Trabajo Fin de Máster

El Problema de Coloración de Grafos

Sergio Pena Seijas

Máster en Técnicas Estadísticas

Curso 2016-2017

Propuesta de Trabajo Fin de Máster

Título en galego: O problema de coloración de grafos

Título en español: El problema de coloración de grafos

English title: The graph colouring problem

Modalidad: Modalidad A

Autor: Sergio Pena Seijas, Universidad de Santiago de Compostela

Directora: Silvia Lorenzo Freire, Universidad de A Coruña

Breve resumen del trabajo:

Uno de los problemas más conocidos de Teoría de Grafos es el problema de coloración de grafos. Tal y como indica su nombre, se trata de asignar etiquetas denominadas colores a los elementos de un grafo. Hay varios tipos de problemas de coloración de grafos, siendo el más conocido el problema de coloración de los vértices de un grafo.

El objetivo en estos problemas es asignar colores a los vértices de un grafo, de tal forma que los vértices adyacentes no compartan el mismo color. Suele ser habitual además buscar el número cromático, que sería el mínimo número de colores necesario para la coloración del grafo. Los problemas de coloración de grafos tienen múltiples aplicaciones. De hecho, se pueden utilizar para resolver muchos problemas de asignación.

El objetivo principal de este proyecto fin de máster sería hacer una revisión bibliográfica del problema de coloración de grafos, centrándose principalmente en el problema de coloración de los vértices de un grafo.

Además, en función de las preferencias y conocimientos del alumno, también se podrían contemplar otras opciones, como hacer un paquete de R que recoja alguna de las técnicas de resolución existentes en la literatura para resolver este tipo de problemas o realizar una revisión bibliográfica de los principales resultados de Teoría de Juegos relacionados con este problema.

Recomendaciones:

Es recomendable que el alumno tenga conocimientos de programación lineal.

Otras observaciones:

Doña Silvia Lorenzo Freire, Profesora Titular de Universidad de la Universidad de A Coruña, informa que el Trabajo Fin de Máster titulado

El Problema de Coloración de Grafos

fue realizado bajo su dirección por don Sergio Pena Seijas para el Máster en Técnicas Estadísticas. Estimando que el trabajo está terminado, da su conformidad para su presentación y defensa ante un tribunal.

En Santiago de Compostela, a 4 de Septiembre de 2017.

La directora:	El autor:
Doña Silvia Lorenzo Freire	Don Sergio Pena Seijas

Índice general

Resumen

Dentro del amplio campo de la *Teoría de Grafos*, en este trabajo realizamos una revisión del *Problema de Coloración de Grafos*. En concreto, nos centraremos en la coloración de los vértices de un grafo. Para ello, presentamos en el capítulo inicial una breve introducción al problema desde su aparición a través del famoso *Teorema de los Cuatro Colores*, y revisando los avances e inconvenientes que han ido surgiendo a medida que se ha profundizado en este problema. En el segundo capítulo, hacemos una revisión de aquellos algoritmos más relevantes asociados al problema de coloración de grafos, analizando cuáles son los pros y contras de cada uno de ellos. Finalmente, introducimos un conjunto de aplicaciones derivadas del análisis del *Problema de Coloración de Grafos*, presentes en nuestro día a día y que hacen ver dicho problema desde un punto de vista más práctico.

Abstract

Within the broad field of the *Graph Theory*, in this work we perform a review of the *Graph Colouring Problem*. Specifically, we will focus on the vertex colouring of a graph. For this, we present in the initial chapter a brief introduction to the problem wich originates in the famous *Four Colour Theorem*, reviewing the advances and inconveniences found in the study of the problem. In the second chapter, we review the most relevant algorithms associated with the graph colouring problem, analyzing what are the pros and cons of each one of them. Finally, we introduce a set of applications derived from the analysis of the *Graph Colouring Problem*, present in our daily life and that give us the opportunity to see that problem from a practical point of view.

Capítulo 1

Introducción a la Coloración de Grafos

La *Teoría de Grafos* surge como concepto por primera vez en el siglo XVIII, de la mano de Leonhard Euler (1707-1783), planteando el conocido *Problema de los siete puentes de Königsberg*, (en la actualidad Kaliningrado, Rusia), problema que resuelve en el año 1736.
Dicho problema consistía en encontrar un camino tal que, pasando una única vez por cada uno de los puentes, se pudiera regresar al punto de partida. La respuesta a dicho problema es que no existe ningún camino cumpliendo esas características.

En su demostración, representó cada puente como una línea (arista) uniendo dos puntos (vértices), cada uno de los cuales se correspondía con una región diferente. Sentenció la demostración indicando que los puntos intermedios de un posible recorrido tienen que estar necesariamente conectados a un número par de aristas, de forma que los puntos inicial y final serían los únicos que podrían estar conectados por un número impar de líneas, lo cual no puede suceder pues este debe ser el mismo. En la Figura 1.1, podemos ver cómo planteó la demostración al problema.

Figura 1.1: Problema de los siete puentes de Königsberg

Posteriormente, en el año 1847, Gustav Kirchhoff utiliza la *Teoría de Grafos* para analizar redes eléctricas, publicando las *Leyes de Kirchhoff*, permitiendo calcular así el voltaje y la corriente en circuitos eléctricos.

Es en el año 1852 cuando Francis Guthrie, que había sido alumno de Augustus De Morgan en la University College de Londres, y que una vez finalizados sus estudios de Matemáticas se encontraba estudiando Derecho, le enseña a su hermano Frederick, que por aquel entonces era alumno de De Morgan, una serie de resultados que había estado intentando demostrar sobre la coloración de mapas (coloreando un mapa de los condados de Inglaterra, se dio cuenta de que sólo cuatro colores eran necesarios para asegurar que todos los condados vecinos se asignaban a colores diferentes), para hacérselo llegar a De Morgan.

Siendo De Morgan incapaz de dar una respuesta al problema, el mismo día escribe la siguiente carta dirigida al matemático irlandés Hamilton:

"A student of mine asked me today to give him a reason for a fact which I did not know was a fact - and do not yet. He says that if a figure be anyhow divided and the compartments differently coloured so that figures with any portion of common boundary line are differently coloured - four colours may be wanted, but not more - the following is the case in which four colours are wanted. Query cannot a necessity for five or more be invented. ..... If you retort with some very simple case which makes me out a stupid animal, I think I must do as the Sphynx did...."

El propio Hamilton, que en ese momento se encontraba trabajando en la Teoría de Cuaternios, le responde de la siguiente manera:

"I am not likely to attempt your quaternion of colour very soon."

Durante años, numerosos matemáticos de prestigio trataron de obtener una demostración para la *Conjetura de los Cuatro Colores*, hasta que en el año 1879, Alfred Bray Kempe, abogado de Londres que había estudiado matemáticas en Cambridge, anuncia en *Nature*, que ha encontrado una demostración para la *Conjetura de los Cuatro Colores*. Ese mismo año, Cayley, que había sido profesor suyo, le sugiere presentar el Teorema en la revista *American Journal of Mathematics*. En su demostración, se apoya en el *Método de las Cadenas de Kempe*. Dicho concepto consiste en lo siguiente:

Definición 1.0.1. Sea $\mathcal{S} = \{S_1, \ldots, S_k\}$ una solución factible (ver Definición 1.1.2). Dado un vértice $v \in S_i$ y una segunda clase S_j $(1 \leq i \neq j \leq k)$, una *Cadena de Kempe* se define como un subgrafo conexo (ver Definición 1.1.1) que contiene a v, y que sólo comprende vértices coloreados con los colores i y j. Un *Intercambio de Cadena de Kempe* implica tomar una cadena de Kempe e intercambiar los colores de todos los vértices de la cadena.

Sin embargo, en 1890, Percy John Heawood publica el artículo *Map colouring theorem*, en el que indica:

"...rather destructive than constructive, for it will be shown that there is a defect in the now apparently recognised proof."

De esta forma, el Teorema volvía a ser considerado Conjetura. En el mismo artículo, Heawood utiliza el mismo argumento que Kempe para probar el siguiente:

Teorema 1.0.2. *Los vértices de cualquier grafo plano que no contiene lazos son* 5*-coloreables.*

Debemos tener en cuenta que, en el contexto de la teoría de grafos, un mapa puede ser representado por un grafo plano sin puentes G, con las caras de G representando las distintas regiones del mapa, las aristas representando la frontera entre regiones, y los vértices representando los puntos de intersección de las fronteras de las regiones. Podemos ver un ejemplo sobre el mapa de Australia en la Figura 1.2.

La importancia de este resultado reside en el hecho de que, dado un grafo plano (es posible dibujar en un plano de forma que ninguna de sus aristas se corten), conexo (ver Definición 1.1.1) y sin lazos (aristas cuyos extremos inciden sobre el mismo vértice), G, y su dual, G^*, entonces los vértices de G son k-coloreables si y sólo si las caras de G^* son k-coloreables.

Dado un grafo plano G, el dual de G, denotado G^*, se construye de la siguiente forma. En primer lugar, dibujamos un único vértice v_i^* dentro de cada cara F_i de G. Después, para cada arista e de G, dibujamos una línea e^* que corta e pero ninguna otra arista de G, y que conecta los dos vértices de G^*

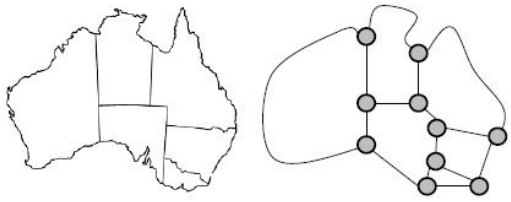


Figura 1.2: Territorios de Australia (izquierda) y su correspondiente grafo plano (derecha).

correspondientes a las dos caras de G que e está separando. En la Figura 1.3 podemos ver un ejemplo de cómo se obtiene el grado dual de uno grafo plano.

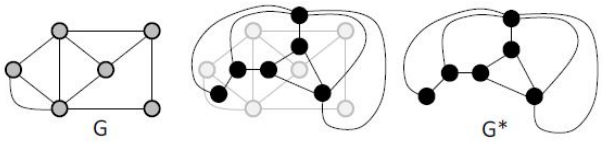


Figura 1.3: Transformación de un grafo G en su dual G^*

Pasaron años de investigación en torno al problema, centrados en demostraciones sobre los vértices de grafos planos sin lazos, es decir, los grafos duales asociados a un mapa.

Observación 1.0.3. *Una* configuración *es parte de una triangulación (disposición de un grafo plano en el que las caras están delimitadas por tres aristas, formando así triángulos) contenida en un circuito, es decir, planteando un grafo plano como un circuito eléctrico, siendo este un ciclo (camino cerrado contenido en el grafo). Un* conjunto inevitable *es un conjunto de configuraciones con la propiedad de que cualquier triangulación debe contener una de las configuraciones del conjunto.*

Finalmente, en el año 1976, los matemáticos Kenneth Appel y Wolfgang Haken demostraron que no puede existir ninguna configuración en un contraejemplo mínimo del *Teorema de los Cuatro Colores*. Basaron su demostración en la reducibilidad, utilizando las *Cadenas de Kempe*. Redujeron el número de configuraciones del conjunto inevitable a 1.936, los cuales fueron comprobados individualmente con ayuda del ordenador. Utilizaron 1200 horas de tiempo computacional para obtener la prueba final. En la Figura 1.4 podemos ver un ejemplo de cómo se puede aplicar el Teorema de los Cuatro Colores sobre las provincias de España.

El *Teorema de los Cuatro Colores* fue el primer gran teorema demostrado utilizando un ordenador, presentando una demostración que no pudo ser verificada directamente por otros matemáticos. Con el paso del tiempo, la demostración ha sido aceptada por la mayoría, proporcionando incluso refinamientos en cuanto al número de configuraciones (hasta 633), aunque siempre queda quien se hace preguntas del tipo *¿Cómo podemos garantizar la fiabilidad de los algoritmos y del hardware?*.

1.1. Descripción del problema

Ahora que ya hemos introducido brevemente el *Problema de Coloración de Grafos*, podemos plantearlo desde un punto de vista más riguroso.

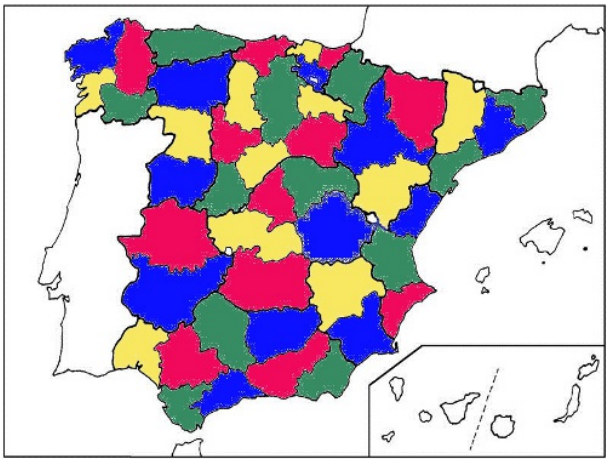


Figura 1.4: Coloración de las provincias de España.

Definición 1.1.1. Un *grafo* G es un par ordenado $G = (V, E)$, donde V es un conjunto de vértices y E es un conjunto de aristas. Un *subgrafo* de G es un grafo $G' = (V', E')$ donde $V' \subseteq V$ y $E' \subseteq E$. Un *grafo conexo* es un grafo tal que, para cada par de vértices del grafo, existe al menos una arista que los une, esto es, $\forall u, v \in V, \{u, v\} \in E$.

Sea $G = (V, E)$ un grafo, formado por un conjunto de n vértices V y un conjunto m de aristas E. Dado el grafo G, el problema de coloración de grafos busca asignar a cada vértice $v \in V$ un entero $c(v) \in \{1, 2, \ldots, k\}$ tal que:

- $c(v) \neq c(u)$ $\forall \{u, v\} \in E$; y
- k es mínimo.

Veamos ahora una serie de definiciones que pueden resultar de utilidad a la hora de describir una solución al problema de coloración de grafos y sus propiedades:

Definición 1.1.2. Una coloración se dice *factible* si y sólo si es *completa* y *apropiada*, es decir, que todos los vértices están asignados a algún color y que ningún par de vértices adyacentes (dados u, v $\in V$, $\{u, v\} \in E$) está asignado al mismo color.

Definición 1.1.3. El *número cromático* de un grafo G, $\chi(G)$, es el número mínimo de colores necesarios en una coloración factible de G. Una coloración factible de G usando exactamente $\chi(G)$ colores se considera *óptima*.

Definición 1.1.4. Un *clique* es un subconjunto de vértices $C \subseteq V$ adyacentes entre sí de forma que $\forall u, v \in C, \{u, v\} \in E$, es decir, que el subgrafo inducido por C es un grafo completo (presenta una arista entre cada par de vértices del grafo). Se denota $\omega(G)$ al número de vértices del mayor clique de G.

Definición 1.1.5. Dado un grafo $G = (V, E)$, una *matriz de adyacencia* es una matriz $\mathbf{A} \in \mathbb{R}_{n \times n}$ para la cual $A_{ij} = 1$ si y sólo si los vértices v_i y v_j son adyacentes (es decir, existe una arista que los une), y $A_{ij} = 0$ en cualquier otro caso.

Definición 1.1.6. La *vecindad* de un vértice v, $\Gamma_G(v)$, es el conjunto de vértices adyacentes a v en el grafo G, es decir, $\Gamma_G(v) = \{u \in V : \{v, u\} \in E\}$. El *grado* de un vértice v es la cardinalidad de su conjunto de vecinos, $|\Gamma_G(v)|$, que se suele denotar como $deg_G(v)$.

Resulta también útil para comprender el problema de coloración de grafos, plantearlo como un tipo de problema de particiones sobre el conjunto de vértices V, donde una solución S es representada por un conjunto de k clases de colores (o colores), $\mathcal{S} = \{S_1, \ldots, S_k\}$. Para que la solución sea factible, es necesario que se cumplan las siguientes restricciones a la vez que se minimiza el número de colores k:

$$\bigcup_{i=1}^{k} S_i = V, \tag{1.1}$$

$$S_i \cap S_j = \emptyset \quad (1 \leq i \neq j \leq k), \tag{1.2}$$

$$\forall u, v \in S_i, \{u, v\} \notin E \quad (1 \leq i \leq k). \tag{1.3}$$

Las restricciones (1.1) y (1.2) establecen que S debe ser una partición del conjunto de vértices V, mientras que la restricción (1.3) obliga a que ningún par de vértices adyacentes sean asignados a la misma clase (o color), es decir, que todas las clases de colores en la solución deben ser conjuntos independientes.

1.2. Dificultad del problema

La primera forma de afrontar el problema, la más intuitiva, podría ser comprobar una a una todas las posibles asignaciones entre colores y vértices para finalmente escoger la mejor de todas ellas. En primer lugar, deberíamos decidir el número máximo de colores que debemos utilizar. Suponiendo que nuestro grafo tiene n vértices, necesitaremos a lo sumo n colores. En ese caso, tendríamos un total de n^n soluciones posibles, número que crece de forma notable a medida que el valor de n aumenta, algo que ni el ordenador más potente podría resolver.

Sin embargo, la tasa de crecimiento exponencial del espacio de soluciones no es la única razón que hace del *Problema de Coloración de Grafos* un problema de dificultad extrema. El hecho de que problemas como este se consideren "complejos" o "intratables" se debe al trabajo realizado por el científico Stephen Cook, quien en el año 1971 introdujo los conceptos de "NP-completitud" y de "reducción en tiempo polinómico". Además, demostró que el problema conocido como el "problema de satisfacibilidad" es NP-completo. Dicho problema se plantea si, dada una expresión Booleana cualquiera, existe alguna asignación entre valores y variables tal que la expresión se evalúe como "verdadero".
Por ejemplo, la expresión "$a \wedge \neg b$" es satisfactoria, pues podemos fijar los valores $a = verdadero$, $b = falso$, haciendo que la expresión "$a \wedge \neg b$"$= verdadero$. Sin embargo, la expresión "$a \wedge \neg a$" resulta insatisfactoria.

En el ámbito de la complejidad computacional, los problemas suelen plantearse como "problemas de decisión", cuyas respuestas son "sí" o "no". Los problemas de coloración de grafos pueden plantearse también como problemas de decisión, viendo el problema de la forma ¿puede un grafo G ser coloreado de forma factible utilizando k colores?, en lugar de preguntarse cuál es el mínimo número de colores necesarios para obtener una coloración factible del grafo G.

El resultado demostrado por Cook sobre la NP-completitud del problema de satisfacibilidad, se puede utilizar para demostrar la NP-completitud de muchos otros problemas aplicando transformaciones polinómicas que, esencialmente, consisten en transformar de forma eficiente un problema de decisión en otro. El *Problema de Coloración de Grafos* generaliza el problema NP-completo de la 3-satisfacibilidad, es decir, que el problema de la 3-satisfacibilidad es polinómicamente reducible al problema de coloración de grafos. En la Figura 1.5, podemos ver un diagrama con las relaciones entre algunos de los problemas NP-completos, cuya prueba parte del trabajo de Cook sobre el problema de la satisfacibilidad.

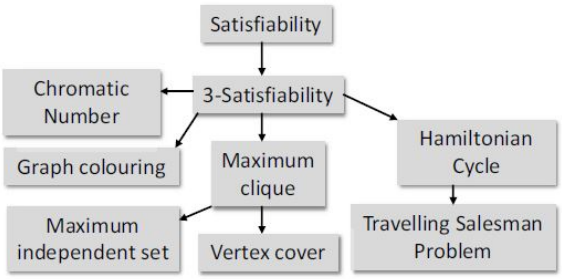


Figura 1.5: Problemas NP-completos

Como conclusión a este apartado, no podemos olvidar que la existencia de la clase NP-completo se basa en la conjetura de que P≠NP, uno de los problemas sin solución más famosos en el ámbito de las matemáticas.

1.3. Soluciones al Problema de Coloración de Grafos

A pesar de lo visto en la sección anterior, el hecho de que el problema de coloración de grafos sea NP-completo no significa que no haya casos para los cuales es posible encontrar una solución en tiempo polinómico. En este apartado veremos algunos de ellos.

1.3.1. Grafos 0-regulares

Los grafos 0-regulares son grafos donde no hay vértices adyacentes, es decir, se trata de grafos $G = (V, E)$, con $E = \emptyset$.
Basándonos en la estructura de dichos grafos, resulta trivial el hecho de que un grafo es 0-regular si y sólo si su número cromático es 1, es decir, pudiendo asignar todos los vértices al mismo color.

1.3.2. Grafos Completos

Los grafos completos con n vértices, denotados por K_n, son grafos que presentan una arista entre cada par de vértices del grafo, lo que proporciona un conjunto E de $m = \frac{n(n-1)}{2}$ aristas.
Puesto que todos los vértices del grafo son adyacentes entre sí, cada vértice debe ser asignado a un color distinto, por lo que el número cromático de un grafo completo es $\chi(K_n) = n$. En la Figura 1.6 podemos ver una coloración para el grafo completo de 5 vértices.

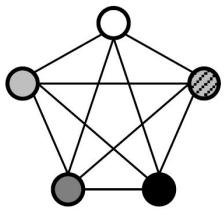
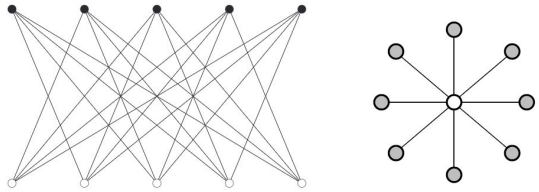


Figura 1.6: Coloración del grafo completo K_5

1.3.3. Grafos Bipartitos

Los grafos bipartitos, denotados por $G = (V_1, V_2, E)$, son grafos cuyos vértices pueden ser particionados en dos conjuntos V_1 y V_2 tales que sólo existen aristas entre vértices de V_1 y vértices de V_2. En consecuencia, V_1 y V_2 son conjuntos independientes, por lo que los grafos bipartitos pueden ser coloreados utilizando sólo 2 colores, asignando todos los vértices de V_1 a un color y todos los vértices de V_2 al otro. Resulta evidente que $\chi(G) = 2$ si y sólo si G es bipartito. En la Figura 1.7 podemos ver ejemplos de coloración de grafos bipartitos, un grafo bipartito aleatorio en 1.7a) y un grafo de estrella en 1.7b).

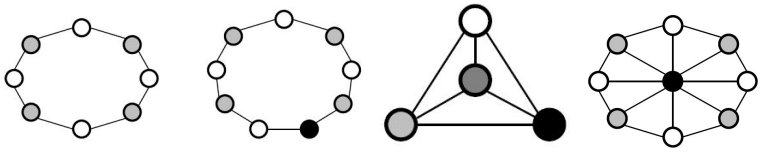


(a) Grafo bipartito aleatorio (b) Grafo de estrella

Figura 1.7: Coloración de grafos bipartitos

1.3.4. Grafos Planos

Los grafos planos son aquellos que pueden ser representados en el plano, sin que ninguna de las aristas del grafo corte a otra. Estos se ajusta a lo que hemos visto en cuanto al Teorema de los Cuatro Colores (si un grafo es plano entonces puede ser coloreado de forma factible utilizando 4 o menos colores), aunque el recíproco no es necesariamente cierto. En la Figura 1.8 podemos ver unos cuantos ejemplos de coloraciones de grafos planos. En 1.8a) vemos cómo dependiendo del número de vértices del grafo cíclico (si es par o impar), son necesarios 2 o 3 colores para obtener una coloración del grafo. En 1.8b) y 1.8c) vemos cómo obtener una coloración de los grafos de rueda, que se obtienen a partir de los cíclicos añadiendo un nuevo vértice en el centro, con aristas conectando el vértice central a todos los vértices exteriores. En consecuencia, para obtener una coloración factible de dichos grafos, serán necesarios 3 o 4 colores, dependiendo de si el número de vértices del grafo es par o impar.



(a) Coloración de C_8 y C_9 (b) Coloración W_4 (c) Coloración W_9

Figura 1.8: Coloración de grafos planos

Capítulo 2

Algoritmos más importantes

Debido a la "'intratabilidad" del problema de coloración de grafos, no existe un algoritmo concreto que permita obtener en tiempo polinómico la solución óptima a la coloración de cualquier grafo. Así como es posible identificar si un grafo presenta número cromático 1 (grafos 0-regulares) o 2 (grafos bipartitos), poder determinar si un grafo arbitrario presenta número cromático 3 es un problema NP-completo.

Es por ello que, para obtener soluciones del problema de coloración de grafos, se dispone de varias alternativas:

En primer lugar, podríamos utilizar un método de resolución exacta. Al tratarse de un problema NP-completo, estas técnicas no son viables en la mayoría de los casos, pues resultan bastante costosas en términos de tiempo computacional. Sólo en la coloración de los grafos más sencillos se podría obtener una solución óptima en un tiempo computacional razonable.

Con el objetivo de reducir dicho tiempo computacional, podríamos hacer uso de un algoritmo heurístico que permita obtener soluciones "de buena calidad" en tiempos computacionales aceptables. Esta opción es la más utilizada en la práctica, a pesar de que los algoritmos heurísticos no garantizan que se vaya a obtener la solución óptima del problema, incluso podría suceder que no se obtenga ninguna solución.

A continuación, describimos las técnicas de resolución exacta y los algoritmos heurísticos más conocidos para obtener soluciones al problema de coloración de grafos.

2.1. Algoritmos Voraz, DSatur y RLF

En este apartado vamos a analizar los algoritmos heurísticos Voraz, DSatur y RLF, tres algoritmos constructivos diseñados específicamente para la resolución del problema de coloración de grafos. Cualquiera de estos algoritmos proporciona una solución factible al problema, pudiendo ser esta óptima o no. Además de describir cada uno de los algoritmos detalladamente, también comentaremos las principales propiedades de cada uno de ellos.

2.1.1. El Algoritmo Voraz

Descripción del algoritmo

Se trata de uno de los algoritmos más simples y al mismo tiempo de uno de los algoritmos heurísticos más importantes en coloración de grafos. El algoritmo opera tomando los vértices del grafo uno a uno, siguiendo un orden (que puede ser aleatorio), y asignando a cada vértice el primer color disponible.
Al tratarse de un algoritmo heurístico, tal y como hemos comentado previamente, la solución proporcionada por este algoritmo puede no ser óptima. Sin embargo, una correcta elección del orden de los vértices para su coloración puede producir una solución óptima para cualquier grafo.
En la práctica, el algoritmo Voraz produce soluciones factibles de forma rápida, aunque estas soluciones pueden resultar "pobres" en base al número de colores que requiere el algoritmo, comparado con el número cromático del grafo. En la Figura 2.1 podemos ver el pseudocódigo asociado al algoritmo Voraz.

```
    GREEDY (S ← ∅, π)
(1) for i ← 1 to |π| do
(2)     for j ← 1 to |S|
(3)         if (S_j ∪ {π_i}) is an independent set then
(4)             S_j ← S_j ∪ {π_i}
(5)             break
(6)         else j ← j + 1
(7)     if j > |S| then
(8)         S_j ← {π_i}
(9)         S ← S ∪ S_j
```

Figura 2.1: Pseudocódigo algoritmo voraz

Partimos de la solución vacía $\mathcal{S} = \emptyset$ y de una permutación aleatoria de los vértices π. En cada iteración, el algoritmo selecciona el vértice i-ésimo en la permutación, π_i, y trata de encontrar una clase de color $S_j \in S$ en la cual pueda ser incluido dicho vértice. Si es posible, se añade dicho vértice a la clase de color correspondiente y el algoritmo pasa a considerar el siguiente vértice π_{i+1}. En caso contrario (línas 8 y 9 del pseudocódigo), se crea una nueva clase de color para el vértice considerado.

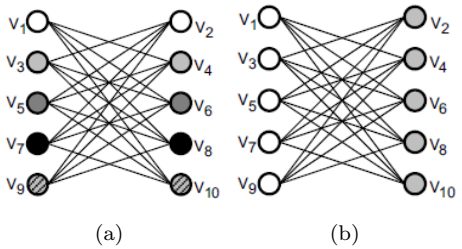


Figura 2.2: Importancia de la ordenación de vértices

En la Figura 2.2 presentamos un ejemplo de cómo afecta la ordenación de los vértices a la hora de obtener una solución óptima para la coloración de un grafo. Para el caso 2.2a), vemos que la permutación de los vértices $\pi = (v_1, v_2, v_3, \ldots, v_n)$ produce una solución utilizando $n/2$ colores (en este ejemplo $n = 10$, por lo que utiliza 5 colores), mientras que en el caso 2.2b), se utiliza la permutación $\pi = (v_1, v_3, \ldots, v_{n-1}, v_2, v_4, \ldots, v_n)$, produciendo así la solución óptima al problema, utilizando únicamente 2 colores.

Propiedades del algoritmo

En el peor de los casos, (por ejemplo, a la hora de establecer una coloración para el grafo K_n), se deberá realizar una comprobación de $0 + 1 + 2 + \cdots + (n-1)$ restricciones, lo que proporciona al algoritmo Voraz una complejidad computacional del orden $O(n^2)$. El Teorema 2.1.1 resulta de gran utilidad para demostrar cómo a partir de una solución con una determinada ordenación de los vértices, se pueden obtener otras mejorando la inicial.

Teorema 2.1.1. *Sea $\mathcal{S}$ una coloración factible de un grafo G. Si cada clase de color $S_i \in \mathcal{S}$ (para $1 \leq i \leq |\mathcal{S}|$) se considera en su momento, y todos los vértices se introducen de uno en uno en el algoritmo Voraz, la solución resultante $\mathcal{S}'$ será también factible, con $|\mathcal{S}'| \leq |\mathcal{S}|$.*

A continuación, veamos un ejemplo donde la desigualdad es estricta, es decir, que dada una solución factible, si modificamos el orden de las clases S_i y el orden de los vértices, en algunos casos es posible mejorar la solución de partida.

En la Figura 2.3, ambas coloraciones se han obtenido aplicando el algoritmo Voraz sobre el grafo. En la Figura 2.3(a), se utiliza la permutación de los vértices $\pi = (v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8)$, obteniendo así la 4-coloración del grafo $\mathcal{S} = \{\{v_1, v_4, v_8\}, \{v_2, v_7\}, \{v_3, v_5\}, \{v_6\}\}$. Apoyándonos en esta solución, se forma una nueva permutación de los vértices $\pi = (v_1, v_4, v_8, v_2, v_7, v_3, v_5, v_6)$. Sin embargo, el uso de conjuntos en la definición de una solución $\mathcal{S}$ significa que tenemos libertad para utilizar cualquier orden de las clases de color en $\mathcal{S}$ para formar π, de hecho, podemos utilizar cualquier orden de los vértices dentro de cada clase de color.

En consecuencia, podemos obtener una permutación alternativa de los vértices a partir de la solución $\mathcal{S}$, $\pi = (v_2, v_7, v_5, v_3, v_6, v_4, v_8, v_1)$. En la Figura 2.3(b) podemos ver el resultado de aplicar el algoritmo Voraz sobre esta última permutación, obteniendo así una 3-coloración del grafo y utilizando, por lo tanto, menos colores que en el primer caso. Estos conceptos dan lugar al siguiente teorema.

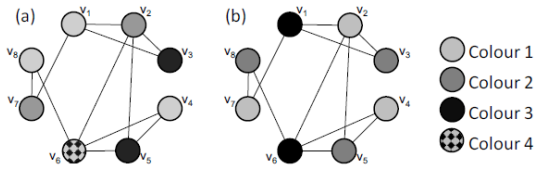


Figura 2.3: Coloración de un grafo variando el número de colores.

Teorema 2.1.2. *Sea G un grafo con una solución óptima $\mathcal{S} = \{S_1, \ldots, S_k\}$ al problema de coloración de grafos, donde $k = \chi(G)$. Entonces, existen al menos*

$$\chi(G)! \prod_{i=1}^{\chi(G)} |S_i|! \tag{2.1}$$

permutaciones de los vértices donde, al aplicar el algoritmo Voraz sobre ellas, se obtendrá una solución óptima del problema.

Demostración. Se deduce inmediatamente del Teorema 2.1.1: puesto que $\mathcal{S}$ es óptima, se puede generar una permutación adecuada de la forma descrita. Además, como las clases de color y los vértices sin clase de color asociada se pueden permutar, la fórmula indicada en la Ecuación (2.1) resulta consistente. □

Finalmente, veamos otra propiedad que se demuestra haciendo uso del algoritmo Voraz, y que permite acotar superiormente el número cromático. Para ello, nos apoyaremos en el grado de los

vértices de un grafo, es decir, el número de aristas incidentes al vértice. Enunciamos dicha propiedad en el siguiente teorema.

Teorema 2.1.3. *Sea G un grafo conexo con grado máximo $\Delta(G)$, con $\Delta(G) = max\{deg(v) : v \in V\}$, y $deg(v)$ el grado del vértice v. Entonces $\chi(G) \leq \Delta(G) + 1$.*

Demostración. Consideremos el comportamiento del algoritmo Voraz. Aquí, el vértice i-ésimo en la permutación π, denotado por π_i, será asignado a la clase de color con menor índice que no contenga a ninguno de sus vértices vecinos. Como cada vértice tiene un máximo de $\Delta(G)$ vecinos, como máximo serán necesarios $\Delta(G) + 1$ colores para colorear de forma factible todos los vértices de G. □

2.1.2. El Algoritmo DSatur

Descripción del algoritmo

El algoritmo DSatur (abreviatura de "Degree Saturation"), propuesto por Brélaz (1979), se comporta de forma muy similar al algoritmo Voraz, con la salvedad de que, en este caso, la ordenación de los vértices es generada por el propio algoritmo. Así como en el algoritmo voraz la ordenación se decidía antes de que ningún vértice fuera coloreado, en el algoritmo DSatur, el orden de los vértices se decide de forma heurística en base a las características del coloreado parcial del grafo en el momento en el que se selecciona cada uno de los vértices. Veamos ahora un concepto clave a la hora de plantear el orden de los vértices del grafo, el *grado de saturación* de los vértices.

Definición 2.1.4. Sea $c(v)$ =NULL para cualquier vértice $v \in V$ que todavía no haya sido asignado a ninguna clase de color. Dado entonces el vértice v, el *grado de saturación* de v, sat(v), es el número de colores diferentes asignados a los vértices adyacentes, es decir, sat$(v) = |\{c(u) : u \in \Gamma(v) \wedge c(u) \neq \text{NULL}\}|$.

```
    DSATUR (S ← ∅, X ← V)
(1) while X ≠ ∅ do
(2)     Choose v ∈ X
(3)     for j ← 1 to |S|
(4)         if (S_j ∪ {v}) is an independent set then
(5)             S_j ← S_j ∪ {v}
(6)             break
(7)         else j ← j + 1
(8)     if j > |S| then
(9)         S_j ← {v}
(10)        S ← S ∪ S_j
(11)    X ← X − {v}
```

Figura 2.4: Pseudocódigo algoritmo DSatur

En la Figura 2.4 podemos ver el pseudocódigo asociado al algoritmo DSatur. Aquí, se hace uso de un conjunto, X, para definir el conjunto de vértices que todavía no han sido asignados a un color. Evidentemente, al inicio de la ejecución, $X = V$. La parte más importante de este algoritmo se encuentra en el paso (2), donde el siguiente vértice en ser coloreado será aquel vértice en X que presente el máximo grado de saturación, y en caso de exista más de uno, aquel con mayor grado (ver Definición 1.1.6). Si sigue habiendo más de un vértice en estas condiciones, se escoge uno de ellos aleatoriamente.

La idea de este algoritmo es priorizar la coloración de aquellos vértices que tienen menos opciones de ser coloreados con un color ya existente (los que presentan un mayor número de restricciones). Una vez que un vértice es coloreado, se elimina del conjunto X y se vuelve a comenzar el algoritmo para un nuevo vértice.

Hemos visto que en el caso del algoritmo Voraz, dependiendo del orden en el que los vértices eran introducidos en el algoritmo, podía variar de forma considerable la optimalidad de nuestra solución. En este caso, la generación del orden como parte del algoritmo produce una reducción de esa variabilidad, haciendo que sea más predecible el comportamiento del algoritmo DSatur.

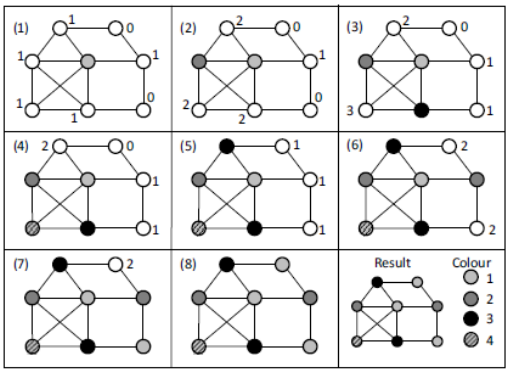


Figura 2.5: Coloración utilizando el algoritmo DSatur

En la Figura 2.5 podemos ver un ejemplo del funcionamiento de este algoritmo.

Propiedades del algoritmo

En cuanto a la complejidad computacional de este algoritmo, en el peor de los casos, nos encontramos en la misma situación que en el algoritmo voraz, con una complejidad del orden $O(n^2)$, aunque en la práctica se puede considerar que el hecho de realizar el seguimiento de la saturación de los vértices no coloreados produce un pequeño extra en cuanto a dicha complejidad.

Vemos ahora un par de situaciones en las cuales el algoritmo DSatur se comporta como un método exacto en lugar de heurístico.

Teorema 2.1.5. ***(Brélaz (1979))*** *El algoritmo DSatur es exacto para grafos bipartitos.*

La utilidad de este teorema se puede ver en la Figura 2.2, analizada previamente. En aquel caso, utilizábamos el algoritmo voraz y la permutación de los vértices para obtener una coloración del grafo, resultando en gran parte de las soluciones más de dos colores, llegando a utilizar $n/2$ colores en el peor de los casos. Sin embargo, tal y como demuestra este teorema, el algoritmo DSatur garantiza obtener la solución óptima para grafos bipartitos, así como también para otras topologías, como demuestra el siguiente teorema.

Teorema 2.1.6. *El algoritmo DSatur es exacto para ciclos y para grafos circulares.*

2.1.3. El Algoritmo RLF

Descripción del algoritmo

El algoritmo que analizamos a continuación sigue un planteamiento un poco diferente con respecto a los dos anteriores. El algoritmo RLF ("Recursive Largest First"), propuesto por Leighton (1979), funciona coloreando un grafo con un color por cada iteración del algoritmo, en vez de un vértice por iteración. En cada iteración, el algoritmo busca conjuntos de vértices independientes en el grafo, los cuales serán asociados a un mismo color. Dicho conjunto independiente será eliminado del grafo, y se procederá de la misma forma con el subgrafo restante, hasta que dicho subgrafo sea vacío, en cuyo

caso todos los vértices estarán asignados a algún color, produciendo así una solución factible.

```
    RLF (S ← ∅, X ← V, Y ← ∅, i ← 0)
(1) while X ≠ ∅ do
(2)     i ← i + 1
(3)     S_i ← ∅
(4)     while X ≠ ∅ do
(5)         Choose v ∈ X
(6)         S_i ← S_i ∪ {v}
(7)         Y ← Y ∪ Γ_X(v)
(8)         X ← X − (Y ∪ {v})
(9)     S ← S ∪ {S_i}
(10)    X ← Y
(11)    Y ← ∅
```

Figura 2.6: Pseudocódigo algoritmo RLF

En la Figura 2.6 podemos ver el pseudocódigo asociado al algoritmo RLF.
En este caso, el algoritmo hace uso de dos conjuntos, X, que contiene a los vértices que no han sido coloreados y que se pueden añadir a la clase S_i sin provocar ningún conflicto (es decir, una situación en la que, dados $u, v \in V$ tales que $\{u, v\} \in E$, se tiene que $c(u) = c(v)$), e Y, que contiene a los vértices que no han sido coloreados y que no pueden ser añadidos de forma factible por S_i (es decir, que no pueden ser coloreados por el color i). Evidentemente, al inicio del algoritmo, $X = V$ e $Y = \emptyset$. Una vez que X e Y estén vacíos, todos los vértices habrán sido coloreados. Entre los pasos (4) y (8) del algoritmo, se selecciona un vértice $v \in X$ y se añade al conjunto S_i (se colorea con el color i). A continuación se mueven al conjunto Y todos los vértices vecinos a v en el subgrafo inducido por X, pues estos no pueden ser coloreados con el color i. Se colorea al resto de elementos de X con el color i, y, por último, se vuelve a mover a todos los elementos de Y a X para volver a comenzar con el primer paso del algoritmo y continuar con la clase de color S_{i+1} en caso de que fuera necesario.

De la misma forma que en el algoritmo DSatur, se da prioridad a aquellos vértices que tienen mayor número de "restricciones", esto es, aquellos vértices que tengan mayor grado.
Para ver de forma clara el funcionamiento de este algoritmo, en la Figura 2.7 presentamos un ejemplo de un grafo plano coloreado mediante el algoritmo RLF.

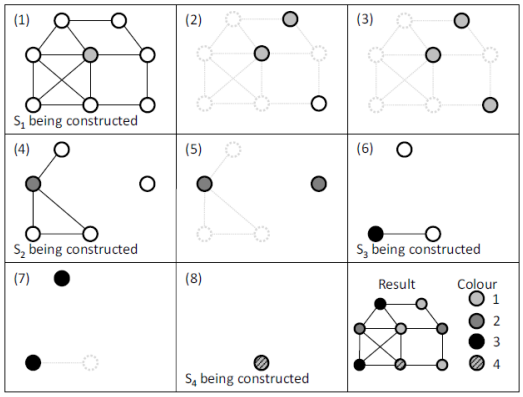


Figura 2.7: Coloración utilizando el algoritmo RLF

Propiedades del algoritmo

Se ha demostrado que la complejidad computacional de este algoritmo es del orden de $O(n^3)$, lo que, para grafos de gran tamaño, provoca un mayor esfuerzo computacional que en los dos algoritmos anteriores.

De la misma forma que sucede con el algoritmo DSatur, el algoritmo RLF también se comporta como un método exacto para algunas topologías. Los siguientes teoremas ratifican dicha afirmación.

Teorema 2.1.7. *El algoritmo RLF es exacto para grafos bipartitos.*

Teorema 2.1.8. *El algoritmo RLF es exacto para grafos cíclicos y para grafos circulares.*

2.1.4. Comparación empírica

Finalmente, en el Cuadro 2.1 podemos ver un resumen de los resultados obtenidos por Lewis (2015), al aplicar los algoritmos Voraz, DSatur y RLF sobre un conjunto de 50 grafos aleatorios para distintos números de vértices, n. Para ello, definimos en primer lugar el concepto de *grafo aleatorio*.

Definición 2.1.9. Un *grafo aleatorio*, denotado como $G_{n,p}$, es un grafo comprendiendo n vértices en donde cada par de vértices es adyacente con probabilidad p. En consecuencia, los grados de los vértices en un grafo aleatorio siguen una distribución binomial, es decir, $deg(v) \sim \mathrm{B}(n-1, p)$.

n	Voraz	DSatur	RLF
100	21.14 ± 0.95	18.48 ± 0.81	17.44 ± 0.61
500	72.54 ± 1.33	65.18 ± 1.06	61.04 ± 0.78
1000	126.64 ± 1.21	115.44 ± 1.23	108.74 ± 0.90
1500	176.20 ± 1.58	162.46 ± 1.42	153.44 ± 0.86
2000	224.18 ± 1.90	208.18 ± 1.02	196.88 ± 1.10

Cuadro 2.1: Comparación de los algoritmos Voraz, DSatur y RLF

A la vista de los resultados mostrados en el Cuadro 2.1, podemos confirmar que es el algoritmo RLF el que produce soluciones de mayor calidad (en el cuadro se muestra el número de colores en $\mu \pm \sigma$).

2.2. Métodos Exactos

Los algoritmos exactos son aquellos que siempre determinan la solución óptima de un problema computacional (pudiendo necesitar un exceso de tiempo en obtenerla). Es posible pues, diseñar algoritmos exactos que resultan significativamente más rápidos que la búsqueda exhaustiva, aunque, en este caso, no es posible aplicar estos algoritmos en tiempo polinómico. Analicemos dos de los métodos más conocidos.

2.2.1. Algoritmos de retroceso

El retroceso es un método general que puede utilizarse para determinar la solución óptima (todas ellas, en caso de ser más de una), a un problema computacional como puede ser el problema de coloración de grafos. Estos algoritmos obtienen la solución de forma sistemática, transformando soluciones parciales en soluciones completas.

Durante el proceso de construcción, dado que no es posible obtener una solución óptima a partir de la solución parcial, el algoritmo retrocede para intentar encontrar el camino adecuado para ajustar la solución parcial actual. En lo referente al problema de coloración de grafos, actuaría de la siguiente forma.

Dado un grafo $G = (V, E)$, en primer lugar se ordenan los vértices de forma que v_i $(1 \leq i \leq n)$ se corresponde con el vértice i-ésimo en dicho ordenación. Además, se debe seleccionar un valor de k denotando el número de colores disponibles. Inicialmente, $k = \infty$.

En ese momento, el algoritmo realiza una serie de pasos hacia delante y hacia atrás. En los pasos hacia delante, el algoritmo colorea los vértices en el orden indicado hasta que identifica un vértice que no puede ser coloreado por ninguno de los k colores disponibles. Por otra parte, en los pasos hacia atrás, el algoritmo retrocede sobre los vértices coloreados y trata de identificar en qué puntos es posible asignar un color diferente a los vértices coloreados previamente. A partir de esos puntos, se vuelve a realizar una serie de pasos hacia delante. Si se encuentra alguna coloración factible, entonces se puede fijar k como el número de colores utilizado en dicha coloración menos 1, continuando con la ejecución del algoritmo. Finalmente, el algoritmo termina cuando un paso hacia atrás alcanza el vértice raíz v_1, o cuando se llega a algún otro criterio de parada (por ejemplo, límite de tiempo máximo).

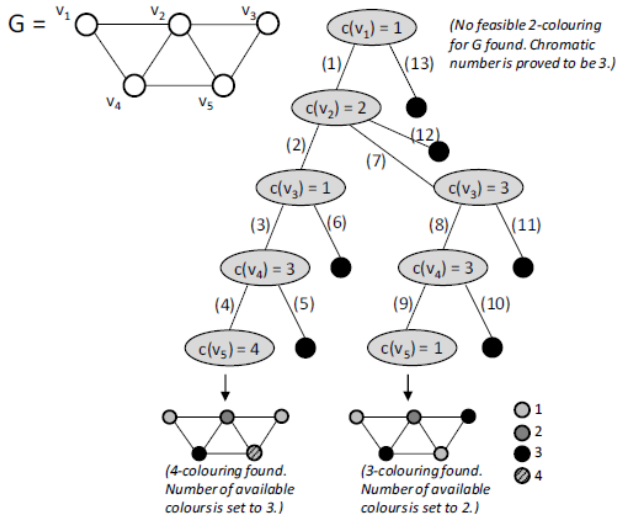


Figura 2.8: Coloración mediante el algoritmo de retroceso

En la Figura 2.8 podemos ver un ejemplo de cómo aplicar el algoritmo de retroceso para mejorar la solución a un problema de coloración de grafos. Así, en los pasos (1) a (4) se realiza una primera coloración mediante el algoritmo voraz, obteniendo una 4-coloración del grafo y fijando así el número de colores disponibles a 3. Retrocediendo en los vértices, no es posible definir ningún color alternativo hasta el v_3. En los pasos (8) y (9) se lleva a cabo la nueva coloración de los vértices, obteniendo así una coloración con 3 colores, mejor que la obtenida inicialmente. En este momento, fijando en 2 el número

de colores disponibles, al retroceder sobre los vértices, alcanzamos el vértice v_1 sin poder modificar la coloración de ninguno de los vértices, de forma que el algoritmo finalizaría con la 3-coloración obtenida. En consecuencia, el número cromático de este grafo es 3.

2.2.2. Modelos de programación entera

Otra forma de obtener un algoritmo exacto para la coloración de grafos es utilizar la programación entera (Integer Programming, IP), que no es más que un tipo de programación lineal en el que algunas de las variables de decisión están restringidas a tomar valores enteros, y cuya función objetivo es minimizar el número de colores utilizados. Veamos cómo podemos formular el problema de coloración de grafos como un problema de programación entera.

Sea $G = (V, E)$ un grafo con n vértices y m aristas. Probablemente, la formulación más simple se obtiene utilizando dos matrices binarias, $X \in \mathbb{R}_{n\times n}$ e $Y \in \mathbb{R}_{n\times 1}$, conteniendo las variables del problema. Dichas matrices, cuyos elementos serán binarios, se interpretan de la siguiente forma:

$$X_{ij} = \begin{cases} 1 & \text{si el vértice } v_i \text{ se asigna al color } j \\ 0 & \text{en otro caso} \end{cases} \tag{2.2}$$

$$Y_j = \begin{cases} 1 & \text{si al menos un vértice es asignado al color } j \\ 0 & \text{en otro caso} \end{cases} \tag{2.3}$$

Como hemos comentado, el objetivo de este modelo es minimizar el número de colores a utilizar en base a la función objetivo:

$$\min \sum_{j=1}^{n} Y_j \tag{2.4}$$

sujeto a las siguientes restricciones:

$$X_{ij} + X_{lj} \leq Y_j \ \forall \{v_i, v_l\} \in E, \ \forall j \in \{1, \ldots, n\} \tag{2.5}$$

$$\sum_{j=1}^{n} X_{ij} = 1 \ \forall v_i \in V, \tag{2.6}$$

donde la restricción (2.5) garantiza que ningún par de vértices adyacentes es asignado al mismo color y que $Y_j = 1$ si y sólo si algún vértice es asignado al color j. Por su parte, la restricción (2.6) especifica que cada vértice debe ser asignado exactamente a un único color.

2.3. Algoritmos basados en técnicas metaheurísticas

Además de los algoritmos heurísticos constructivos que hemos visto en la primera sección de este capítulo, (Voraz, DSatur y RLF), los cuales están definidos específicamente para resolver este tipo de problemas, se pueden construir otros algoritmos heurísticos combinando cualquiera de ellos con técnicas heurísticas para resolver problemas de optimización en general, como pueden ser los métodos TS, de "Tabu Search" (Búsqueda Tabú), SA, de "Simulated Annealing" (Recocido/Templado Simulado) o ACO, de "Ant Colony Optimization" (Optimización por Colonia de Hormigas), entre otros. La elección

de una u otra técnica, basadas cada una de ellas en una filosofía diferente, dependerá del problema a tratar así como de los estudios experimentales que se realicen.

En esta sección nos centraremos en el estudio de tres algoritmos basados en técnicas metaheurísticas, que son el algoritmo TabuCol, el PartialCol y el HEA. Como veremos más adelante, estos algoritmos son los que producen mejores resultados en cuanto a calidad de las soluciones y tiempo de ejecución para grafos aleatorios. Eso provoca que, salvo algunas excepciones muy concretas, sean los algoritmos basados en técnicas metaheurísticas más utilizados para resolver problemas relacionados con la coloración de grafos.

Búsqueda Tabú

La Búsqueda Tabú, es una técnica metaheurística propuesta por Fred Glover (1986). El método se centra en la búsqueda de óptimos locales, mediante la prohibición de ciertos movimientos. El objetivo de clasificar algunos movimientos como prohibidos ("tabú"), es evitar que el algoritmo entre en bucles cíclicos. Estos movimientos pierden su estatus de tabú para volver a ser accesibles después de un período relativamente corto de tiempo.

Desde el punto de vista de la Inteligencia Artificial, la Búsqueda Tabú se desvía en gran medida de lo que cabría esperar acerca del comportamiento inteligente del ser humano. Frecuentemente, el ser humano tiende a actuar en base a algún criterio aleatorio (o probabilístico) que promueve un cierto nivel de inconsistencia. Sin embargo, la Búsqueda Tabú opera sin tener en cuenta la aleatorización.

La Búsqueda Tabú actúa de acuerdo a la suposición de que no tiene valor escoger un movimiento pobre, accidentalmente o por el diseño del problema, salvo cuando el propósito es evitar un camino que ya ha sido examinado previamente, por lo que el método busca en cada paso el mejor movimiento posible.

En base a estas consideraciones, el procedimiento inicial apunta directamente a la obtención de un óptimo local. Sin embargo, esta no es una condición que conduzca a la interrupción del proceso de búsqueda, ya que la posibilidad de encontrar un movimiento disponible que mejore la solución, sigue estando ahí.

Para evitar volver a recorrer un camino que ya se ha tomado previamente, el método registra información sobre los últimos movimientos realizados, utilizando una o varias *listas tabú*. Su objetivo es conseguir que no se repitan ciertos movimientos y evitar así el ciclado del algoritmo.

Las *listas tabú* se irían actualizando a medida que avanza el algoritmo, siguiendo habitualmente una disciplina FIFO (First In First Out) con un tamaño fijo, de forma que un movimiento que en un momento dado sea tabú podría dejar de serlo después de una serie de iteraciones.

Finalmente, analizamos el pseudocódigo asociado a este método, propuesto por el propio Glover (1989), para entender mejor su funcionamiento. Para ello planteemos, en primer lugar, un problema de optimización combinatorio como sigue:

$$\text{Minimizar } f(x) : x \in X \text{ en } R_n,$$

donde R_n es un conjunto de n restricciones que deben satisfacer todos los elementos que estén en el conjunto X.

Nos interesa, por tanto, minimizar la función objetivo, denotada por $f(x)$[1], donde x es un elemento del conjunto X, verificando una serie de restricciones, R_n.

Una vez definidos estos conceptos, veamos el pseudocódigo asociado al método de la Búsqueda Tabú.

(1) Seleccionamos un elemento inicial $x \in X$ y fijamos $x^* \leftarrow x$. Fijamos el contador de iteraciones $k = 0$ y la lista tabú $T = \emptyset$.

(2) Si $S(x) \setminus T = \emptyset$, pasar al Paso (4). En otro caso, fijamos $k \leftarrow k+1$ y seleccionamos $s_k \in S(x) \setminus T$ tal que $s_k(x) = \textbf{ÓPTIMO}(s(x) : s \in S(x) \setminus T)$.

(3) Sea $x \leftarrow s_k(x)$. Si $f(x) < f(x^*)$, donde x^* denota la mejor solución encontrada hasta el momento, entonces $x^* \leftarrow x$.

(4) Si ha transcurrido un número determinado de iteraciones, bien sea en total o desde que x^* fue mejorado por última vez, o si $S(x) \setminus T = \emptyset$ en caso de llegar a este paso directamente desde el Paso (2), finalizar la ejecución. En otro caso, actualizar T como se ha definido previamente y volver a ejecutar desde el Paso (2).

Interpretemos ahora el objetivo de cada uno de los pasos del pseudocódigo. En el Paso (1), x^* denotará la mejor solución encontrada hasta el momento, k el número de iteraciones del algoritmo y T el conjunto de movimientos tabú. En cada paso, a medida que el algoritmo avance, todos ellos se irán actualizando. En el Paso (2), teniendo en cuenta que $S(x)$ denota el conjunto de movimientos que se pueden aplicar a la solución x, es decir, que darían lugar a un elemento dentro del conjunto X, seleccionamos el movimiento (que se denotará por s_k) dentro de ese conjunto y fuera de la lista tabú que nos garantiza el mejor elemento posible, es decir, el elemento en el que se alcanza el valor mínimo de la función objetivo (que se denotará por $s_k(x)$). Finalmente, en el Paso (3), se compara la solución obtenida en el paso anterior con la mejor solución encontrada hasta el momento. Si la mejora, se actualiza la mejor solución.

Una elección natural de la función **ÓPTIMO** viene dada por la selección de $s_k(x)$ tal que:

$$f(s_k(x)) = \text{Min}\{f(s(x)) : s \in S(x) \setminus T\}.$$

Aunque hemos considerado que la lista tabú está formada por movimientos que no queremos permitir, ya que va a ser el procedimiento que se utilice en los algoritmos TabuCol y PartialCol, la lista tabú podría estar formada por soluciones o por atributos de una solución que no queremos que se repitan.

En muchos de los algoritmos en los que se hace uso de la técnica tabú, es habitual tener en cuenta también lo que se conoce como criterio de aspiración. De acuerdo a este criterio, un movimiento que esté en la lista tabú podría ser aceptado si cumple una serie de criterios que serán medidos por la función de aspiración.

2.3.1. El algoritmo TabuCol

Este algoritmo, propuesto inicialmente por Hertz y de Werra (1987), utiliza un operador de búsqueda local para obtener una coloración factible del grafo. El algoritmo TabuCol opera en el espacio de

[1] si el objetivo del problema fuera el de maximizar la función objetivo $f(x)$, bastaría con resolver el problema asociado a minimizar la función objetivo $(-f(x))$.

las k-coloraciones completas inapropiadas (que contienen algún conflicto, es decir, algún par de vértices adyacentes asignados al mismo color), utilizando una función objetivo que cuenta el número de conflictos que tienen lugar en el grafo, por lo que podemos expresar dicha función objetivo como:

$$f(S) = \sum_{\forall\{u,v\}\in E} g(u,v), \tag{2.7}$$

donde

$$g(u,v) = \begin{cases} 1 & \text{si } f(u) = f(v) \\ 0 & \text{en otro caso} \end{cases}$$

El objetivo de los algoritmos que, de la misma forma que el TabuCol, se mueven en este espacio de soluciones, es realizar modificaciones sobre la k-partición del conjunto de vértices, de forma que el número de conflictos se reduzca a cero.

Dada una solución candidata $\mathcal{S} = \{S_1, \ldots, S_k\}$, el algoritmo TabuCol se apoya en el método de la Búsqueda Tabú, de forma que los movimientos en el espacio de soluciones se ejecutan seleccionando un vértice $v \in S_i$ cuya asignación a la clase S_i provoca un conflicto, asignándolo en ese caso a una nueva clase de color $S_j \neq S_i$. Esto significa que volver a mover v a la clase S_i es *tabú* (es decir, está prohibido). Además, el algoritmo TabuCol utiliza un *criterio de aspiración* para permitir algún movimiento tabú de forma ocasional.

Tal y como sucede en los algoritmos basados en el método de la Búsqueda Tabú, en cada iteración del algoritmo TabuCol, se considera todo el conjunto de soluciones vecinas. La solución candidata inicial se construye tomando un orden aleatorio sobre los vértices y aplicando una versión adaptada del algoritmo voraz en la que sólo se permite utilizar k colores.

2.3.2. El algoritmo PartialCol

De forma similar al algoritmo anterior, el algoritmo PartialCol, propuesto por Blöchliger y Zufferey (2008), utiliza el método de la Búsqueda Tabú para encontrar una k-coloración apropiada. Sin embargo, en el algoritmo PartialCol no se considera el espacio de las soluciones inapropiadas, sino que, en su lugar, los vértices que no pueden ser asignados a alguno de los k colores sin provocar un conflicto, se mueven al conjunto U, formado por todos aquellos vértices que no han sido coloreados.

Una solución inicial para este algoritmo se genera, igual que en el caso anterior, a partir del algoritmo voraz, limitado a un máximo de k colores, con la única diferencia de que, cuando se encuentra un vértice para el cual no existe ningún color al que poder asignarlos sin provocar un conflicto, entonces ese vértice se mueve al conjunto U.

El objetivo de este algoritmo es realizar una serie de modificaciones sobre la solución parcial S, de forma que el conjunto U pueda quedar vacío, resultando así la función objetivo $f = |U| = 0$ y obteniendo una solución factible con k colores. Esto hace que el algoritmo se mueva por el espacio de soluciones de forma diferente a como lo hace en el algoritmo anterior. En este caso, un movimiento en el espacio de soluciones se ejecuta seleccionando un vértice $v \in U$, y asignando este a la clase de color $S_j \in S$. El movimiento se completa seleccionando todos los vértices $u \in S_j$ adyacentes a v y transfiriéndolos de S_j a U.

De la misma forma que en el TabuCol, en cada iteración, el PartialCol considera todo el conjunto de soluciones vecinas.

2.3.3. El algoritmo HEA

El Algoritmo Híbrido Evolutivo (HEA), propuesto por Galinier y Hao (1999), opera manteniendo un conjunto de soluciones candidatas evolucionadas a través de un operador de recombinación y un método de búsqueda local. De la misma forma que el algoritmo TabuCol, el algoritmo HEA opera en el espacio de las k-coloraciones completas e inapropiadas, utilizando la función de costes f definida en la Ecuación (2.7).

El algoritmo comienza creando un conjunto inicial de soluciones candidatas. Cada una de estas soluciones se crea utilizando una versión modificada del algoritmo DSatur, para el cual el número de colores k está fijado previamente. Para generar un conjunto de soluciones variado, el primer vértice se selecciona de forma aleatoria y se asigna al primer color. El resto de los vértices se seleccionan en base al máximo grado de saturación y se asignan a la clase de color con menor índice S_i factible para dicho vértice (donde $1 \leq i \leq k$). Si alguno de los vértices no puede ser asignado de forma factible a una clase de color existente, se mantienen al margen y al final del proceso se asignan a las clases de color de forma aleatoria. Una vez creado este conjunto inicial de soluciones, se trata de mejorar cada una de las soluciones aplicando el método de búsqueda local.

Como suele ser habitual en los algoritmos evolutivos, el algoritmo evoluciona las soluciones del conjunto inicial utilizando recombinación, mutación y presión evolutiva. En cada iteración, se seleccionan aleatoriamente dos "soluciones padre" S_1 y S_2 del conjunto inicial, y se utiliza una copia de estas soluciones, junto con el operador de recombinación, para generar una "solución hijo" $\mathcal{S}'$. Esta nueva solución es mejorada posteriormente a través del operador de búsqueda local y se introduce sobre el conjunto inicial, reemplazando así a la "solución padre" más débil. La presión evolutiva existe debido a que la "solución hijo" reemplaza a la "solución padre" más débil, independientemente de si el coste asociado a la "solución padre" es menor que el asociado a la "solución hijo".

Galinier y Hao (1999), utilizan el operador de recombinación conocido como GPX, de Greedy Partition Crossover (o "división codiciosa"), cuyo funcionamiento se basa en construir nuevas "soluciones hijo" utilizando las clases de color de mayor tamaño heredadas de ambas "soluciones padre".

	Parent $\mathcal{S}_1$	Parent $\mathcal{S}_2$	Offspring $\mathcal{S}'$	Comments
1)	$\{\{v_1,v_2,v_3\}, \{v_4,v_5,v_6,v_7\}, \{v_8,v_9,v_{10}\}\}$	$\{\{v_3,v_4,v_5,v_7\}, \{v_1,v_6,v_9\}, \{v_2,v_8,v_{10}\}\}$	$\{\}$	To start, the offspring solution $\mathcal{S} = \emptyset$.
2)	$\{\{v_1,v_2,v_3\}, \{v_8,v_9,v_{10}\}\}$	$\{\{v_3\}, \{v_1,v_9\}, \{v_2,v_8,v_{10}\}\}$	$\{\{v_4,v_5,v_6,v_7\}\}$	Select the colour class with most vertices and copy it into $\mathcal{S}'$. (Class $\{v_4,v_5,v_6,v_7\}$ from $\mathcal{S}_1$ in this case.) Delete the copied vertices from both $\mathcal{S}_1$ and $\mathcal{S}_2$.
3)	$\{\{v_1,v_3\}, \{v_9\}\}$	$\{\{v_3\}, \{v_1,v_9\}\}$	$\{\{v_4,v_5,v_6,v_7\}, \{v_2,v_8,v_{10}\}\}$	Select the largest colour class in $\mathcal{S}_2$ and copy it into $\mathcal{S}'$. Delete the copied vertices from both $\mathcal{S}_1$ and $\mathcal{S}_2$.
4)	$\{\{v_9\}\}$	$\{\{v_9\}\}$	$\{\{v_4,v_5,v_6,v_7\}, \{v_2,v_8,v_{10}\}, \{v_1,v_3\}\}$	Select the largest colour class in $\mathcal{S}_1$ and copy it into $\mathcal{S}'$. Delete the copied vertices from both $\mathcal{S}_1$ and $\mathcal{S}_2$.
5)	$\{\{v_9\}\}$	$\{\{v_9\}\}$	$\{\{v_4,v_5,v_6,v_7\}, \{v_2,v_8,v_{10},v_9\}, \{v_1,v_3\}\}$	Having formed k colour classes, assign any missing vertices to random colours to form a complete but not necessarily proper offspring solution $\mathcal{S}$.

Figura 2.9: Aplicación del GPX de Galinier y Hao (1999), con $k = 3$.

El funcionamiento de este operador se puede ver en la Figura 2.9. En este ejemplo, se considera

un conjunto formado por 10 vértices y se determina el número de clases de color $k = 3$. Se selecciona en primer lugar la clase de color de mayor tamaño dentro del conjunto de "soluciones padre" (que no tiene por qué ser apropiada), y se copia sobre el conjunto de descendencia. Con la intención de evitar vértices duplicados en el conjunto de descendencia en posteriores etapas, estos vértices que han sido copiados se eliminan de ambas "soluciones padre".

Para crear la siguiente clase de color, las "soluciones padre" resultantes se vuelven a considerar, seleccionando de nuevo aquella cuya clase de color sea mayor y copiándola de nuevo sobre el conjunto de descendencia, y eliminándola de ambas "soluciones padre". El proceso continúa hasta que el conjunto de descendencia esté formado por k clases de color. En ese momento, cada clase de color del conjunto de descendencia será un subconjunto de una clase de color existente en alguna de las "soluciones padre". Es decir:

$$\forall S_i \in \mathcal{S}' \; \exists S_j \in (S_1 \cup S_2) : S_i \subseteq S_j \tag{2.8}$$

donde $\mathcal{S}'$, $\mathcal{S}_1$ y $\mathcal{S}_2$ representan la "solución hijo" y las dos "soluciones padre", respectivamente. Sin embargo, puede que alguno de los vértices no aparezca en la "solución hijo". Para solucionarlo, se asignan a clases de color aleatoriamente aquellos vértices que no aparezcan en la "solución hijo".

Una vez que se ha construido una "solución hijo" completa, esta es modificada y mejorada mediante un procedimiento de búsqueda local antes de ser insertado en el conjunto de soluciones. Para ello, se utiliza el algoritmo TabuCol con un número fijo de iteraciones I, siguiendo el mismo criterio a la hora de considerar un elemento "tabú" que en el descrito en la Sección 2.3.1.

2.3.4. Comparativa entre los diferentes algoritmos

En los últimos años, se han propuesto numerosos métodos con el objetivo de explorar el espacio de las k-coloraciones completas e inapropiadas, incluyendo técnicas basadas en:

- Algoritmos evolucionarios (entre ellos el algoritmo HEA) (Dorne y Hao, 1998; Eiben et al., 1998; Fleurent y Ferland, 1996; Galinier y Hao, 1999);
- Iteración de búsqueda local (Chiarandini y Stützle, 2002; Paquete y Stützle, 2002);
- Algoritmos GRASP (Laguna y Marti, 2001);
- Búsqueda de vecinos variable (Avanthay et al., 2003);
- Optimización por colonia de hormigas (Thompson y Downsland, 2008).

Para finalizar este capítulo, veamos una comparativa entre los algoritmos que hemos analizado en las secciones previas, tras ser aplicados sobre grafos aleatorios $G_{n,0.5}$.

En el Cuadro 2.2 vemos el resultado obtenido por Lewis (2015), después de aplicar cada uno de estos algoritmos sobre 25 grafos, mostrando el número de colores utilizados en $\mu \pm \sigma$. Podemos concluir que para grafos de menor tamaño ($n = 250$), los algoritmos TabuCol y HEA son los que producen las mejores soluciones (no muy distantes de las obtenidas con el PartialCol), mientras que para grafos de mayor tamaño ($n = 1000$), es el HEA el que produce las soluciones con menor número de colores.

En el Cuadro 2.3, presentamos una comparación realizada por Lewis (2015) entre los algoritmos anteriores en términos de tiempo computacional. Se muestra el tiempo (en segundos) en completar ejecuciones de 5×10^{11} restricciones sobre grafos aleatorios $G_{n,0.5}$, utilizando un PC Windows 7, con procesador de 3.0 GHz y con 3.87 GB RAM. Podemos ver como los algoritmos TabuCol, PartialCol

n	**TabuCol**	**PartialCol**	**HEA**
250	28.04 ± 0.20	28.08 ± 0.28	28.04 ± 0.33
500	49.08 ± 0.28	49.24 ± 0.44	47.88 ± 0.51
1000	88.92 ± 0.40	89.08 ± 0.28	85.48 ± 0.46

Cuadro 2.2: Resultados obtenidos al alcanzar el límite computacional utilizando grafos aleatorios $G_{n,0.5}$.

n	250	500	1000
TabuCol	1346	1622	1250
PartialCol	1435	1372	1356
HEA	1469	1400	1337

Cuadro 2.3: Comparación en tiempos de CPU

y HEA tienen unos tiempos de ejecución similares. Aunque no se muestra en este trabajo, en Lewis (2015) se puede observar una comparación completa entre los tres algoritmos que hemos comentado junto con los algoritmos AntCol, Hill-Climbing y Backtracking DSatur, siendo los mostrados en el Cuadro 2.3 los más rápidos en ejecutarse.

Capítulo 3

Aplicaciones y casos de uso

Ahora que ya hemos estudiado con mayor detalle el *Problema de Coloración de Grafos*, vamos a analizar una serie de problemas para los cuales, afrontándolos a través de la coloración de grafos, se puede obtener soluciones a los mismos.

3.1. Cuadrados Latinos y Sudoku

El primero de los casos que vamos a analizar es el famoso puzzle conocido como *Sudoku*. Para ello, definiremos en primer lugar los *Cuadrados Latinos*. Se trata de rejillas de tamaño $\ell \times \ell$ que se rellenan con ℓ símbolos diferentes, apareciendo cada uno de ellos únicamente una vez por fila y una vez por columna.

Estos cuadrados latinos presentan algunas aplicaciones prácticas como, por ejemplo, dentro del ámbito del diseño experimental, en los centros de ensayos médicos, donde se desea probar el efecto de ℓ medicamentos como remedio para una enfermedad, teniendo lugar durante ℓ semanas y actuando sobre ℓ pacientes diferentes.

En este caso, podríamos utilizar un cuadrado latino de tamaño $\ell \times \ell$ para asignar los medicamentos, de forma que las filas representarían a los pacientes y las columnas, las semanas. Así, durante las ℓ semanas, cada paciente recibe cada uno de los ℓ medicamentos una única vez, y en cada semana, se comprueban los efectos de los ℓ medicamentos. En la Figura 3.1 podemos ver el planteamiento de un *Cuadrado Latino* como un *Problema de Coloración de Grafos*.

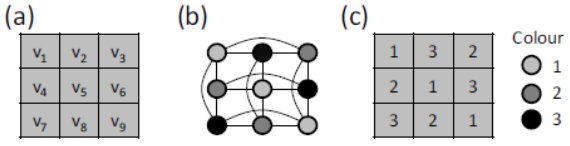


Figura 3.1: Cuadrados Latinos y Coloración de Grafos

Así, cada celda de la rejilla se asocia con un vértice, y las aristas se añaden entre aquellos pares de vértices que están en la misma fila y aquellos pares de vértices que están en la misma columna. Además, podemos observar que el conjunto de vértices de cada fila (o de cada columna), forma un clique de tamaño ℓ, de forma que no será posible obtener una solución utilizando menos de ℓ colores.

La *Coloración de Grafos* resulta más útil a la hora de resolver estos problemas cuando se consideran los *Cuadrados Latinos Parciales*. Este problema, perteneciente a los problemas del tipo NP-completo, consiste en partir de una rejilla de tamaño $\ell \times \ell$ parcialmente cubierta y decidir si es posible o no completarla, de forma que se obtenga un *Cuadrado Latino*.

En la Figura 3.2 podemos ver un ejemplo de cómo afrontar un problema de cuadrados latinos parcial utilizando el problema de coloración de grafos. Básicamente, sigue el mismo criterio que en el caso visto en la Figura 3.1, salvo que en este caso, en primer lugar, se genera una coloración del subgrafo que inducen los vértices disponibles en la rejilla del problema de partida, contrayendo aquellas aristas que unen vértices asignados al mismo color y finalmente coloreando el grafo asociado a la rejilla por completo. De esta forma, una ℓ-coloración se corresponde con un cuadrado latino completo de tamaño $\ell \times \ell$. Dependiendo de los valores presentes en la rejilla del problema de partida, puede haber cero, una o varias ℓ-coloraciones factibles.

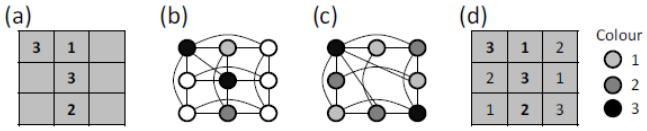


Figura 3.2: Cuadrados Latinos Parciales y Coloración de Grafos

En los *Sudoku*, partiendo de un cuadrado latino parcial, nuestro objetivo es completar las casillas restantes, de forma que cada columna y cada fila contengan los caracteres $1, \ldots, \ell$ una única vez. Además, la rejilla del *Sudoku* está dividida en ℓ "cajas", las cuales deben contener los caracteres $1, \ldots, \ell$ también una única vez.

Dicho esto, podemos considerar los *Sudoku* como un caso especial de cuadrados latinos parciales, con esta última restricción asociada a cada "caja", por lo que se pueden transformar en un *Problema de Coloración de Grafos* de la misma forma que hemos visto en la Figura 3.2, añadiendo las aristas necesarias para garantizar las restricciones asociadas a las "cajas" de la rejilla.

3.2. Distribución de Asientos

En este capítulo analizamos, desde el punto de vista de la coloración de grafos, la resolución a una situación que, la mayoría de las personas, en algún momento de su vida, deben afrontar, como es la planificación de un banquete.

Imaginemos un evento social, como podría ser un bautizo, comunión o boda, en cuyo banquete, N invitados deben ser distribuidos en k mesas. Es evidente que esperamos que los invitados disfruten del evento, por lo que será necesario diseñar un plan para que cada uno de ellos pueda, en la medida que sea posible, sentarse en la mesa junto a las personas que ellos prefieran. Para ello, deberemos tener en cuenta numerosos factores, como por ejemplo:

- Aquellos invitados que acuden en grupo, como puede ser el caso de parejas o familias, deberían sentarse en la misma mesa, preferiblemente unos al lado de los otros.
- Si algunos de los invitados no tuvieran buena relación, estos deberían sentarse en mesas diferentes. Del mismo modo, si algunos de ellos tuvieran muy buena relación, sería deseable que pudieran sentarse en la misma mesa.

- Algunos invitados puede que necesiten sentarse en alguna mesa concreta (como podría ser, cerca del lavabo, o cerca de una ventana), o también puede ser que algunos no deban sentarse en alguna otra (lejos de la barra libre, o lejos de la mesa presidencial, por ejemplo).

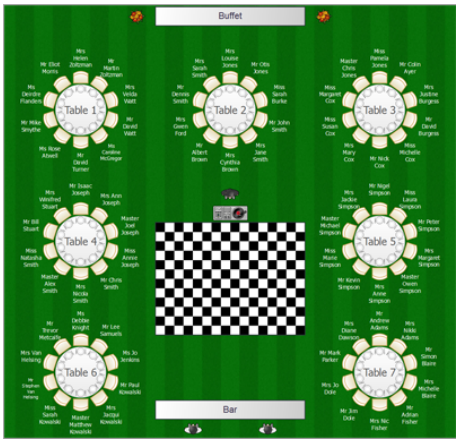


Figura 3.3: Ejemplo del problema de distribución de asientos.

Resulta evidente que, para valores grandes de N y k, optar por considerar todas las distribuciones posibles y seleccionar la más adecuada sería demasiado costoso en el tiempo como para resolver el problema. Puesto que este método no resulta factible, en la actualidad existen varias compañías que ofrecen sus servicios a través de páginas web, en las cuales puedes descargar un software para diseñar tu propio banquete. Con la ayuda de una serie de restricciones, dicho software utiliza un algoritmo que, a pesar de que cada compañía mantiene en secreto, se trata de un evolutivo de alguno de los ya conocidos públicamente. En la Figura 3.3 podemos ver un ejemplo, obtenido al utilizar el software ofrecido por una de esas páginas web[2].

3.2.1. Relación con los Problemas de Grafos

Dado un conjunto de N invitados, la forma más simple para construir un plan de distribución de asientos se puede definir utilizando una matriz binaria $\mathbf{W} \in \mathbb{R}_{N \times N}$, donde el elemento $W_{ij} = 1$ si los invitados i y j deben sentarse separados y $W_{ij} = 0$ en cualquier otro caso. Además, se puede considerar que dicha matriz es simétrica, es decir, $W_{ij} = W_{ji}$. Tomando esta matriz de entrada, el objetivo es dividir el conjunto de N invitados en k subconjuntos $\mathcal{S} = \{S_1 \ldots, S_k\}$, tales que minimicen la siguiente función objetivo:

$$f(\mathcal{S}) = \sum_{t=1}^{k} \sum_{\forall i,j \in S_t : i<j} W_{ij}. \tag{3.1}$$

De esta forma, el problema de confirmar la existencia de una solución de coste cero para este problema es equivalente al problema de coloración de grafos, donde $G = (V, E)$ se construye utilizando el conjunto de vértices $V = \{v_1, \ldots, v_N\}$ y el conjunto de aristas $E = \{\{v_i, v_j\} : W_{ij} = 1 \wedge v_i, v_j \in V\}$, es decir, cada invitado se corresponde con un vértice, y dos vértices v_i y v_j se consideran adyacentes si y sólo si $W_{ij} = 1$. Los colores se corresponden con las mesas, y el objetivo es obtener una coloración de G utilizando k colores.

[2] www.perfecttableplan.com

En algunas situaciones, puede ser preferible que **W** sea una matriz con valores enteros o incluso reales, permitiendo así a los usuarios dar diferentes grados de importancia a sus preferencias con respecto al resto de los invitados.

3.2.2. El Problema del Banquete de Boda

A continuación, describimos el conocido como WSP (Wedding Seating Problem), muy relacionado con el problema de coloración de grafos, pero que incluye además una restricción para que se agrupe a aquellos invitados que prefieren sentarse juntos, siempre y cuando se mantenga un número apropiado de invitados por mesa.

Inicialmente, los N invitados se dividen en $n \leq N$ *grupos de invitados*. Cada grupo se asocia a un subconjunto de invitados que prefieren sentarse juntos (parejas, familias, etc.) y que el organizador conoce de antemano. En base a esto, el WSP puede formularse como un tipo de problema de particionamiento de grafos. Dicho de forma más técnica, partimos de un grafo $G = (V, E)$, donde cada vértice $v_i \in V$ $(i = 1, \dots n)$, representa un grupo de invitados. El tamaño de cada grupo de invitados $v \in V$ se denota por s_v, y el número total de invitados se corresponde con $N = \sum_{v \in V} s_v$.

Cada arista $\{u, v\} \in E$ define un relación entre los vértices u y v, de acuerdo a un peso w_{uv}, donde $w_{uv} = w_{vu}$. Si $w_{uv} > 0$, entonces se interpreta que los invitados asociados a los vértices u y v es preferible que se sienten en mesas diferentes. Análogamente, valores negativos de w_{uv} implican que los invitados asociados a los vértices u y v se sienten en la misma mesa.

De esta forma, una solución para el WSP se define como una partición de los vértices en k subconjuntos $\mathcal{S} = \{S_1, \dots, S_k\}$, donde cada subconjunto S_i define a los invitados asignados a la mesa i y k es un valor definido por el usuario, representando el número total de mesas.

Se ha demostrado que el WSP pertenece a la clase de problemas del tipo NP-completos, pues generaliza dos de los problemas NP-completos más conocidos, como son el problema de las k-particiones y el problema de las k-coloraciones de grafos, por lo que para obtener soluciones al problema en un tiempo computacional razonable, deberemos recurrir a alguno de los algoritmos analizados en el Capítulo 2.

Ahora que ya hemos definido el problema, veamos cuáles son las funciones objetivo a minimizar y así poder calificar la calidad de las soluciones candidatas al WSP. En primer lugar, la función objetivo

$$f_1 = \sum_{i=1}^{k} \sum_{\forall u,v \in S_i : \{u,v\} \in E} (s_v + s_u) w_{uv}, \tag{3.2}$$

refleja la medida en que se obedecen las normas encargadas de determinar quién se sienta con quién. Podemos ver como w_{uv} se multiplica por el tamaño total de los dos grupos involucrados, $s_v + s_u$, de forma que al incumplir restricciones que afectan a un mayor número de personas, contribuya en mayor medida al coste de la función objetivo.

La segunda de las funciones objetivo asociadas al WSP pretende fomentar la igualdad con respecto al número de invitados asignados a cada una de las mesas. Asumiendo que el tamaño de todas las mesas es el mismo, esta segunda función objetivo mide el grado en que el número de invitados por mesa se desvía del valor requerido, es decir, tanto de $\lfloor N/k \rfloor$ como de $\lceil N/k \rceil$[3]. De esta forma, se define la función objetivo f_2 como

[3] $\lfloor\ \rfloor$ y $\lceil\ \rceil$ representan las funciones parte entera: función suelo y función techo, respectivamente.

$$f_2 = \sum_{i=1}^{k} (\min(|\tau_i - \lfloor N/k \rfloor|, |\tau_i - \lceil N/k \rceil|)), \tag{3.3}$$

donde $\tau_i = \sum_{\forall v \in S_i} s_v$ denota el número de invitados asignados a la mesa i.

El WSP también puede ser planteado como un problema de Programación Entera (IP), considerando una extensión del modelo visto en la Sección 2.2.2 del Capítulo 2. Teniendo en cuenta que disponemos de n grupos de invitados que queremos distribuir en k mesas, podemos recoger las preferencias de los invitados en una matriz simétrica $\mathbf{W} \in \mathbb{R}_{N \times N}$ donde

$$W_{ij} = \begin{cases} \infty & \text{si los invitados } i \text{ y } j \text{ no se pueden sentar juntos;} \\ 1 & \text{si los invitados } i \text{ y } j \text{ es preferible que se sienten en mesas diferentes;} \\ -1 & \text{si los invitados } i \text{ y } j \text{ es preferible que se sienten en la misma mesa;} \\ 0 & \text{en otro caso.} \end{cases}$$

De la misma forma que en el planteamiento como problema de particionamiento de grafos, definimos s_i como el tamaño de cada grupo de invitados, $i \in \{1, \ldots, n\}$. Así, una solución al problema se puede representar mediante una matriz $\mathbf{X} \in \mathbb{R}_{n \times n}$, donde

$$X_{it} = \begin{cases} 1 & \text{si el grupo de invitados } i \text{ es asignado a la mesa } t, \\ 0 & \text{en otro caso,} \end{cases} \tag{3.4}$$

y el vector binario $Y \in \mathbb{R}^n$, donde

$$Y_t = \begin{cases} 1 & \text{si al menos un grupo de invitados es asignado a la mesa } t, \\ 0 & \text{en otro caso,} \end{cases} \tag{3.5}$$

verificando las siguientes restricciones:

$$\sum_{t=1}^{n} X_{it} = 1 \ \forall i \in \{1, \ldots, n\} \tag{3.6}$$

$$X_{it} + X_{jt} \leq Y_t \ \forall i, j : W_{ij} = \infty, \forall t \in \{1, \ldots, n\} \tag{3.7}$$

$$X_{it} = 0 \ \forall i \in \{1, \ldots, n\}, \ \forall t \in \{i+1, \ldots, n\} \tag{3.8}$$

$$X_{it} = \sum_{j=t-1}^{i-1} X_{j,t-1} \ \forall i \in \{2, \ldots, n\}, \ \forall t \in \{2, \ldots, i-1\} \tag{3.9}$$

$$\sum_{t=1}^{n} Y_t \leq k, \tag{3.10}$$

donde la Restricción (3.6) garantiza que todos los grupos de invitados se asignan a una única mesa, la Restricción (3.7) obliga a cumplir todas aquellas restricciones en las que los invitados no se pueden

sentar juntos y que $Y_t = 1$ si y sólo si existe un grupo de invitados que ha sido asignado a la mesa t. Las Restricciones (3.8) y (3.9) imponen las condiciones de anti-simetría y finalmente la Restricción (3.10) determina que no pueden utilizarse más de k mesas.

En cuanto a la calidad de las soluciones factibles candidatas, en este caso también se puede cuantificar a partir de dos funciones objetivo, siendo

$$f_1 = \sum_{t=1}^{k} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{it} X_{jt} (s_i + s_j) W_{ij}, \tag{3.11}$$

expresión equivalente a la utilizada en la Ecuación (3.2), y f_2 la misma que la utilizada en la Ecuación (3.3), salvo por el hecho de que τ_i en este caso se calcula como $\tau_i = \sum_{j=1}^{n} X_{jt} s_j$, y siendo también equivalente a la Ecuación (3.3).

3.3. Horarios Universitarios

Finalmente, veamos cómo aplicar los conceptos de coloración de grafos en la construcción de horarios universitarios (u otros tipos de centros educativos, como pueden ser colegios, institutos o centros de formación profesional). Aunque para cada caso concreto puede existir alguna restricción que haga que la dificultad de resolver el problema aumente, en general, detrás de la mayoría de los problemas de horarios, subyace un problema de coloración de grafos.

Podemos considerar un horario como una asignación de eventos (tales como clases teóricas, seminarios o exámenes), sobre un número finito de aulas e intervalos de tiempo de acuerdo a una serie de restricciones, algunas de las cuales serán obligatorias y otras, opcionales. En un trabajo de Corne et al. (1995), se propone una clasificación de las posibles restricciones en cinco clases:

- Restricciones Unitarias: involucran un único evento, como por ejemplo "el evento a no puede tener lugar los Martes" o "el evento a debe tener lugar en el intervalo de tiempo b".
- Restricciones Binarias: involucran pares de eventos, como por ejemplo "el evento a debe tener lugar antes del evento b" o la restricción del *evento conflictivo*, el cual determina pares de eventos que no pueden tener lugar al mismo tiempo.
- Restricciones de Capacidad: asociadas a la capacidad de cada aula, por ejemplo, "todos los eventos deben ser asignados en un aula que tenga capacidad suficiente".
- Restricción de Propagación del Evento: involucran aquellos requisitos sobre "separar" o "agrupar" eventos en el horario, de forma que se facilite la carga de trabajo tanto de alumnos como de profesores, siempre que se cumpla la política de horarios propia de la universidad.
- Restricciones de Agente: impuestas con el objetivo de promover los requisitos y preferencias de aquellas personas que vayan a utilizar dicho horario, como por ejemplo "al profesor a le gusta dar clase del evento b los Lunes" o "el profesor c debe tener d mañanas libres por semana".

De la misma forma que sucede en muchos problemas de investigación operativa, existe una convención con respecto a los horarios automatizados para agrupar las restricciones en dos clases: restricciones *fuertes* y restricciones *débiles*, donde resulta evidente que las fuertes, que suelen tener la condición de ser obligatorias, tienen prioridad con respecto a las débiles, las cuales deben cumplirse siempre que sea posible.

Tal y como proponen McCollum et al. (2010), el problema de construir un horario universitario puede dividirse en dos categorías: problemas de horarios de exámenes y problemas de horarios de curso académico, pudiendo dividir este último en otras dos categorías: "horarios de curso académico posteriores a la matrícula", donde las restricciones están determinadas por los datos asociados a las matrículas de los alumnos, y "horarios de curso académico basados en los planes de estudio", donde las restricciones se basan en los planes de estudio especificados por la universidad.

La restricción binaria conocida como *evento conflictivo* es una de las restricciones más importantes a la hora de diseñar un horario, pues especifica que, si una persona (o algún otro recurso del que se disponga una única unidad), es requerido para estar presente en un par de eventos, entonces estos no pueden ser asignados al mismo intervalo de tiempo, pues esto implicaría que dicha persona tendría que estar en dos lugares al mismo tiempo. Esta restricción permite establecer una conexión con el problema de coloración de grafos, considerando los eventos como vértices, los eventos conflictivos como aristas y los intervalos de tiempo como colores.

A pesar de que cada universidad tendrá su política de horarios y sus propias necesidades, dando lugar a diferentes restricciones, de una forma o de otra, sobre todos ellos subyace el problema de coloración de grafos, y muchos de los algoritmos utilizados se basan en informaciones heurísticas extraídas del problema de coloración de grafos. En consecuencia, trabajos como el de Cooper y Kingston (1996), han ayudado a demostrar que los problemas de horarios universitarios pertenecen a la clase de problemas NP-completos.

A continuación, analizamos en detalle el problema de los horarios de curso académico posteriores a la matrícula. Esta formulación modela una situación real en la que los estudiantes proporcionan una selección de conferencias (o clases) a las que les gustaría asistir, construyendo posteriormente el horario correspondiente en base a dichas preferencias.

3.3.1. El Problema de Horarios Pos-matrícula

Definición del problema

Este problema involucra siete tipos de restricciones fuertes cuyo cumplimiento es obligatorio, y tres restricciones débiles, cuyo cumplimiento es deseable, pero no esencial. En consecuencia, el problema consiste en asignar un conjunto de eventos sobre 45 intervalos de tiempo (cinco días, con nueve intervalos de tiempo por día), de acuerdo a las restricciones anteriores.

Veamos en primer lugar el planteamiento de las restricciones fuertes. Para cada evento, hay un conjunto de estudiantes que se han matriculado con la intención de asistir. Los eventos deben ser asignados a los intervalos de tiempo de forma que ningún estudiante tenga que estar en más de un evento en el mismo intervalo de tiempo. Además, cada evento requiere un conjunto de características asociadas al aula, como por ejemplo, un número determinado de sillas, un ordenador o un proyector, lo que quizá no cumplan todas las aulas, de forma que cada evento deberá ser asignado a un aula que cumpla con las condiciones requeridas. Es evidente que tampoco se permite asignar el mismo aula para dos eventos diferentes. Finalmente, se añade la restricción de que algunos eventos no pueden tener lugar en algún intervalo de tiempo concreto, así como restricciones de precedencia, estableciendo que algunos eventos deben ser programados antes o después que otros.

Formalmente, el problema consiste en un conjunto de eventos $E = \{e_1 \ldots, e_n\}$, un conjunto de intervalos de tiempo $T = \{t_1, \ldots, t_k\}$ (donde $k = 45$), un conjunto de estudiantes $S = \{s_1, \ldots, s_s\}$, un conjunto de aulas $R = \{r_1, \ldots, r_r\}$ y un conjunto de características asociadas a cada aula $F = \{f_1, \ldots, f_f\}$. Cada aula $r_i \in R$ tiene asignada una capacidad $c(r_i)$, representando el número de sillas

disponibles. De esta forma, se dispone de

- una matriz de *asistencia* $\mathbf{P}^{(1)} \in \mathbb{R}_{s\times n}$ donde

$$P_{ij}^{(1)} = \begin{cases} 1 & \text{si el estudiante } s_i \text{ debe asistir al evento } e_j, \\ 0 & \text{en otro caso,} \end{cases} \tag{3.12}$$

- una matriz de *propiedades de aulas* $\mathbf{P}^{(2)} \in \mathbb{R}_{r\times f}$ donde

$$P_{ij}^{(2)} = \begin{cases} 1 & \text{si el aula } r_i \text{ tiene la propiedad } f_j, \\ 0 & \text{en otro caso,} \end{cases} \tag{3.13}$$

- una matriz de *propiedades de eventos* $\mathbf{P}^{(3)} \in \mathbb{R}_{n\times f}$ donde

$$P_{ij}^{(3)} = \begin{cases} 1 & \text{si el evento } e_i \text{ requiere la propiedad } f_j, \\ 0 & \text{en otro caso,} \end{cases} \tag{3.14}$$

- una matriz de *disponibilidad de eventos* $\mathbf{P}^{(4)} \in \mathbb{R}_{n\times k}$ donde

$$P_{ij}^{(4)} = \begin{cases} 1 & \text{si el evento } e_i \text{ puede ser asignado al intervalo de tiempo } t_j, \\ 0 & \text{en otro caso,} \end{cases} \tag{3.15}$$

- y una matriz de *precedencia* $\mathbf{P}^{(5)} \in \mathbb{R}_{n\times n}$ donde

$$P_{ij}^{(5)} = \begin{cases} 1 & \text{si el evento } e_i \text{ debe ser asignado a un intervalo de tiempo anterior al evento } e_j, \\ -1 & \text{si el evento } e_i \text{ debe ser asignado a un intervalo de tiempo posterior al evento } e_j, \\ 0 & \text{en otro caso.} \end{cases} \tag{3.16}$$

Para la Ecuación (3.16), asociada a la matriz de precedencia, se necesita introducir dos condiciones adicionales para que las relaciones sean consistentes: $P_{ij}^{(5)} = 1$ si y sólo si $P_{ji}^{(5)} = -1$, y $P_{ij}^{(5)} = 0$ si y sólo si $P_{ji}^{(5)} = 0$.

Además de las cinco matrices ya planteadas, se pueden añadir dos matrices más, de forma que podamos detectar más rápidamente el incumplimiento de las restricciones fuertes. En primer lugar, definimos la matriz de *idoneidad del aula* $\mathbf{R} \in \mathbb{R}_{n\times r}$ como:

$$R_{ij} = \begin{cases} 1 & \text{si } \left(\sum_{l=1}^{s} P_{li}^{(1)} \leq c(r_j)\right) \wedge \left(\nexists f_l \in f : \left(P_{il}^{(3)} = 1 \wedge P_{jl}(2) = 0\right)\right), \\ 0 & \text{en otro caso.} \end{cases} \tag{3.17}$$

La segunda de ellas es la matriz *de conflictos* $\mathbf{C} \in \mathbb{R}_{n\times n}$, definida como:

$$C_{ij} = \begin{cases} 1 & \text{si } \left(\exists s_l \in s : \left(P^{(1)}_{li} = 1 \wedge P^{(1)}_{lj} = 1\right)\right) \\ & \vee\left(\exists r_l \in r : \left(R_{il} = 1 \wedge R_{jl} = 1\right)\right) \wedge \left(\sum_{l=1}^{r} R_{il} = 1\right) \wedge \left(\sum_{l=1}^{r} R_{jl} = 1\right) \\ & \vee\left(P^{(5)}_{ij} \neq 0\right) \\ & \vee\left(\nexists t_l \in t : \left(P^{(4)}_{il} = 1 \wedge P^{(4)}_{jl} = 1\right)\right), \\ 0 & \text{en otro caso.} \end{cases} \tag{3.18}$$

La matriz $\mathbf{R}$ especifica aquellas aulas que cumplen las condiciones necesarias para cada evento, mientras que la matriz $\mathbf{C}$ es una matriz simétrica ($C_{ij} = C_{ji}$) que especifica pares de eventos que no pueden ser asignados al mismo intervalo de tiempo (aquellos que provocan un conflicto). Esto puede suceder si, por ejemplo, dos eventos e_i y e_j tienen un estudiante en común, necesitan impartirse en el mismo aula, están sujetos a una relación de precedencia o son mutuamente excluyentes con respecto a los intervalos de tiempo para los que están disponibles.

Resulta interesante ver cómo se relaciona el problema de horarios con el problema de coloración de grafos, viendo que la matriz $\mathbf{C}$ es análoga a la matriz de adyacencia de un grafo $G = (V, E)$ con n vértices. Sin embargo, a diferencia del problema de coloración de grafos, en este caso el orden de los intervalos de tiempo, que se corresponde con las clases de color, influye en gran medida a la hora de obtener una solución. En consecuencia, una solución al problema de horarios se representa como un conjunto ordenado de conjuntos $\mathcal{S} = \{S_1, \ldots, S_{k=45}\}$ sujeto a las siguientes restricciones fuertes.

$$\bigcup_{i=1}^{k} S_i \subseteq E \tag{3.19}$$

$$S_i \cap S_j = \emptyset \ (1 \leq i \neq j \leq k) \tag{3.20}$$

$$\forall e_j, e_l \in S_i, C_{jl} = 0 \ (1 \leq i \leq k) \tag{3.21}$$

$$\forall e_j \in S_i, P^{(4)}_{ji} = 1 \ (1 \leq i \leq k) \tag{3.22}$$

$$\forall e_j \in S_i, e_l \in S_{q<i}, P^{(5)}_{jl} \neq 1 \ (1 \leq i \leq k) \tag{3.23}$$

$$\forall e_j \in S_i, e_l \in S_{q>i}, P^{(5)}_{jl} \neq -1 \ (1 \leq i \leq k) \tag{3.24}$$

$$S_i \in \mathcal{M} \ (1 \leq i \leq k) \tag{3.25}$$

En las Restricciones (3.19) y (3.20) se determina que S debe particionar el conjunto de eventos E (o un subconjunto de este), en un conjunto ordenado de conjuntos, etiquetados como $S_1, \ldots, S_k$. Cada conjunto $S_i \in \mathcal{S}$ contiene eventos asignados al intervalo de tiempo t_i. La restricción (3.21) determina que ningún par de eventos conflictivos deben ser asignados al mismo conjunto $S_i \in \mathcal{S}$, mientras que la restricción (3.22) determina que cada evento debe ser asignado a un conjunto $S_i \in S$ cuyo intervalo de tiempo correspondiente t_i está disponible en base a la matriz $\mathbf{P}^{(4)}$ y las restricciones (3.23) y (3.24) determinan los requerimientos de precedencia. En último lugar, la restricción (3.25) tiene el objetivo

de garantizar que los eventos asignados a un conjunto $S_i \in \mathcal{S}$ pueden ser asignados a un aula apropiada del conjunto de aulas R. Para que esto sea posible, es necesario resolver un problema de maximización de correspondencias bipartitas. Veamos esto en detalle. Sea $G = (S_i, r, E)$ un grafo bipartito con S_i y r como conjuntos de vértices y un conjunto de aristas $E = \{\{e_j \in S_i, r_l \in r\} : R_{lj} = 1\}$. Dado el grafo G, el conjunto S_i es un elemento de $\mathcal{M}$ si y sólo si existe una correspondencia bipartita máxima de G de tamaño $2|S_i|$, donde en cada caso, las restricciones asociadas a cada aula para ese intervalo de tiempo deben cumplirse.

Una solución $\mathcal{S}$ se considera válida si y sólo si se verifican todas estas restricciones. La calidad de una solución válida se mide en base a la *distancia a factibilidad* (DTF), medida que se calcula como la suma de los tamaños de todos los eventos que no están presentes en la solución:

$$DTF = \sum_{e_i \in \mathcal{S}'} \sum_{j=1}^{|s|} P_{ij}^{(1)} \tag{3.26}$$

donde $\mathcal{S}' = E \setminus \left(\bigcup_{i=1}^{k} S_i\right)$. Si la solución $\mathcal{S}$ es válida y tiene DTF cero, es decir, $E = \left(\bigcup_{i=1}^{k} S_i\right)$ y equivalentemente $\mathcal{S}' = \emptyset$, entonces $\mathcal{S}$ se considera *factible*, pues todos los eventos han sido asignados a un intervalo de tiempo de forma factible.

Veamos, para finalizar, el planteamiento de las restricciones débiles. En este problema, planteamos tres:

- Los alumnos no deben asistir a un evento en el último intervalo de tiempo de cada día;
- Los alumnos no deben asistir a tres o más eventos en intervalos de tiempo consecutivos en el mismo día; y,
- Los alumnos no deben asistir a un único evento en un día.

La forma en la que estas restricciones se cumplen o no, se mide en base a la medida de *Coste de las Restricciones Débiles* (SCC), que actúa como sigue:

Si un alumno asiste a un evento en el último intervalo de tiempo del día, esto se registra como un punto de penalización. Evidentemente, si en el evento correspondiente hay x alumnos, entonces se registran x puntos de penalización. Si un alumno asiste a tres eventos de forma consecutiva, entonces se registra un punto de penalización. Si asiste a cuatro, dos puntos de penalización, y así sucesivamente. Si un alumno tiene una única clase en un día determinado, se registra como un punto de penalización. En consecuencia, la medida SCC se puede obtener como la suma de estos tres valores. Más formalmente, la SCC puede calcularse utilizando dos matrices: $\mathbf{X} \in \mathbb{R}_{s\times k}$, que determina los intervalos de tiempo para los cuales cada alumno ya está asignado a un evento, y $\mathbf{Y} \in \mathbb{R}_{s\times 5}$, que determina cuándo un estudiante es requerido para asistir a un único evento en cada uno de los cinco días. Así, resulta:

$$X_{ij} = \begin{cases} 1 & \text{si } \exists\ e_l \in S_j : P_{il}^{(1)} = 1 \\ 0 & \text{en otro caso.} \end{cases} \tag{3.27}$$

$$Y_{ij} = \begin{cases} 1 & \text{si } \sum_{l=1}^{9} X_{i,9(j-1)+l} = 1 \\ 0 & \text{en otro caso.} \end{cases} \tag{3.28}$$

Utilizando estas matrices, la SCC se calcula como:

$$SCC = \sum_{i=1}^{s}\sum_{j=1}^{5}\left((X_{i,9j}) + \left(\sum_{l=1}^{7}\prod_{q=0}^{2} X_{i,9(j-1)+l+q}\right) + (Y_{i,j})\right), \tag{3.29}$$

donde cada uno de los elementos entre paréntesis determina el número de veces que cada estudiante incumple cada una de las restricciones débiles anteriormente definidas.

A la hora de comparar soluciones, aquel problema que tenga el menor DTF será considerado el mejor horario. Si dos o más soluciones tuvieran el mismo DTF, el mejor horario sería aquel con menor SCC. De esta forma, se refleja la mayor importancia de las restricciones fuertes sobre las débiles.

Para finalizar, mostraremos, con un ejemplo sencillo, la resolución de un problema de horarios utilizando la coloración de grafos. Para ello, en la Figura 3.4(a) presentamos un problema de horarios como un problema de coloración de grafos. Partimos de nueve eventos que deben ser distribuidos en cuatro intervalos de tiempo. En la Figura 3.4(b), presentamos una 4-coloración factible del grafo y en el Cuadro 3.1, el horario asociado a dicha coloración. Para obtener este resultado, consideramos que durante el Intervalo 1, se dispone de al menos tres aulas, para que puedan tener lugar los tres eventos en ese intervalo de tiempo.

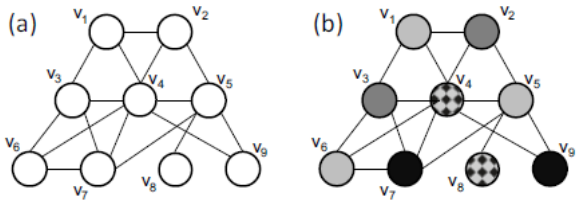


Figura 3.4: Ejemplo de un problema de horarios mediante coloración de grafos.

Intervalo 1	**Intervalo 2**	**Interval 3**	**Intervalo 4**
Evento 1	Evento 2	Evento 7	Evento 4
Evento 5	Evento 3	Evento 9	Evento 8
Evento 6			

Cuadro 3.1: Solución al problema de horarios.

Bibliografía

[1] Avanthay C, Hertz A, Zufferey N (2003) A variable neighborhood search for graph coloring. European Journal of Operational Research, 151(2): 379-388.

[2] Blöchliger I, Zufferey N (2008) A graph coloring heuristic using partial solutions and a reactive tabu scheme. Computers and Operations Research, 35: 960-975.

[3] Brélaz D (1979) New methods to color the vertices of a graph. Commun. ACM, 22(4): 251-256.

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

[5] Cooper T B, Kingston J H (1995, Agosto) The complexity of timetable construction problems. In International Conference on the Practice and Theory of Automated Timetabling (pp. 281-295). Springer, Berlin, Heidelberg.

[6] Corne D, Ross P, Fang H (1995) Evolving timetables. The practical handbook of genetic algorithms, 1: 219-276.

[7] Dorne R, Hao J K (1998, Septiembre) A new genetic local search algorithm for graph coloring. In International Conference on Parallel Problem Solving from Nature (pp. 745-754). Springer, Berlin, Heidelberg.

[8] Eiben Á E, van Der Hauw J K, van Hemert J I (1998) Graph coloring with adaptive evolutionary algorithms. Journal of Heuristics, 4(1): 25-46.

[9] Fleurent C, Ferland J A (1996) Genetic and hybrid algorithms for graph coloring. Annals of Operations Research, 63(3): 437-461.

[10] Galinier P, Hamiez J P, Hao J K, Porumbel D (2013) Recent advances in graph vertex coloring. Handbook of optimization: 505-528.

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

[12] Glover F (1986) Future paths for integer programming and links to artificial intelligence. Computers and operations research, 13(5): 533-549.

[13] Glover F (1989) Tabu search-part I. ORSA Journal on computing, 1(3): 190-206.

[14] Glover F (1990) Tabu search-part II. ORSA Journal on computing, 2(1): 4-32.

[15] Hertz A, de Werra D (1987) Using tabu search techniques for graph coloring. Computing, 39(4): 345-351.

[16] Korman S M (1979) The graph-colouring problem. Combinatorial optimization: 211-235.

[17] Laguna M, Martí R (2001) A GRASP for coloring sparse graphs. Computational optimization and applications, 19(2): 165-178.

[18] Leighton F T (1979) A graph coloring algorithm for large scheduling problems. Journal of Research of the National Bureau of Standards, 84(6):489-506.

[19] Lewis R (2009) A general-purpose hill-climbing method for order independent minimum grouping problems: A case study in graph colouring and bin packing. Computers and Operations Research, 36(7): 2295-2310.

[20] Lewis R (2015) A Guide to Graph Colouring. Springer, Berlin.

[21] McCollum B, Schaerf A, Paechter B, McMullan P, Lewis R, Parkes A J, Di Gaspero L, Qu R, Burke E K (2010). Setting the research agenda in automated timetabling: The second international timetabling competition. INFORMS Journal on Computing, 22(1): 120-130.

[22] O'Connor J J y Robertson E F (1996) The four colour theorem. MacTutor History of Mathematics archive. $http://www-history.mcs.st-andrews.ac.uk/HistTopics/The\_four\_colour\_theorem.html$. Accedido en Mayo de 2017.

[23] Paquete L, Stützle T (2002, Enero) An experimental investigation of iterated local search for coloring graphs. In EvoWorkshops (pp. 122-131).

[24] Ramírez Rodríguez J (2001) Extensiones del Problema de Coloración de Grafos. Tesis, Facultad de Ciencias Matemáticas, Universidad Complutense de Madrid.

[25] Stadler M M (2006) Mapas, colores y números. Matemáticas en la ciencia y la cultura contemporáneas, 6(18): 27.

[26] Thompson J M, Dowsland K A, (2008) An improved ant colony optimisation heuristic for graph colouring. Discrete Applied Mathematics, 156(3): 313-324.