
9.07870	8.77260	10.31030	7.57490	8.18160	12.17420
8.80030	8.81860	8.92450	8.08790	8.20260	
8.37790	9.07350	9.09470	8.33720	7.57490	12.87010
7.91770	8.91920	8.73820	8.26180	7.85380	11.42870
8.96680	8.24220	9.31110	7.97460	8.61880	12.13010
8.85080	8.73840	8.89240	7.57490	8.76150	11.07670

Tabla 8. Datos para 30 vértices con GRASP/VNS El fondo negro es el valor menor y se aclara a medida que crece.

En este caso, la mejor combinación es nuevamente "delta" y aparentemente también "épsilon". Se utilizó en este caso la "delta" para el resto de los experimentos porque parece ser ligeramente mejor que "Épsilon".

6.3 Comparación de resultados obtenidos en las instancias AL.

A continuación se muestra una tabla comparativa con los resultados obtenidos con el resto de las instancias. En la primera columna se muestra las instancias Al, en la segunda *n* representa la cantidad de vértices y en la tercera columna *k* muestra el numero de colores. En la cuarta columna, se muestran los resultados obtenidos de un algoritmo Tabú [Ramírez 2000].

En la quinta columna, están los resultados obtenidos con el algoritmo GRASP, sin la combinación de otro algoritmo, se muestra el resultado del los dato y el tiempo de ejecución [Lara 2005]. En la sexta columna se muestran los resultado del algoritmo GRASP/VNS. La séptima columna presenta los resultados de un algoritmo de Búsqueda Dispersa realizado anteriormente [Gutiérrez 2011] y finalmente, en la última columna, se encuentran los resultados obtenidos del algoritmo RS/VNS. El valor incluido para nuestros algoritmos es el mejor de 5 corridas por instancia y por color.

G_n,0.5	N	k	Tabú	GRASP	GRASP/VNS	Scatter S.	RS/VNS
al(20)	20	7	7.0970 / < 1	7.1423 / 0.14	6.9446/6.7503	6.9046 / 1.3	6.9046/37.209
al(20)	20	8	4.7710 / <1	4.6934 / 0.15	4.6934/4.5166	4.6934 / 1.3	4.6934/35.0828
al(30)	30	10	8.0623 / <1	7.5749 / 0.44	7.5748/13.728	7.5749 / 3.6	7.5748/42.4382
al(30)	30	11	6.0565 / <1	5.9318 / 0.49	5.8890/13.2535	5.8890 / 3.6	5.9318/52.921
al(40)	40	14	7.1709 / 15	7.3950 / 1.0	7.0837/30.8917	7.0837 / 8.5	4.4876/49.0398
al(40)	40	15	5.8173 / 14	6.3117 / 1.0	5.8707/30.1628	5.6708 / 7.2	5.9542/48.763
al(50)	50	17	9.8259 / 33	8.9531 / 1.9	8.9541/62.279	8.2587 / 13	8.8191/68.9818
al(50)	50	18	7.4966 / 33	7.1464 / 1.9	7.2036/63.4753	6.7164 / 12	7.9752/69.9698
al(60)	60	20	9.8331 / 69	9.9687 / 3.3	9.7206/125.327	8.8676 / 19	10.1977/128.822
al(60)	60	21	8.2181 / 69	8.1430 / 3.4	8.0473/118.269	7.2380 / 20	7.6153/69.7598
al(70)	70	24	11.1307 / 128	11.2388 / 5.31	10.5508/212.552	9.2634 / 36	10.2291/210.8057
al(70)	70	25	9.5478 / 128	9.2145 / 5.56	9.2349/203.722	7.7048 / 33	8.4124/115.2565
al(80)	80	27	11.1946 / 218	11.7512 / 8.0	11.8461/333.38	9.9835 / 52	11.6413/85.427
al(80)	80	28	10.5845 / 219	10.2631 / 8.2	10.1906/345.465	8.5961 / 43	9.7847/105.662
al(90)	90	30	12.2832 / 350	13.4919 / 11.5	12.6237/523.729	10.8911 / 47	13.2353/146.296
al(90)	90	31	11.3699 / 349	11.5060 / 11.9	10.4708/534.496	9.5008 / 89	11.5109/158.493
al(100)	100	34	12.1932 / 544	12.8675 / 15.8	12.1251/979.206	10.0470 / 123	12.0477/276.753
al(100)	100	35	12.0650 / 885	11.1317 / 15.6	11.2543/832.09	9.4259 / 124	10.8755/240.69
al(110)	110	37	----------	12.7681 / 20.7	12.0703/1425.02	10.8463 / 149	13.3544/317.45

al(110)	110	38	----------	11.6574 / 20.2	12.302/550.657	9.9558 / 171	11.9499/346.959
al(120)	120	40	----------	15.0014 / 26.8	14.7965/704.39	11.3507 / 190	14.3225/264.974
al(120)	120	41	----------	13.5266 / 27.5	12.9872/587.553	10.1258 / 191	11.5402/296.158

Tabla 9. Tabla comparativa con los resultados obtenidos con el resto de las instancias.

7. Conclusiones

Se desarrollaron dos algoritmos metaheurísticos híbridos para resolver el problema de coloración robusta: el primero es un GRASP/VNS y el segundo un RS/VNS. Ambos algoritmos presentan algunas mejorías en los resultados pero existe un aumento en el tiempo de ejecución, por ejemplo:

Al comparar el método Tabu con GRASP/VNS se puede observar que en la instancia al(20) con 20 vértices y 7 colores, Tabu presenta un resultado de 7.0970 y un tiempo menor a un segundo mientras que GRASP/VNS arroja un resultado de 6.9446 con 6.7503 segundos lo cual indica que existe una mejoría para encontrar un resultado óptimo. Por otro lado comparando este resultado con el método GRASP existe una mejoría, ya que en GRASP se obtiene 7.1423 con 0.14 segundos.

En otro ejemplo, en la instancia al(90) con 90 vértices y 31 colores. Tabu presenta un resultado de 11.3699 con un tiempo de 349 segundos y GRASP con un resultado de 11.5060 con un tiempo de 11.9 segundos y GRASP/VNS arrojo un resultado de 10.4708 con un tiempo de ejecución de 534.496 segundos, se observa una mejora en el resultado pero hay un aumento en el tiempo de ejecución.

En la instancia al(110) con 110 vértices y 37 colores; Tabu no presenta un resultado ya que nunca encontró un resultado óptimo. GRASP muestra un valor de 12.7681 y un tiempo de 20.7 y GRASP/VNS tiene un resultado de 12.0703 con 1425.02 segundos. Aunque hay un aumento en el tiempo de ejecución en este último existe una notable mejoría.

Por otro lado, haciendo comparación entre el algoritmo Tabú, Scatter S. y RS/VNS, en la instancia al(20) con 20 vértices y 7 colores, Scatter S. muestra un resultado de 6.9046 con 1.3 segundos y en RS/VNS tiene el mismo resultado de 6.9046 con un tiempo de 37.209 segundos y en Tabu es de 7.0970 y un tiempo menor a un segundo. En la instancia al(40) con 40 vértices y 14 colores, Tabu muestra un resultado de 7.1709 con 15 segundos, Scatter S. presenta 7.0837 y un tiempo de 8.5 segundos y por último RS/VNS presenta una mejoría con 4.4876 con un tiempo de ejecución de 49.0398 segundos.

8. Referencias

[Avanthay 2003] C. Avanthay, A. Hertz, N. Zufferey. A Variable Neighborhood Search For Graph Coloring. European Journal of Operational Research 151 (2003) 379–388.

[Brownlee 2012] J. Brownlee. Cleaver Algorithms. Nature Inspired Programming Recipes. Licensed Under Creative Commons. 2012.

[Diestel 2000] R. Diestel, *Graph Theory*. Springer-Verlag, New York (Electronic Edition), 2000.

[Frausto 2009] I. Frausto Benítez, *"Implementación de un algoritmo de búsqueda tabú para resolver el problema de coloración robusta"*, proyecto terminal, División de Ciencias Básicas e Ingeniería, Universidad Autónoma Metropolitana Azcapotzalco, México D.F., 2009.

[Gutiérrez 1991] M. A. Gutiérrez Andrade, *"Técnica de recocido simulado y sus aplicaciones"*, tesis de doctorado, Facultad de Ingeniería, Universidad Nacional Autónoma de México, México D.F., 1991.

[Gutiérrez 2011] M. A. Gutiérrez Andrade, P. Lara Velázquez, R. López Bracho, J. Ramírez Rodríguez, *Heuristics for the Robust Coloring Problem*. Revista de Matemática: Teoría y Aplicaciones, 18(1): 137-147, 2011.

[Hansen, 1997] "Variable neighborhood search". *Computers and Operations Research* 24 (11): 1097–1100.

[Lara 2005] P. Lara Velázquez, *"Un algoritmo evolutivo para resolver el problema de coloración robusta"*, tesis de doctorado, Facultad de Ingeniería, Universidad Nacional Autónoma de México, México D.F., 2005.

[Martí 2003] R. Martí. *Procedimientos metaheurísticos en optimización combinatoria*. Matematiques, vol. 1, No 1, pp. 3-62, 2003.

[Metropolis 1953] N. Metropolis, M. Rosenbluth, A Rosenbluth, A. Teller and E Teller. *Equation of State Calculations by Fast Computing Machines.* Journal of Chemical Physics 21, pp. 1087-1092.

[Ramírez 2000] J. Ramírez, *Extensiones del problema de coloración de grafos*. Tesis de Doctorado, Facultad de Ciencias Matemáticas, Universidad Complutense de Madrid, 2000.

[Resende 2002] M Resdende, C. Ribeiro, *Greedy adaptive Search procedures*, AT & T Labs Research Technical Reporte TD-53RSJY 2002.

[Toda 1983] M Toda, R Kubo, N Saito. Statistical Physics, Springer Verlag, 1983.

[Yañez 2003] J. Yañez, J. Ramirez, *The Robust Coloring Problem*. European Journal of Operational Research, Vol. 148, No. 3, 2003, pp.546-558.

Apéndice A

Código del algoritmo híbrido GRASP/VNS en lenguaje C.

```
// Biblioteca Estándard de C

#include <stdio.h>

#include <time.h>

#include <stdlib.h>

#define MAX 150

//Declaración de variables globales

int nver, ncol;

float mat[MAX][MAX], col[MAX]={0}, col2[MAX]={0}, col3[MAX]={0},
rig=0, inc=0;

double alpha=.7;

// Función carga_datos: lee los archivos que contienen las
matrices original y complementaria.

void carga_datos(){

      FILE *arch1;

    int i, j;

    arch1=fopen("al40.txt","r");

    if(arch1==NULL){

        printf("No se encontró el archivo");

        return;}

    fscanf(arch1,"%i",&nver); fscanf(arch1,"%i",&ncol);

    for(i=1; i<=nver;i++){

        for(j=1; j<=nver;j++){

            fscanf(arch1,"%f",&mat[i][j]);

        }

    }
```

```
}

//Función rigidez: calcula el valor de una función multiobjetivo
ponderada mediante un parámetro alfa predeterminada, que considera
las funciones mono-objetivo de número de aristas incompatibles y
el valor de la suma de las ponderaciones en las aristas
complementarias que comparten el mismo color en sus vértices
extremos.

float rigidez(float col3[50]){

      int i,j;

      col3[nver+1]=0; col3[nver+2]=0;

      for(i=1;i<=nver;i++){

            for(j=i+1;j<=nver;j++){

                  if(col3[i]==col3[j]){

                        if(mat[i][j]==0){col3[nver+1]++;}

                        else col3[nver+2]+=mat[i][j];

                  }

            }

      }

    return alpha*col3[nver+1]+(1-alpha)*col3[nver+2];

}

//Función ini_rnd: Aleatoriza el conjunto de vértices

void ini_rnd(){

      int i;

    for(i=1;i<=nver;i++){

            col[i]=rand()%ncol+1;

      }

      rigidez(col);

}
```

```
// Función shuffle: Genera un orden de los vértices para ser coloreados dentro de la función ini_gl
void shuffle(int vec[MAX], int n){
	int i,sel;
	float aux;
	for (i=1;i<=n;i++) vec[i]=i;
	for (i=n;i>=1;i--){
		sel=rand()%(i)+1;
		aux=vec[i];
		vec[i]=vec[sel];
		vec[sel]=aux;
		//printf("%i ",vec[i]);
	}
	//printf("\n\n");
}
// Función ini_gl: construye una solución inicial paso a paso en el orden definido por la función shuffle y en cada uno de ellos selecciona un elemento para incluirlo en la solución que sea el mejor entre todos aquellos que sean elegibles para incluirse.
void ini_gl(){
	int i, j, k, ocol[MAX],over[MAX];
	float rigpar,mrigpar;
	shuffle(over,nver);
	for(i=1;i<=nver+2;i++) {col[i]=0;}

	for(i=1;i<=nver;i++){
		shuffle(ocol,ncol);
		mrigpar=1e6;
```

```
            for(j=1;j<=ncol;j++){
                  rigpar=0.0;
                  for(k=1;k<=nver;k++){
                        if(ocol[j]==col[k]) {
                            if(mat[over[i]][k]==1 ||
mat[k][over[i]]==1){

rigpar+=alpha*(mat[over[i]][k]+mat[k][over[i]]);
                            }
                            else {rigpar+=(1-
alpha)*(mat[over[i]][k]+mat[k][over[i]]);}


                       }
                       }
                       if(rigpar<mrigpar){
             col[over[i]]=ocol[j]; mrigpar=rigpar;
                       //printf("col[%i]= %.0f %i\n",i,col[i],j);
                       }
            }


      }
      rigidez(col);
}
//Función imprime: Imprime el vector de coloración indicado,
incluyendo incompatibilidad y rigidez.
void imprime(float col3[50]){
            int i;
    for(i=1;i<=nver;i++){
```

```
            printf("%.0f ",col3[i]);
      }
      printf(" - %.0f  %f\n",col3[nver+1],col3[nver+2]);
}
// Función vecino: en cada iteración se busca una solución vecina
de las soluciones actuales que mejoren el valor de la función
objetivo.
void vecino(float col3[50],int n){
    int rver, rver2, rcol,aux;
   /* if(n<=60){ // 1 cambio
        rver=rand()%nver+1;
        do{
                rcol=rand()%ncol+1;
           }while(col3[rver]==rcol);
        col3[rver]=rcol;
        rigidez(col3);
    }*/
    /*if(n<=20){ //2 cambio
        rver=rand()%nver+1;
        do{
                rcol=rand()%ncol+1;
           }while(col3[rver]==rcol);
        col3[rver]=rcol;

        rver=rand()%nver+1;
        do{
                rcol=rand()%ncol+1;
           }while(col3[rver]==rcol);
```

```
        col3[rver]=rcol;
        rigidez(col3);
    }*/

/*if(n<30){ //3 cambio
        rver=rand()%nver+1;
        do{
                rcol=rand()%ncol+1;
           }while(col3[rver]==rcol);
        col3[rver]=rcol;

        rver=rand()%nver+1;
        do{
                rcol=rand()%ncol+1;
           }while(col3[rver]==rcol);
        col3[rver]=rcol;

        rver=rand()%nver+1;
        do{
                rcol=rand()%ncol+1;
           }while(col3[rver]==rcol);
        col3[rver]=rcol;
        rigidez(col3);
    }*/
     if(n<=60) { // 1 permut
           do{
          rver=rand()%nver+1;
```

```
      rver2=rand()%nver+1;
    }while(col3[rver]==col3[rver2]);
    aux=col3[rver];
    col3[rver]=col3[rver2];
    col3[rver2]=aux;
    rigidez(col3);
}
/*else if(n<=55){ // 2 permut
       do{
      rver=rand()%nver+1;
      rver2=rand()%nver+1;
    }while(col3[rver]==col3[rver2]);
    aux=col3[rver];
    col3[rver]=col3[rver2];
    col3[rver2]=aux;
    rigidez(col3);
}*/
/*else{ //3 permut
       do{
      rver=rand()%nver+1;
      rver2=rand()%nver+1;
    }while(col3[rver]==col3[rver2]);
    aux=col3[rver];
    col3[rver]=col3[rver2];
    col3[rver2]=aux;

       do{
```

```
            rver=rand()%nver+1;
            rver2=rand()%nver+1;
        }while(col3[rver]==col3[rver2]);
        aux=col3[rver];
        col3[rver]=col3[rver2];
        col3[rver2]=aux;
     do{
            rver=rand()%nver+1;
            rver2=rand()%nver+1;
        }while(col3[rver]==col3[rver2]);
        aux=col3[rver];
        col3[rver]=col3[rver2];
        col3[rver2]=aux;
        rigidez(col3);
    }*/
}
// Función copy: Copia la mejor coloración encontrada.
void copy(float col1[50],float col2[50]){
    int i;
    for(i=1; i<nver+3;i++){
        col1[i]=col2[i];
    }
}
// Función main: función principal que ejecuta la función GRASP,
realizando 100 iteraciones y en cada una de ellas, se genera una
solución glotona con una mejora local considerando VNS.
int main(){
    int i,j, noimpr=0,tempo;
```

```
    tempo=time(NULL);
     srand(time(NULL));
     carga_datos();
     ini_gl();
     imprime(col);
    //return 0;

     copy(col3,col2);
     for (i=1; i<=100; i++){
         ini_gl(); copy(col2,col); imprime(col);
         alpha=0.7;
         noimpr=0; //alpha=.3+(rand()%5)/10.0;
         //for(j=1; j<5e3;j++){
         while(noimpr<100*nver){

             vecino(col,rand()%60);
             if(rigidez(col)<=rigidez(col2)){
                 copy(col2,col);
                 noimpr=0;
            }
            else {
                copy(col,col2);
                noimpr++;
            }
        }
        if(rigidez(col2)<rigidez(col3) &&
col2[nver+1]==0){copy(col3,col2);}
```

```
        imprime(col2); printf("\n");
      }
      printf("Tiempo de ejecución: %li segs.\n",time(NULL)-tempo);
      imprime(col3);
    return 0;
}
```

Apéndice B

Código del algoritmo híbrido RS/VNS en lenguaje C.

```
//Biblioteca Estándard de C
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#define MAX 150

//Declaración de variables globales
int nver, ncol;
float mat[MAX][MAX], col[MAX]={0}, col2[MAX]={0}, col3[MAX]={0},
rig=0, inc=0;
double alpha=.7;
// Función carga_datos: lee los archivos que contienen las
matrices original y complementaria.
void carga_datos(){
     FILE *arch1;
    int i, j;
    arch1=fopen("al40.txt","r");
    if(arch1==NULL){
        printf("No se encontró el archivo");
        return;}
    fscanf(arch1,"%i",&nver); fscanf(arch1,"%i",&ncol);
    for(i=1; i<=nver;i++){
        for(j=1; j<=nver;j++){
            fscanf(arch1,"%f",&mat[i][j]);
        }
```

```
    }
}
//Función rigidez: calcula el valor de una función multiobjetivo
ponderada mediante un parámetro alfa predeterminada, que considera
las funciones mono-objetivo de número de aristas incompatibles y
el valor de la suma de las ponderaciones en las aristas
complementarias que comparten el mismo color en sus vértices
extremos.
float rigidez(float col3[50]){
       int i,j;
       col3[nver+1]=0; col3[nver+2]=0;
       for(i=1;i<=nver;i++){
              for(j=i+1;j<=nver;j++){
                     if(col3[i]==col3[j]){
                            if(mat[i][j]==0){col3[nver+1]++;}
                            else col3[nver+2]+=mat[i][j];
                     }
              }
       }
    return alpha*col3[nver+1]+(1-alpha)*col3[nver+2];
}
//Función ini_rnd: Aleatoriza el conjunto de vértices
void ini_rnd(){
       int i;
    for(i=1;i<=nver;i++){
              col[i]=rand()%ncol+1;
       }
       rigidez(col);
```

```
}

// Función shuffle: Genera un orden de los vértices para ser
coloreados dentro de la función ini_gl

void shuffle(int vec[MAX], int n){

	int i,sel;

	float aux;

	for (i=1;i<=n;i++) vec[i]=i;

	for (i=n;i>=1;i--){

		sel=rand()%(i)+1;

		aux=vec[i];

		vec[i]=vec[sel];

		vec[sel]=aux;

		//printf("%i ",vec[i]);

	}

	//printf("\n\n");

}

// Función ini_gl: construye una solución inicial paso a paso en
el orden definido por la función shuffle y en cada uno de ellos
selecciona un elemento para incluirlo en la solución que sea el
mejor entre todos aquellos que sean elegibles para incluirse.

void ini_gl(){

	int i, j, k, ocol[MAX],over[MAX];

	float rigpar,mrigpar;

	shuffle(over,nver);

	for(i=1;i<=nver+2;i++) {col[i]=0;}

	for(i=1;i<=nver;i++){

		shuffle(ocol,ncol);

		mrigpar=1e6;
```

```
            for(j=1;j<=ncol;j++){
                    rigpar=0.0;
                    for(k=1;k<=nver;k++){
                           if(ocol[j]==col[k]) {
                               if(mat[over[i]][k]==1 ||
mat[k][over[i]]==1){
rigpar+=alpha*(mat[over[i]][k]+mat[k][over[i]]);
                               }
                               else {rigpar+=(1-
alpha)*(mat[over[i]][k]+mat[k][over[i]]);}
                 }
                    }
                    if(rigpar<mrigpar){
            col[over[i]]=ocol[j]; mrigpar=rigpar;
                    //printf("col[%i]= %.0f %i\n",i,col[i],j);
                    }
              }
       }
       rigidez(col);
}
//Función imprime: Imprime el vector de coloración indicado,
incluyendo incompatibilidad y rigidez.
void imprime(float col3[50]){
              int i;
    for(i=1;i<=nver;i++){
              printf("%.0f ",col3[i]);
       }
       printf(" - %.0f  %f\n",col3[nver+1],col3[nver+2]);
```

```
}

// Función vecino: en cada iteración se busca una solución vecina
de las soluciones actuales que mejoren el valor de la función
objetivo.

void vecino(float col3[50],int n){

    int rver, rver2, rcol,aux;

    /*if(n<=60){ // 1 cambio

        rver=rand()%nver+1;

        do{

                    rcol=rand()%ncol+1;

              }while(col3[rver]==rcol);

        col3[rver]=rcol;

        rigidez(col3);

    }

    /*if(n<=25){ //2 cambio

        rver=rand()%nver+1;

        do{

                    rcol=rand()%ncol+1;

              }while(col3[rver]==rcol);

        col3[rver]=rcol;

        rver=rand()%nver+1;

        do{

                    rcol=rand()%ncol+1;

              }while(col3[rver]==rcol);

        col3[rver]=rcol;

        rigidez(col3);

    }
```

```
if(n<30){ //3 cambio
        rver=rand()%nver+1;
        do{
                    rcol=rand()%ncol+1;
              }while(col3[rver]==rcol);
        col3[rver]=rcol;
        rver=rand()%nver+1;
        do{
                    rcol=rand()%ncol+1;
              }while(col3[rver]==rcol);
        col3[rver]=rcol;
        rver=rand()%nver+1;
        do{
                    rcol=rand()%ncol+1;
              }while(col3[rver]==rcol);
        col3[rver]=rcol;
        rigidez(col3);
   }*/
    if(n<=60) { // 1 permut
              do{
          rver=rand()%nver+1;
          rver2=rand()%nver+1;
        }while(col3[rver]==col3[rver2]);
        aux=col3[rver];
        col3[rver]=col3[rver2];
        col3[rver2]=aux;
```

```
    rigidez(col3);
}
/*else if(n<=55){ // 2 permut
          do{
      rver=rand()%nver+1;
      rver2=rand()%nver+1;
    }while(col3[rver]==col3[rver2]);
    aux=col3[rver];
    col3[rver]=col3[rver2];
    col3[rver2]=aux;
    rigidez(col3);
}
else{ //3 permut
          do{
      rver=rand()%nver+1;
      rver2=rand()%nver+1;
    }while(col3[rver]==col3[rver2]);
    aux=col3[rver];
    col3[rver]=col3[rver2];
    col3[rver2]=aux;

          do{
      rver=rand()%nver+1;
      rver2=rand()%nver+1;
    }while(col3[rver]==col3[rver2]);
    aux=col3[rver];
```

```
        col3[rver]=col3[rver2];
        col3[rver2]=aux;
        do{
          rver=rand()%nver+1;
          rver2=rand()%nver+1;
        }while(col3[rver]==col3[rver2]);
        aux=col3[rver];
        col3[rver]=col3[rver2];
        col3[rver2]=aux;
        rigidez(col3);
    }*/
}
// Función copy: Copia la mejor coloración encontrada.
void copy(float col1[50],float col2[50]){
    int i;
    for(i=1; i<nver+3;i++){
        col1[i]=col2[i];
    }
}
// Función rnd: Genera un número aleatorio entre 0 y 1 quasi-
uniforme a partir de la función rand()nativa de C.
float rnd(){
    return rand()/32768.0;
}
// Función main: función principal que ejecuta la función RS,
realizando 100 iteraciones y en cada una de ellas, se genera una
solución glotona con una mejora local considerando VNS.
```

```
int main(){
    int noimpr=0,tempo;
    float aux, temp=1.0;
    tempo=time(NULL);
       srand(time(NULL));
       carga_datos();
       ini_gl();
       //imprime(col);
    //return 0;

       //for (i=1; i<=100; i++){
           ini_rnd(); copy(col2,col); imprime(col);
           copy(col3,col2);
           noimpr=0;
           while(noimpr<500*nver){
               vecino(col,rand()%60);
               aux=rigidez(col)-rigidez(col2);
               if(exp(-aux*1e-7/temp)>rnd() || aux<=0.0){
                   copy(col2,col);
                   imprime(col2); printf("\n");
                   noimpr=0;
                   temp*=.99;
            }
            else {
                copy(col,col2);
                noimpr++;
```

```
            }
            //rigidez(col2);rigidez(col3);
            if(rigidez(col2)<rigidez(col3) && col2[nver+1]==0){
                copy(col3,col2);
                printf("HOLA!!!");
                imprime(col3); printf("1\n");
                //imprime(col2); printf("1\n");
            }
         }
         //imprime(col3); printf("\n");
        //}
        printf("Tiempo de ejecucion: %li segs.\n",time(NULL)-tempo);
        imprime(col3);
    return 0;

}
```