Universidad de La Laguna

Sección de Matemáticas | Facultad de Ciencias

Daniel Díaz Ríos

Optimización de Redes de Comunicaciones Móviles

Mobile Network Optimization

Trabajo Fin de Grado
Grado en Matemáticas
La Laguna, Junio de 2018

Dirigido por

Antonio A. Sedeño Noda
David Ortega Sicilia

Antonio A. Sedeño Noda
Departamento de Matemáticas, Estadística e Investigación Operativa
Universidad de La Laguna
38271 La Laguna, Tenerife

David Ortega Sicilia
Optimización Radio Móvil Telefónica

Agradecimientos

A Antonio por guiarme.

A David por asesorarme.

A mi familia por apoyarme.

Y a mis amigos por encontrarme.

Simplemente, gracias.

Resumen · Abstract

Resumen

Dentro de las redes de comunicaciones móviles, existe un problema de asignaciones de un recurso limitado llamado "Physical Cell Id", que identifica las células a las que se conecta un terminal móvil. Este trabajo trata de modelizar y resolver este problema partiendo de un caso real, encontrando similitudes con los problemas de coloración de grafos, y realizando una búsqueda bibliográfica para encontrar y utilizar los mejores algoritmos. El objetivo persigue optimizar esta asignación para mejorar el funcionamiento de la red de comunicaciones LTE.

Palabras clave: *Physical Cell Id – Optimización – Problema de coloración de grafos – Adaptación de grafos – Algoritmos de coloración – Algoritmo estocástico.*

Abstract

Within the mobile communications networks, there is an assignment problem with a limited resource called "Physical Cell Id", which identifies the cells that a mobile terminal connects to. This paper tries to model and solve this problem starting out from a real case, finding some similarities with graph colouring problems, and carrying out a bibliographic search to find and use the best algorithms. The aim is to optimize this allocation to improve the behaviour of the LTE communication network.

Keywords: *Physical Cell Id – Optimization – Graph colouring problem – Graph adaptation – Graph colouring algorithms – Stochastic algorithm.*

Contenido

Introducción

A lo largo del Grado en Matemáticas se adquiere una gran cantidad de conocimientos en diferentes ámbitos, pero no solo se limitan estrictamente a nuestro campo, sino que se extienden a disciplinas afines como la informática y otras ciencias. Sin embargo, permanece una característica latente entre la mayoría de las materias cursadas, y es que se estudian a modo teórico, muy rara vez sobre datos reales donde podamos ayudar a resolver problemas y ser de utilidad. En este TFG se parte de un problema real, el proceso que seguiremos vendrá determinado por el propósito de mejorar los resultados, centrándonos más en el aspecto práctico y resolutivo que en el teórico.

Concretamente, el objetivo de este estudio es reducir el número Handovers fallidos mientras se utiliza el terminal móvil, ya que una de las principales causas de este suceso, son las colisiones y confusiones entre células de telefonía, conceptos que se definirán más rigurosamente a lo largo del trabajo. Esto puede ser evitado con una correcta distribución de los recursos usados, la cual tiene similitudes con un problema de optimización matemático llamado problema de coloración. Además trataremos de reducir la utilización de estos recursos.

Comenzaremos con la parte más teórica del trabajo, donde se definirán conceptos necesarios que serán de ayuda o usaremos a lo largo de la resolución del problema, así como la explicación de algunos algoritmos ya existentes que utilizaremos. A continuación describiremos de forma general el problema real, y algunos procesos que deben ser seguidos para afrontarlo. Finalmente abordaremos nuestro problema concreto, donde se especifica como ha sido resuelto, todos los obstáculos con los que nos hemos encontrado, qué pasos hemos seguido para poder resolverlos, y qué soluciones acabamos obteniendo.

1

Fundamentos teóricos

En esta sección se definirá la terminología básica y conceptos necesarios que se usarán a lo largo del proyecto.

1.1. Teoría de grafos y representación

- Un **grafo dirigido G=(V,A)** está definido por un conjunto V de nodos o vértices y un conjunto $A \subseteq V \times V$, cuyos elementos denominados arcos son pares ordenados de nodos.

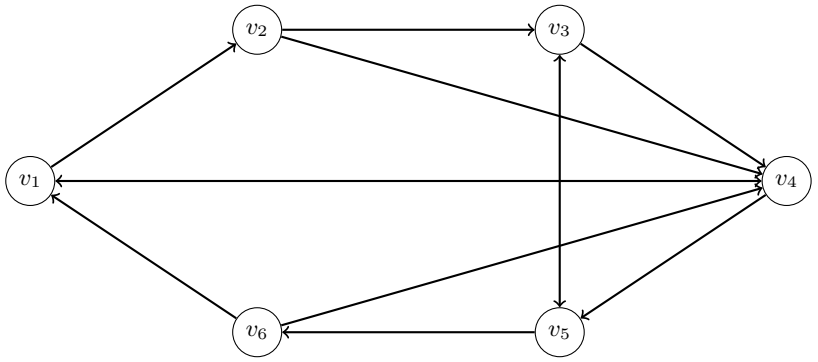


Figura 1.1. Grafo dirigido.

Se denota típicamente $|V| = n$ como cardinal del conjunto V (número de vértices) y $|A| = m$ cardinal del conjunto A (número de arcos).

- Un **grafo no dirigido G=(V,E)** está definido por un conjunto V de nodos o vértices y un conjunto E, cuyos elementos denominados aristas son pares no ordenados de nodos.

  Nuevamente $|V| = n$ denota el número de vértices y $|E| = m$ en ese caso el número de aristas.

- Sea G un grafo y a un arco o arista, se dirá que a es un **lazo o bucle** si los dos nodos que enlaza son el mismo, y se dirá que G es un ***p-grafo*** si contiene a lo sumo p arcos o aristas para cada par de nodos $i, j \in V$.
- Si estamos ante un *1-grafo* y no contiene lazos, entonces lo llamaremos **grafo simple**.
- Los **grafos planos**, son aquellos que pueden ser dibujados en un plano sin que ningún arco o arista se cruce.
- Sea G un grafo, entonces G' será un **subgrafo** de G si es un grafo, su conjunto de vértices está contenido en el de G ($V' \subseteq V$) y su conjunto de arcos o aristas está también contenido en el de G.
- Sea $G = (V, E)$ un grafo no dirigido, entonces $G' = (V', E')$ es un **clique** de G si es un subgrafo del mismo tal que $\forall i, j \in V', \exists \{i, j\} \in E'$.
- Sea $G = (V, A)$ un grafo dirigido:
  - El nodo j es un **sucesor** del nodo i, si $\exists (i, j) \in A$.
  - El nodo i es un **predecesor** del nodo j, si $\exists (i, j) \in A$.
  - El **conjunto de sucesores** del nodo i se define por: $\Gamma_i^+ = \{j : (i, j) \in A\}$
  - El **conjunto de predecesores** del nodo i se define por: $\Gamma_i^- = \{j : (i, j) \in A\}$
- Sea $G = (V, E)$ un grafo no dirigido, definimos los **nodos adyacentes** al nodo i por el conjunto de todos los nodos del grafo que estén unidos a i por alguna arista: $\Gamma_i = \{j : (i, j) \in E\}$.
- Dado un grafo dirigido, el **grado de salida** del nodo i vendrá dado por: $\delta^+(i) = |\Gamma_i^+|$, el **grado de entrada** por: $\delta^-(i) = |\Gamma_i^-|$ y el **grado** por: $\delta = \delta^+(i) + \delta^-(i)$. Si el grafo es no dirigido entonces el **grado** será: $2|E|$.

1.1.1. Matrices

La **matriz de incidencias** de un grafo es una matriz $B = (b_{ij})$, donde las filas representan los nodos y las columnas los arcos o aristas, de forma que:

- En un grafo dirigido, habrá un 1 ($b_{ij} = 1$) en la fila i columna j si el nodo i es el nodo inicial del arco j, habrá un -1 ($b_{ij} = -1$) si el nodo i es el nodo final del arco j y habrá un 0 ($b_{ij} = 0$) en otro caso.
- En un grafo no dirigido, habrá un 1 ($b_{ij} = 1$) en la fila i columna j si el nodo i es un extremo de la arista j y 0 en caso contrario.

La **matriz de adyacencia** de un *1-grafo* es una matriz cuadrada $A = (a_{ij})$, donde las filas y las columnas representan los nodos del grafo en el mismo orden, y habrá un 1 ($a_{ij} = 1$) en la fila i columna j si los nodos i, j están unidos por un arco (i, j) en un grafo dirigido (o una arista $\{i, j\}$ en un grafo no dirigido) y 0 ($a_{ij} = 0$) en caso contrario. Nótese que si el grafo es no dirigido, esta matriz es simétrica. Aquí un ejemplo del grafo en la Figura 1.1.

$$M = \begin{pmatrix} 0\,1\,0\,1\,0\,0 \\ 0\,0\,1\,1\,0\,0 \\ 0\,0\,0\,1\,1\,0 \\ 1\,0\,0\,0\,1\,0 \\ 0\,0\,1\,0\,0\,1 \\ 1\,0\,0\,1\,0\,0 \end{pmatrix}$$

1.2. Problema de coloración sobre grafos

Una **coloración de un grafo** es una distribución de colores asignados a cada vértice, de forma que cualquier par de vértices adyacentes no compartan el mismo color. El menor número de colores distintos necesarios para realizar dicha coloración se llama **número cromático**. Se sabe que el número cromático en los grafos planos es a lo sumo 4, gracias al *teorema de los cuatro colores*, véase en [5].

Una vez definidos todos estos conceptos, hablaremos del **problema de coloración de grafos**, cuyo objetivo es encontrar una coloración del grafo. El enfoque de este problema puede variar, es habitual además buscar el mínimo número de colores usados para la coloración, idealmente el número cromático. Otras veces el número de colores a usar está preestablecido, y no siempre se puede obtener una coloración, ya que puede no existir, en cuyo caso el objetivo del problema variaría a minimizar el número de vértices en contacto con el mismo color, de ahora en adelante, conflictos.

Una forma de modelar matemáticamente los problemas de coloración, es usando la programación lineal entera, cuya función objetivo será minimizar el número de colores usados, y las restricciones equivaldrán a que las distribuciones de colores obtenidas sean coloraciones.

Sea G=(V,E) un grafo no dirigido, las variables que utilizaremos las definimos como:

- Matriz binaria $X \in \Re_{n\times n}$ que representará los colores asociados a cada vértice, donde:

$$X_{ij} = \begin{cases} 1 \text{ si el nodo } v_i \text{ es asignado al color } j \\ 0 \text{ en caso contrario} \end{cases}$$

- Vector binario $Y \in \Re_{n\times 1}$ que representará los colores usados (a lo sumo tantos como nodos), donde:

$$Y_j = \begin{cases} 1 \text{ si al menos un nodo es asignado al color } j \\ 0 \text{ en caso contrario} \end{cases}$$

Cuya función objetivo será $min \sum_{j=1}^{n} Y_j$, y las restricciones vendrán dadas por:

- Todo par de vértices adyacentes tienen colores distintos, y $Y_j = 1$ si y solo si ese color es usado:

$$X_{ij} + X_{kj} \leq Y_j \quad \forall\{v_i, v_k\} \in E \wedge \forall j \in \{1, ..., n\}$$

- Cada vértice será asignado solamente a un color:

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

Por tanto el modelo final sería:

$$\begin{aligned} min \quad & \textstyle\sum_{j=1}^{n} Y_j & \\ sujeto\ a: \quad & X_{ij} + X_{kj} \leq Y_j & \forall\{v_i, v_k\} \in E \wedge \forall j \in \{1, ..., n\} \\ & \textstyle\sum_{j=1}^{n} X_{ij} = 1 & \forall v_i \in V \\ & X_{ij} \in \{0, 1\} & \forall i, j \in \{1, ..., n\} \\ & Y_j \in \{0, 1\} & \forall j \in \{1, ..., n\} \end{aligned}$$

Los problemas de coloración de grafos de tamaño lo suficientemente grande, son difíciles de resolver empleando un tiempo de cómputo razonable. Dado que estos pertenecen a una clase de problemas complicados, a continuación introduciremos unas nociones sobre complejidad computacional.

Si existe un algoritmo que resuelve un problema en tiempo de ejecución polinómico, es decir, que para un problema con datos de entrada de tamaño n, es resuelto en $O(n^k)$ para cierta constante k con un algoritmo determinista, se dirá que este problema pertenece a la **clase de complejidad P**. Si dada una solución a un problema, existe un algoritmo que nos puede verificar si esta es la solución óptima o no en tiempo de ejecución polinómico, se dirá que este problema pertenece a la **clase de complejidad NP**. Todo problema perteneciente a P también está contenido en NP, ya que si un problema puede ser resuelto en

tiempo polinómico, también puede ser verificado en tiempo polinómico, es decir $P \subset NP$.

Una **transformación polinomial**, es una función que permite cambiar la representación de un problema D_1 a otro problema D_2 aplicando un algoritmo determinista de tiempo polinomial. Se dirá que D_1 se transforma a D_2 y se representará como $D_1 \propto D_2$. Un problema D_1 pertenece a la **clase de complejidad NP-completo** (NPC) si pertenece a la clase NP, y para cualquier otro problema D_2 perteneciente a la clase NP, $D_2 \propto D_1$. No obstante, si todos los problemas de clase NP se pueden transformar en un problema A ($D_{NP} \propto A$), no siempre se puede demostrar que A pertenezca a la clase NP-completo, pero se dirá que pertenecen a la **clase de complejidad NP-duro**, la cual contiene a NP-completo (NP-completo $\subset$ NP-duro).

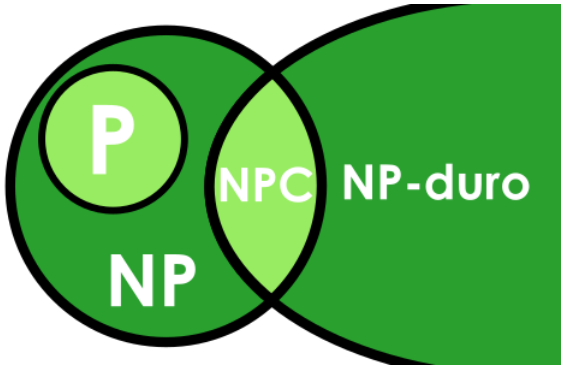


Figura 1.2. Clases de complejidad ($P \neq NP$).

Los problemas de coloración pertenecen a la clase NP-completo, por tanto si el grafo es grande, nos tendríamos que conformar con soluciones aproximadas dada la clase de complejidad a la que pertenece este problema.

1.3. Técnicas y algoritmos para la resolución exacta o aproximada del problema de coloración

1.3.1. Algoritmo de enumeración

Este algoritmo recursivo nos enumera todas las coloraciones posibles de un grafo, y toma la que menos colores necesita. Definimos el algoritmo con dos funciones, donde k es el número de nodos coloreados, n el número de nodos, A la matriz de adyacencia (booleana), *color* un vector con los colores que se le

asocian a cada vértice, y *ncol* el número de colores máximo a usar donde c va tomando el valor de cada uno de ellos.

Algorithm 1 Algoritmo de enumeración

```
La llamada inicial será coloración(0, A, color)
procedure COLORACION(k, A, color)
    if k = n then
        imprimir(color)
    else
        for c ← [1 : ncol] do
            color(k + 1) ← c
            if aceptable(k + 1, A, color) then
                coloracion(k + 1, A, color)
            end if
        end for
    end if
end procedure

Donde la función aceptable es:
procedure ACEPTABLE(ultimo, A, color)
    acept ← true
    nodo ← 1
    while nodo < ultimo ∧ acept do
        if A[nodo,ultimo] ∧ color(nodo)=color(ultimo) then
            acept ← false
        end if
        nodo ← nodo +1
    end while
    return acept
end procedure
```

Como se puede sospechar, si el número de vértices es grande este algoritmo es inviable, por tanto a efectos prácticos nos tendríamos que conformar con algoritmos heurísticos. Para ello, se pueden implantar ciertas condiciones y así fomentar el descubrimiento anticipado de buenas soluciones en el árbol de búsqueda. Algunas de ellas podrían ser las siguientes:

- Ordenar los vértices en orden decreciente de grado.
- Ordenar los vértices de modo que los que tengan el menor número de colores disponibles, sean coloreados primero.
- Asignar a los vértices aquellos colores que van siendo los más usados.
- Asignar a los vértices aquellos colores que van siendo los más usados en nodos de grados altos.

1.3.2. Algoritmo de conjunto independiente

Este algoritmo no tiene preestablecido el número máximo de colores a utilizar, pero siempre nos proporciona una coloración minimizando el número de colores usados, que vendrán representados por r. Se inicializa X, que contendrá aquellos nodos que aún no han sido coloreados, empezando con el total de los vértices del grafo. A continuación se inicializa Y, en este caso, a medida que se colorean los vértices de un determinado color, contendrá aquellos nodos a los que se le puede asignar dicho color, es decir, que aún no han sido coloreados, ni están adyacentes a un nodo con el color en cuestión.

En resumen, este algoritmo escoge un color, y se lo va asignando a vértices (cuyo método de elección puede variar) hasta que ya no se lo puede asignar a ninguno más sin crear conflicto. A continuación escoge otro color y se sigue el mismo procedimiento, repitiendo este proceso hasta que no quedan nodos por colorear. Aunque este algoritmo es bastante sencillo, suele proporcionar muy buenos resultados, siendo siempre además, muy eficiente.

Algorithm 2 Algoritmo de conjunto independiente

```
X ← vértices del grafo
Y ← X
r ← 1
while X ≠ ∅ do
    while Y ≠ ∅ do
        Elegir i ∈ Y
        Asignar color r a vértice i
        X ← X \ {i}
        Y ← Y \ {i ∧ j : j es adyacente a i}
    end while
    Y ← X
    r ← r + 1
end while
```

1.3.3. Algoritmo de aprendizaje sin comunicación (CFL)

Es un algoritmo de coloración estocástico descentralizado diseñado por Cleith y Clifford [8, 11] que converge a la solución óptima con alta probabilidad. Este método elige el color de cada uno de los nodos ($k \in 1, ..., n$) en cada instante $t \in \{0, 1, ...\}$, donde vendrá representado por $c_k(t)$ que depende del nodo y el instante en el que nos encontremos. La cantidad de colores a usar en este algoritmo está predefinida $\{1, ..., C\}$, y el criterio de elección es aleatorio, el cual sigue la distribución de probabilidad $p_k(t)$, que se inicializa para $t = 0$

como $p_k(0) = (1/C, ..., 1/C)\ \forall k \in \{1, ..., n\}$ y para $t \geq 1$ sigue como se ve a continuación. Además esta distribución depende del valor β a elegir en $(0, 1)$.

$$p_k(t+1) = \begin{cases} \delta_{c_k(t)} & \text{si } c_k(t) \neq c_i(t)\ \forall i \in \Gamma_k \\ (1-\beta)p_k(t) + \frac{\beta}{C-1}\bar{\delta}_{c_k(t)} & \text{si } \exists i \in \Gamma_k : c_k(t) = c_i(t) \end{cases}$$

El vector $\delta_i \in \Re^C$ tiene valor 1 en la coordenada asociada al nodo i-ésimo y 0 en las demás, mientras que $\bar{\delta}_i$ vale 0 en la coordenada del nodo y 1 en los demás.

Se observa que si un nodo tiene color distinto a todos sus vecinos, entonces no cambiará de color en esta iteración. Además, cuanto más pequeño sea el valor de β, más improbable será que el nodo cambie de color, y cuanto más grande, más probable, pudiéndose apreciar más fácilmente cambios globales en todo el grafo.

Un aspecto importante de este algoritmo es el siguiente [2]:

- "Con alta probabilidad el algoritmo de coloración converge a una solución factible en tiempo exponencial. Es decir, para cualquier grafo y $\epsilon \in (0, 1)$, el número de veces m que el grafo debe ser recorrido para obtener una solución factible, es obtenido con probabilidad $1 - \epsilon$ en orden menor que"

$$nexp(nlog(\gamma^{-}1))log(\epsilon^{-}1),$$

Podemos minimizar esta expresión encontrando el β que maximice

$$\gamma(\beta) = (1-\beta)^2(\frac{\beta}{C-1})^{2c+1} \Longrightarrow \beta^* = (2C+1)/(2C+2),$$

obteniendo

$$log(\gamma^{-1}(\beta^*)) = 2log(2C+2) + (2C+1)log\left(\frac{(C-1)(2C+2)}{2C+1}\right)$$

No obstante, esto no nos sugiere que β^* sea el mejor valor para el algoritmo, solo lo es para la cota de la complejidad teórica.

1.3.4. Algoritmo de enumeración combinado con el algoritmo DSatur

Este algoritmo en esencia funciona igual al algoritmo de enumeración mencionado anteriormente, pero como vimos se pueden elegir algunas heurísticas para mejorar la eficiencia del mismo. Pues bien, ya Kubale y Jackowski [6] discuten cuál de esas heurísticas es la más adecuada y proporciona mejores resultados

empíricamente. Finalmente concluyen que lo es la segunda mencionada: ordenar los vértices de modo que los que tengan el menor número de colores disponibles, sean coloreados primero.

Este orden debe ser actualizado en cada paso, ya que depende de la distribución de colores del grafo hasta el momento, a diferencia de otras heurísticas en las que el orden puede ser definido desde el principio y mantenerse fijo hasta el final. Se dice que está combinado con el algoritmo DSatur porque justamente utiliza esta misma heurística.

Si se dispone del tiempo suficiente este es un algoritmo exacto, es decir, que nos garantizará la solución óptima. Sin embargo no siempre nos encontraremos ante este escenario, pero aunque el tiempo esté limitado, nos proporcionará la mejor solución encontrada hasta el momento, proporcionando buenos resultados en comparación con otros algoritmos heurísticos. Tras una búsqueda bibliográfica, este es uno de los dos algoritmos más esperanzadores y que mejores resultados podría proporcionar tras observar recomendaciones y comparativas. En concreto es la mejor opción para aquellos grafos que no son excesivamente densos.

1.3.5. Algoritmo híbrido evolutivo (HEA)

Este es un algoritmo híbrido evolutivo creado por Galinier y Hao [3], donde se combinan los algoritmos DSatur y TabuCol. Ambos son algoritmos heurísticos que ya proporcionan buenos resultados, HEA al igual que TabuCol opera en el espacio de soluciones no factibles utilizando una función objetivo que minimiza el número de conflictos.

El algoritmo comienza creando una población inicial de soluciones candidatas. Cada solución se forma utilizando una versión modificada del algoritmo DSatur, donde el número de colores usados está fijado desde el principio, añadiendo ciertas condiciones para proporcionar diversidad. A continuación se intenta mejorar cada una buscando soluciones alternativas locales, y se sigue con un procedimiento evolutivo. Se escogen dos soluciones aleatoriamente y se genera una solución "hijo"proveniente de ambas, se intenta mejorar nuevamente y se sustituye por la peor de las soluciones "padre".

Este es el otro algoritmo que hemos concluido tras la búsqueda bibliográfica, que nos resuelve con mejores resultados los problemas de coloración. En este caso, es el mejor algoritmo para resolver grafos muy grandes o casos difíciles, comportándose muy adecuadamente ante estos escenarios. No obstante, es un algoritmo donde se usan conceptos nuevos como búsqueda local, recombinación, mutación, presión evolutiva, etc. En este trabajo, no detallaremos su esquema para no extendernos.

2

Presentación del problema real

2.1. Descripción del problema

Las redes de comunicaciones móviles, se caracterizan por tener una estructura celular que permite reutilizar los limitados recursos radio, para garantizar la calidad y capacidad del servicio a los usuarios que acceden a la red. Uno de estos recursos es el Physical Cell Id (PCI), el identificador de célula en la capa física de la red LTE (Long Term Evolution), que tiene como objetivo identificar las células a las que se conecta un terminal. Pero el número de PCIs es finito, por diseño está limitado a 504.

La comunicación entre la red y los terminales se realiza mediante una interfaz radio no guiada con estructura celular, en la que intervienen varios factores en la formación y dimensión de sus células. Entre estos factores se encuentran la potencia de emisión, el diagrama de radiación de las antenas, su orientación e inclinación, permitiendo todos ellos cierto grado de control. Sin embargo, las múltiples reflexiones y refracciones de la señal en el medio de propagación, así como la orografía del terreno, pueden degradar la homogeneidad de la estructura celular, generando sobrealcances y múltiples solapes entre células distantes aunque no sean geométricamente colindantes entre sí. Como consecuencia, se producen dos eventos no deseados: colisión y confusión de PCIs.

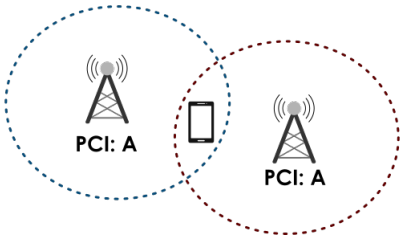


Figura 2.1. Colisión.

Se produce una colisión cuando dos células colindantes tienen el mismo PCI, como en la Figura 2.1.

Se produce confusión cuando una célula tiene dos colindantes con el mismo PCI, como en la Figura 2.2.

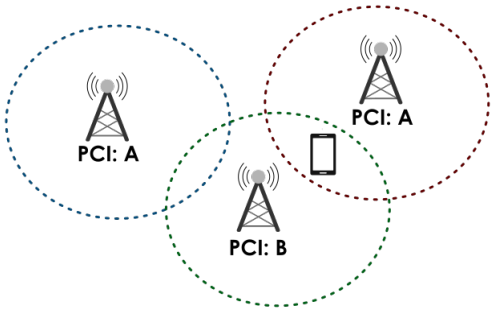


Figura 2.2. Confusión.

Si se da alguno de estos dos conflictos, la eficiencia de la red disminuye cuando los terminales se desplazan por la misma haciendo traspasos entre unas células y otras (en adelante Handover o HO). Si se da alguno de estos casos, la llamada o sesión de datos pierde continuidad y se producen errores de conexión.

2.2. Formulación del problema

Para resolver los conflictos de colisión y confusión de PCIs, utilizaremos el problema de coloración de grafos, donde los vértices representarán las células, los colores los números de sus PCIs y las aristas las relaciones de colindancias, que serán obtenidas como se dijo anteriormente. No se trata exactamente de un problema de coloración, ya que aunque las colisiones sí se puedan minimizar siguiendo este procedimiento, también tenemos que minimizar las confusiones.

Para reducir las colisiones, tendremos que reducir el número de vértices vecinos con el mismo color, pero para reducir las confusiones, tendremos que reducir el número de vértices que tienen el mismo color a los vecinos de los vecinos. Por tanto lo primero que haremos, será adaptar los grafos para que los vecinos de los vecinos de un vértice también sean vecinos entre sí.

2.2.1. Adaptación del grafo

- Sea $G = (V, E)$ un grafo no dirigido, e $i \in V$, los nodos vecinos a i vendrán dados por $N_i = \Gamma_i = \{i_1, ..., i_k\}$, por otro lado los vértices adyacentes a

estos exceptuando i serán los vecinos de los vecinos, $N_i^2 = \bigcup_{j=1}^{k} \Gamma_{i_k} \setminus \{i\}$. Por ello los nodos que deberán llevar distinto color a i vendrán dados por: $M_i = N_i \cup N_i^2$, y el grafo obtenido por tanto será $G = (V, E^*)$, donde E^* incluye las nuevas conexiones entre nodos.

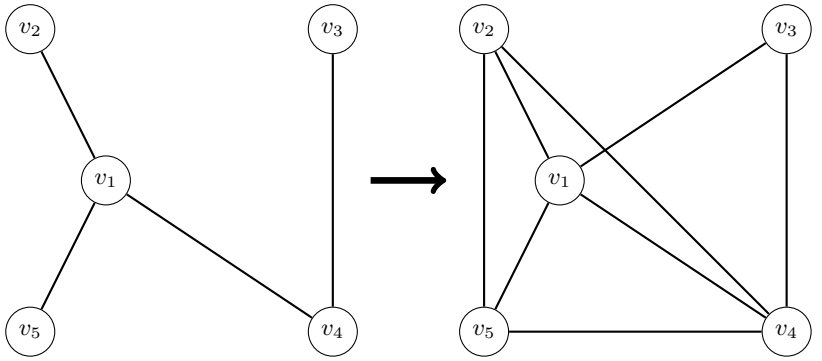


Figura 2.3. Adaptación de grafo no dirigido.

- Sea $G = (V, A)$ un grafo dirigido, e $i \in V$, los vecinos de i vendrán dados por $N_i = \Gamma_i^+ \cup \Gamma_i^-$, que son los nodos que deberán tener color distinto a i para evitar la colisión. Sin embargo, los nodos que deberán llevar diferente color a i para evitar la confusión vendrán dados por: $N_i^2 = \bigcup_{j \in \Gamma_i^-} \Gamma_j^+ \setminus \{i\}$. Finalmente para evitar ambos casos, los nodos que no podrán compartir el mismo color que i serán: $M_i = N_i \cup N_i^2$.

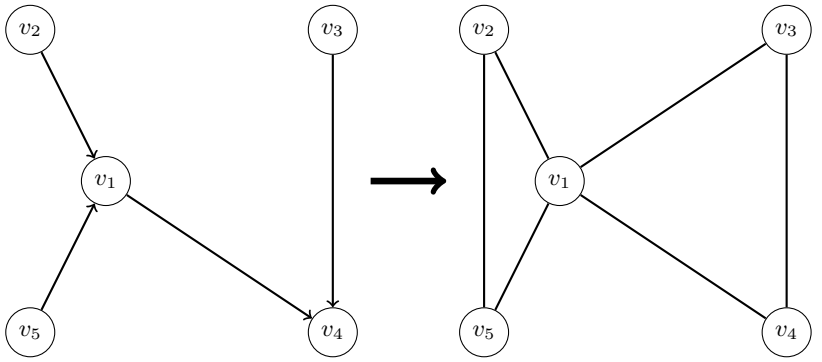


Figura 2.4. Adaptación de grafo dirigido.

Nótese que el grafo final lo podemos tomar como no dirigido, ya que en todos los casos que dos nodos no deban llevar el mismo color será recíproco tanto por la colisión como por la confusión.

3

Resolución del caso real

3.1. Datos

```
Celula,PCI,Celula_Colindante,PCI_Colindante
100393,405,101789,311
100393,405,100278,388
100393,405,101970,273
100393,405,101586,400
100393,405,102195,59
100393,405,100392,407
100393,405,100974,483
100393,405,100394,406
100393,405,100004,240
100393,405,101402,278
100393,405,100323,322
100393,405,101587,399
100393,405,102192,403
100393,405,100466,70
100393,405,101588,368
100393,405,101585,401
100393,405,100479,273
100393,405,101236,37
100393,405,101615,348
100394,406,100466,70
100394,406,100392,407
100394,406,100278,388
100394,406,101402,278
100394,406,101403,431
```

Figura 3.1. Fragmentos de datos cifrados de la red de partida.

Los datos de partida consisten en un grafo dirigido perteneciente a un clúster de la red móvil de Telefónica en Madrid, obtenidos mediante la firma de un Acuerdo de Confidencialidad. Para respetar las condiciones acordadas en este acuerdo, se han cifrado los datos de las células y de sus PCIs con el objeto de que no puedan ser identificados. Los vértices corresponden con las células del clúster, y las aristas con las relaciones de colindancias, resultando un grafo equivalente y válido para este Trabajo de Fin de Grado, pero sin posibilidad de que pueda asociarse a la red real de la que se parte. Además, las soluciones que se obtengan, podrán aplicarse a la red original de Telefónica. En la Figura 3.1 se puede ver un fragmento de los datos cifrados que se van a utilizar (tanto el grafo como la asignación de PCIs actual).

Por otro lado este grafo corresponde solo a la localidad de Madrid, siendo un subgrafo de otro aún mayor, este hecho implica que algunos nodos situados en los extremos realmente están conectados a otros nodos exteriores a nuestro subgrafo. Estos nodos serán reconocidos como aquellos que no tienen sucesores. o lo que es lo mismo, que en ningún caso son predecesores (dentro de nuestro subgrafo).

3.2. Condiciones

Se pretende desarrollar un modelo matemático de optimización, que planifique adecuadamente los PCIs en una red, minimizando las colisiones y las confusiones, y habrá que tener en cuenta las siguientes restricciones:

- El límite de PCIs es 504.
- Se probarán valores menores que 504 con el objeto de dejar reservados algunos PCIs para ajustes manuales.
- No deben ocurrir colisiones ni confusiones.
- Debido a que algunos nodos están conectados a otros externos fuera de nuestro grafo, dado que la red completa no ha sido facilitada, se añadirá la restricción de que estos nodos no podrán sufrir un cambio de PCI.

3.3. Preprocesamiento de datos

En esta sección se describe el funcionamiento de una serie de scripts, cuya finalidad es la adaptación de los datos de acuerdo a las restricciones (previa a la aplicación de los algoritmos). Han sido realizados con Python 3 debido a la facilidad de implementar los mismos en este lenguaje de alto nivel.

3.3.1. 1ª Fase - Gestión de confusiones

Como ya se ha explicado anteriormente, debemos adaptar el grafo añadiendo nuevas aristas para evitar las confusiones como se ha visto en la Figura 2.4, además este nuevo grafo será no dirigido, ya que el conflicto que se pueda crear entre dos nodos es siempre recíproco, tanto con las colisiones como con las confusiones. Teniendo esto en cuenta cambiamos el formato del grafo, de forma que reducimos a la mitad el número de aristas siendo ahora bidireccionales.

Además, los scripts que se ejecutarán posteriormente con estos datos asumen una ordenación y un formato concreto del grafo que se especificará a continuación para funcionar correctamente.

- Debe haber al principio una línea que empiece por: "p edge num_{nodos} $num_{aristas}$".
- A continuación una arista en cada línea, escritas de la siguiente forma: "e $nodo_1$ $nodo_2$" donde $nodo_1 > nodo_2$.
- Las aristas vienen ordenadas en función de $nodo_1$, de menor a mayor, y se asume que los nodos están enumerados desde 1 hasta num_{nodos}.
- Las líneas que empiecen por c se emplearán a modo de comentarios, y serán ignoradas.

```
c Grafo tipo redes de comunicación
p edge 1431 117011
e 2 1
e 3 1
e 3 2
e 5 1
e 5 2
e 5 3
e 5 4
e 6 4
e 6 5
e 17 16
e 18 16
e 18 17
e 20 19
e 21 19
e 21 20
e 25 1
e 25 2
e 25 3
e 25 4
e 25 5
e 25 6
e 26 1
e 26 2
```

Figura 3.2. Fragmento del grafo tras aplicar la 1ª Fase.

Llegados a este punto, se pueden aplicar los algoritmos para resolver el problema de coloración sobre este grafo, sin tener en cuenta aquellos nodos cuyo color se debe mantener fijo, y comparar las soluciones con las utilizadas por Telefónica. Además las nuevas asignaciones aplicadas a cada nodo son guardadas para poder obtener la solución real tras aplicar los algoritmos.

3.3.2. 2ª Fase - Fijación de nodos

Como se ha mencionado anteriormente, hay algunos nodos cuyo color se debe mantener fijo. Para solventar este problema se eligió efectuar una 2ª Fase del preprocesado del grafo, de forma que añadir la restricción dentro de cada uno de los algoritmos será equivalente a resolver este nuevo grafo como si de un problema de coloración se tratase.

Para conseguir este cometido, se parte del grafo obtenido tras la 1ª Fase y se sigue la siguiente estrategia, que se puede seguir paralelamente en la Figura 3.4 donde los PCIs vendrán representados por colores.

1. Encontramos aquellos nodos cuyo color no debe cambiar, en adelante los llamaremos nodos fijos, para guardarlos ordenados en un nuevo fichero así como sus respectivos PCIs.

2. Detectaremos los nodos fijos que deben llevar el mismo PCI, guardándolos también en un fichero ordenados.
3. Unificamos cada conjunto de nodos fijos que tengan el mismo PCI en uno solo. En nuestro caso elegimos uno de los nodos de cada grupo, y sustituimos en todas las aristas este nodo sobre el resto de los que están dentro de su conjunto (es decir, se conectan las aristas con el primero, y se desconectan las demás), generándose así en algunos casos aristas repetidas. Las réplicas que aparecen son consecuencia de que dos nodos con el mismo PCI tuvieran vecinos en común, en cuyo caso se eliminan la réplicas.

   Por otro lado se sabe que con este procedimiento no se generan aristas cuyo nodo coincide en ambas coordenadas (lazo), porque para ello tendría que haberse dado el caso anteriormente de que dos nodos fijos estén conectados entre sí. Pero esto no ocurre porque los nodos fijos son aquellos que solo aparecían en la segunda coordenada de la arista dirigida en el caso inicial, y ni aún generando nuevas aristas para tener en cuenta las confusiones se da el caso. Porque cada nodo se conecta con el predecesor de su sucesor, y en nuestro caso un nodo fijo solo podrían llegar a ser el sucesor del predecesor del otro por lo explicado anteriormente.
4. Finalmente tenemos una serie de nodos fijos, todos ellos con PCIs distintos. Ahora se unen todas las combinaciones de dos nodos posibles sobre este clúster, quedando todos unidos con todos, generando así $\binom{num_{nodosfijos}}{2}$ nuevas aristas (un clique). No se generan aristas repetidas por la misma razón que no se generan lazos en el punto anterior.

En esta fase, es esencial nuevamente el formato en el que se guardan los datos, ya que los scripts lo asumen para poder funcionar correctamente, así como los nodos elegidos en cada clúster. Hemos decidido realizarlo de esta forma para poder tratar los datos con comodidad al estar ordenados en todo momento, aprovechando esta característica para una mayor sencillez en los scripts. Nótese que el nuevo grafo, es obtenido finalmente en el mismo formato que al finalizar la 1ª Fase y puede ser observado un fragmento en la Figura 3.3.

Se ha decidido añadir aristas entre los nodos fijos que tienen PCIs distintos para que en la solución no puedan compartir el mismo color y así posteriormente se puedan intercambiar los PCIs y devolverles los que tenían inicialmente. Pero aquellos nodos que comparten el mismo son un obstáculo, porque la solución proporcionada por los algoritmos podrían otorgarles distintos PCIs, impidiéndonos así poder hacer el intercambio posterior. Es por ello que los "unificamos" previamente.

Una vez acabada esta 2ª Fase se pueden aplicar algoritmos de coloración sobre el grafo, donde ahora las soluciones cumplen la restricción de que los

```
c Grafo tipo redes de comunicación - nodos fijados
p edge 1431 167274
e 2 1
e 3 1
e 3 2
e 5 1
e 5 2
e 5 3
e 5 4
e 6 4
e 6 5
e 8 7
e 9 7
e 9 8
e 10 7
e 10 8
e 10 9
e 11 7
e 11 8
e 11 9
e 11 10
e 12 7
e 12 8
e 12 9
e 12 10
```

Figura 3.3. Fragmento del grafo obtenido tras aplicar la 2ª Fase.

nodos fijos mantengan el mismo color (siempre y cuando se adapte la solución posteriormente, que correspondería a la fase de postprocesamiento).

3.4. Aplicación de algoritmos

Una vez llegados a este punto solo queda aplicar los algoritmos y contrastar los resultados, pero también es importante saber de donde partimos para tomarlo de referencia respecto a donde llegamos, sabiendo así si hemos mejorado y cuánto. Para ello elaboramos algunos scripts que cuentan y enumeran algunos datos importantes a tener en cuenta en nuestra comparativa de antes y después, que se muestran a continuación:

- Grafo 1
  - Partimos de un grafo de 1.431 nodos y 29.509 arcos.
  - Se utilizan 481 PCIs distintos y se dan 3 colisiones y 111 confusiones.
  - Tras aplicar la 1ª Fase, pasamos a tener 117.011 aristas.
  - En total hay 473 nodos fijos que no deben cambiar su PCI, entre los cuales utilizan un total de 318 PCIs distintos, cota inferior del número de PCIs que podemos obtener en nuestros resultados.
  - Tras aplicar la 2ª Fase pasamos finalmente a tener un total de 167.274 aristas.

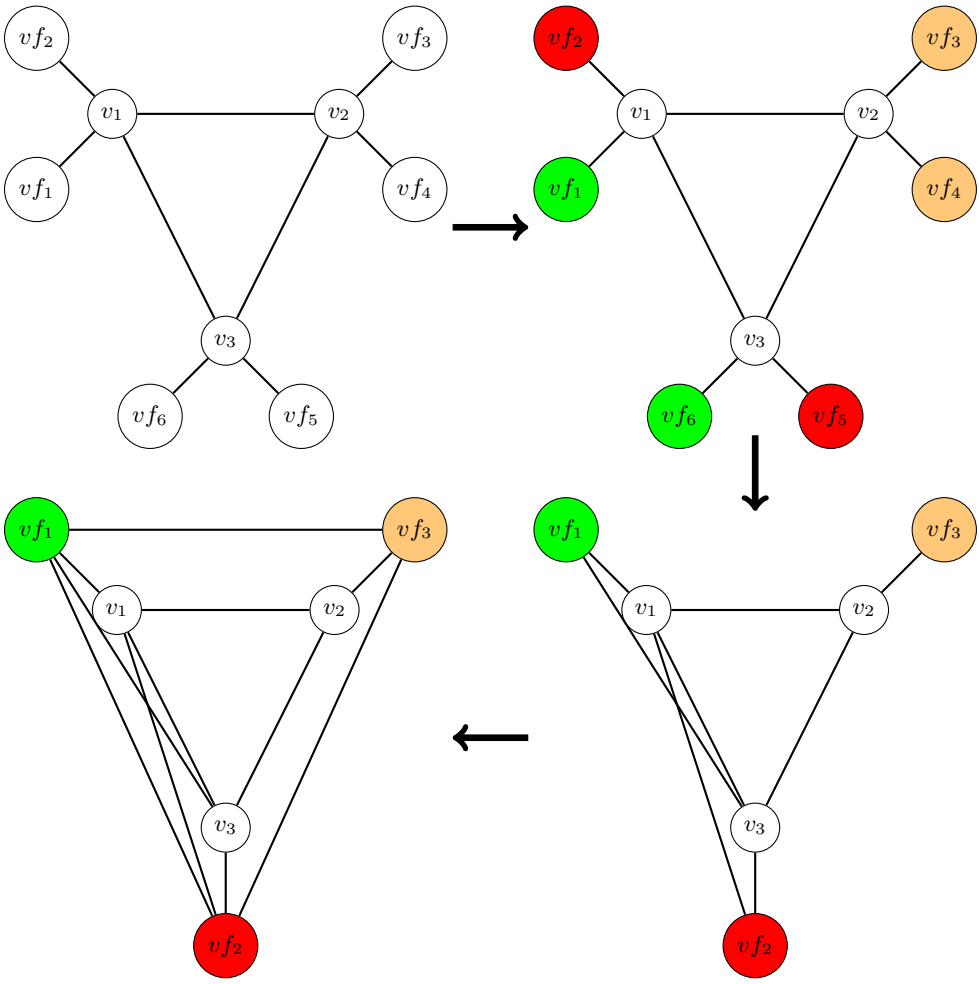


Figura 3.4. Pasos de la modificación del grafo durante la 2ª Fase

En la red LTE (comentada anteriormente) hay una funcionalidad que se llama ANR (Automatic Neighbor Relation), que consiste en la definición automática de nuevas colindancias sin intervención humana, en función de varias situaciones de la red. Debido a los buenos resultados que se obtuvieron con el primer grafo, que se irán mostrando posteriormente, se nos proporcionó uno nuevo sobre el que aplicar los algoritmos. Es más denso porque define nuevas colindancias con células más lejanas, y está ampliado porque incluye células adi-

cionales que esta funcionalidad podría definir como colindantes en determinadas circunstancias.

- Grafo 2
  - Partimos de un grafo de 1.841 nodos y 34.506 aristas dirigidas.
  - Se utilizan 485 PCIs distintos y se dan 47 colisiones y 238 confusiones.
  - Tras aplicar la 1ª Fase, pasamos a tener 165.403 aristas no dirigidas.
  - En total hay 675 nodos fijos que no deben cambiar su PCI, entre los cuales utilizan un total de 375 PCIs distintos, cota inferior del número de PCIs que podemos obtener en nuestros resultados.
  - Tras aplicar la 2ª Fase pasamos finalmente a tener un total de 235.308 aristas no dirigidas.

A continuación se procede a los resultados obtenidos de los algoritmos que han sido aplicados.

3.4.1. Algoritmo de aprendizaje sin comunicación (CFL)

En primer lugar se recurrió al algoritmo estocástico ya explicado, porque como no se han encontrado casos en los que ya se haya probado empíricamente y comparado con otros, estamos ante la incertidumbre de si puede tener potencial resolviendo este tipo de problemas. Ha sido implementado en C++, y se empezó ejecutando el primer grafo sin la restricción de los nodos fijos, es decir sometiendo el grafo solo ante la primera fase de adaptación. Este es el caso más sencillo al que nos enfrentamos, sin embargo los resultados no son demasiado satisfactorios en comparación con los demás algoritmos.

Como ya se ha visto, este algoritmo depende de varios parámetros que deben ser definidos desde el principio: el número de colores a utilizar y β. Debido a que podría no acabar si no existe una coloración factible con el número de colores dado, predefinimos un número de iteraciones máximo (1.000 iteraciones).

A continuación tenemos la intención de buscar una solución con un número de colores dado, e ir disminuyendo este número hasta que no encontremos una mejor. Pero nos encontramos ante el obstáculo de que el número mínimo de colores que necesita para encontrar una solución es demasiado elevado (>400). Por otro lado si disminuimos este número y tenemos en cuenta la cantidad de nodos implicados en algún conflicto, podríamos obtener soluciones mucho mejores. Porque si la solución utiliza x colores y hay y nodos implicados en conflictos, tenemos la garantía de que si le cambiamos el color a $y-1$ de los nodos, asignándole un nuevo color no usado, desaparecen los conflictos y obtendremos una solución factible. Por esta razón, variamos el número de colores a utilizar por el algoritmo sobre también, un intervalo de números más pequeños, llegando a un número óptimo en torno a 200.

A continuación variamos el parámetro β, y rápidamente nos damos cuenta que cuanto menor es su valor, mejores soluciones obtenemos independientemente del número de colores a utilizar por el algoritmo. Es por ello que nos decidimos a darle un valor lo más pequeño posible. Una vez ya elegidos los parámetros, ejecutamos el algoritmo e incrementamos el número de iteraciones a 10.000, obteniendo una solución que utiliza tan solo 230 colores (200 colores de entrada + 30 nodos a los que se les asigna un nuevo color ya que 31 nodos están implicados en conflictos). Nótese que si el número de conflictos es 0, hay 0 colisiones y 0 confusiones.

El tiempo de ejecución de este algoritmo es relativamente alto, en un ordenador con procesador de 2,5 GHz y 8 GB de memoria RAM, la última ejecución llevó en torno a 15 minutos. Además se intuye que se pueden obtener soluciones mejores, ya que añadir 30 colores a los 200 ya predefinidos, no es más que una cota superior de los colores a utilizar, sin tener en cuenta la distribución de aristas sobre estos nodos. Antes de empezar a implementar algoritmos posteriormente, sobre cómo reducir esos 30 nuevos colores teniendo en cuenta las colindancias de los nodos implicados, o ejecutar este algoritmo sobre los otros grafos mucho más complejos, procedemos a utilizar los algoritmos más recomendados tras nuestra búsqueda bibliográfica.

3.4.2. Algoritmo híbrido evolutivo (HEA)

Debido a que no aún no sabemos como puede reaccionar un algoritmo exacto ante nuestros grafos debido a su complejidad, utilizamos primero HEA, ya que es el algoritmo que hemos concluido que funciona mejor ante situaciones adversas. Está programado en C++, y en este caso hemos obtenido el código de [9], que requiere un formato determinado del grafo tanto de entrada como de salida como ya se ha especificado anteriormente. Probaremos el algoritmo ante 4 casos distintos, en todos ellos los resultados son soluciones ya factibles donde se reduce el número de colisiones y confusiones a 0. A continuación se muestran el número de PCIs utilizados en cada caso.

1. Grafo 1 donde no hay restricción de los nodos que deben mantener su color: 146 PCIs.
2. Grafo 1 donde sí hay restricción de los nodos que deben mantener su color: 318 PCIs.
3. Grafo 2 donde no hay restricción de los nodos que deben mantener su color: 176 PCIs.
4. Grafo 2 donde sí hay restricción de los nodos que deben mantener su color: 375 PCIs.

En los casos 2 y 4 tenemos la garantía de que es la solución óptima, ya que usa el mismo número de PCIs que usan en total aquellos nodos que no deben

cambiar su color. En la Figura 3.5 se contempla un fragmento de la solución obtenida de este algoritmo sin haber aplicado aún el postprocesado ante el 2º caso, donde el primer número es la cantidad de nodos (que vienen definidos por su posición) y continuación se muestran los PCIs asociados a cada uno de ellos.

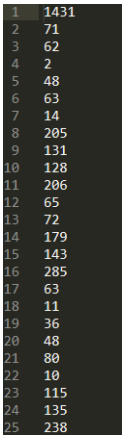


Figura 3.5. Solución obtenida por el algoritmo sin aplicar el postprocesado.

3.4.3. Algoritmo de enumeración combinado con el algoritmo DSatur

Ya que hemos visto lo bien que se comporta HEA, hemos decido aplicar por último este algoritmo enumerativo, con el riesgo de que no nos resuelva el problema en un tiempo razonable. Al igual que los demás algoritmos de coloración, está programado en C++ y ha sido obtenido de [9]. Las soluciones se muestran a continuación (todas las soluciones nuevamente son factibles con 0 colisiones y confusiones):

1. Grafo 1 donde no hay restricción de los nodos que deben mantener su color: 146 PCIs.
2. Grafo 1 donde sí hay restricción de los nodos que deben mantener su color: 318 PCIs.
3. Grafo 2 donde no hay restricción de los nodos que deben mantener su color: 176 PCIs.
4. Grafo 2 donde sí hay restricción de los nodos que deben mantener su color: 375 PCIs.

Como se observa, los resultados obtenidos son igual de buenos que los de HEA, pero sin embargo ha llevado un mayor esfuerzo computacional. No

obstante los dos algoritmos lo resuelven en un tiempo razonable (menos de diez segundos).

3.5. Postprocesamiento de datos

Como acabamos de ver, tras finalizar la segunda fase obtuvimos un grafo no dirigido, ya listo sobre el que se aplicaron los algoritmos de coloración. Pero una vez los algoritmos han sido aplicados, y las soluciones obtenidas, aún queda por adaptar la solución para que los nodos fijos mantengan sus PCIs iniciales.

A partir de ahora, partimos de una solución proporcionada por algún algoritmo sobre un grafo adaptado similar al obtenido de la Figura 3.4. Lo primero que hacemos es desunir los nodos fijos, mostrando nuevamente aquellos nodos que habían desaparecido y otorgándoles a cada uno el mismo PCI obtenido por la solución que aquel nodo con el que se había unido previamente, ya que ambos deben compartir el mismo PCI. Alcanzando a modo de ejemplo un grafo como el de la izquierda de la Figura 3.6.

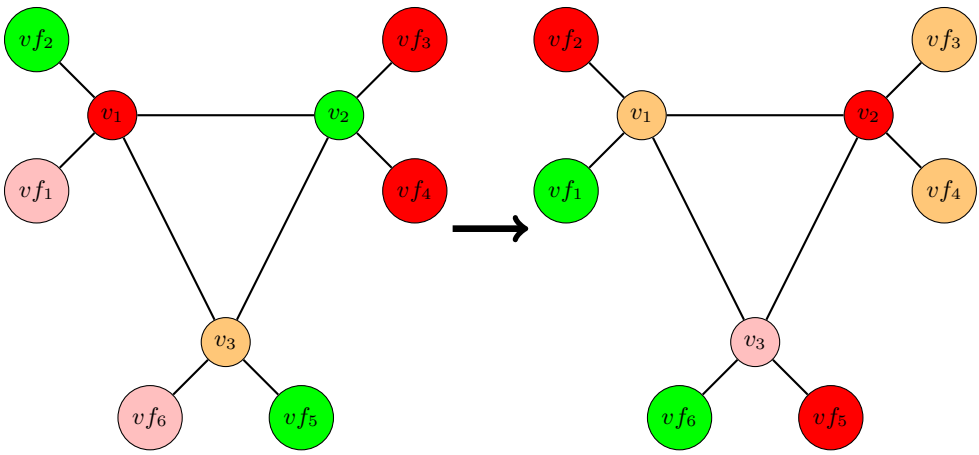


Figura 3.6. Redistribución de los colores en el postprocesado, intercambiados de la siguiente manera: {verde → rojo, rojo → naranja, roja → verde, naranja → rosa}.

A continuación, se comprueba en la solución aquellos PCIs obtenidos en los nodos fijos, y como lo más probable es que no coincidan con el que llevaban inicialmente, se efectúa un intercambio sobre los mismos para que cada nodo fijo vuelva a llevar el PCI inicial. Pero estos cambios se efectúan sobre el grafo

completo, teniendo en cuenta que si este PCI ya está siendo utilizado, a sus respectivos nodos les otorgamos el que estamos dejando de utilizar para así evitar nuevos posibles conflictos. Donde tenemos la garantía de que podremos asignarles a todos los nodos fijos sus PCIs correspondientes. Y no existirá conflicto con los vecinos de un nodo no elegido sobre el que hacer la unificación, ya que al elegido se le añadieron los vecinos de los demás.

A modo de ejemplo, véase la Figura 3.6, donde a la izquierda se ve una solución obtenida del grafo, y a la derecha se intercambian los colores para que se cumpla la restricción de que los nodos fijos mantengan el mismo color que en el grafo de la Figura 3.4.

La solución que se obtiene es siempre equivalente e igual de óptima que la anterior, ya que siempre usará el mismo número de colores, además obsérvese que la solución que obtengamos nunca usará menos colores, que los usados en total por los nodos fijos. Por último, un fragmento de una solución resultante tras aplicar esta fase se puede ver en la Figura 3.7 (concretamente sobre la solución de la Figura 3.5).

```
Celula,color,1431 celulas,318 PCIs
100004,127
100005,37
100006,18
100007,72
100008,285
100009,4
100013,336
100014,338
100015,337
100016,104
100017,103
100018,102
100022,206
100023,204
100024,202
100028,212
100029,146
100030,28
100040,372
100041,146
100042,100
100046,483
100047,82
100048,86
```

Figura 3.7. Fracción de solución de ejemplo obtenida tras aplicar el postprocesado.

4

Conclusiones

En primer lugar, los resultados obtenidos concuerdan con las conclusiones y recomendaciones que se han observado tras la búsqueda bibliográfica. Es decir, los dos algoritmos concluidos como mejores nos han proporcionado muy buenas soluciones, mejorando incluso las distribuciones de PCIs utilizados inicialmente en tiempos muy razonables. Esto permite que exista un mayor número de PCIs disponibles por si son necesitados en determinadas circunstancias, como la habilitación de nuevas células sin crear conflictos.

En el primer grafo, se ha reducido las 3 colisiones y 111 confusiones a 0, que es el principal objetivo de este trabajo, ya que estos son causantes de Handovers fallidos. El número de PCIs utilizados se ha reducido de 481 a 146 sin tener en cuenta aquellos nodos que deben mantener su PCI, y a 318 teniéndolo en cuenta (solución óptima).

En el segundo, se han reducido las 47 colisiones y 238 confusiones a 0 nuevamente. El número de PCIs utilizados se ha reducido de 485 a 176 sin tener en cuenta aquellos nodos que deben mantener su PCI, y a 375 teniéndolo en cuenta (solución óptima).

Tras la realización de este trabajo, se pretende estudiar si la asignación propuesta de PCIs resulta beneficiosa para la red de comunicaciones. Para ello se medirá la bondad de la red antes y después de la reasignación de PCIs. La bondad de red se establece tomando como medidas el nº de Handovers fallidos y el nº de Handovers establecidos diariamente en un ciclo semanal.

Además, dados los buenos resultados se prevé utilizar los algoritmos en otras localidades, para lo cual bastaría con aplicar las distintas fases y los algoritmos a cada uno de los grafos de las mismas. Sobre todo convendría en aquellas con mayor cantidad de estos conflictos, para ayudar a solventar el problema de los Handovers fallidos.

	Inicialmente	**Algoritmo CFL**	**Algoritmo HEA**	**Algoritmo enumerativo DSatur**
Grafo 1				
PCIs utilizados	481	230	146	146
Colisiones	3	0	0	0
Confusiones	111	0	0	0
Grafo 1 - Nodos fijados				
PCIs utilizados	481	-	318 (óptima)	318 (óptima)
Colisiones	3	-	0	0
Confusiones	111	-	0	0
Grafo 2				
PCIs utilizados	485	-	176	176
Colisiones	47	-	0	0
Confusiones	238	-	0	0
Grafo 2 - Nodos fijados				
PCIs utilizados	485	-	375 (óptima)	375 (óptima)
Colisiones	47	-	0	0
Confusiones	238	-	0	0

Tabla 4.1. Resumen resultados finales.

A

Apéndice

Como ya se ha hablado, los scripts elaborados para manipular los grafos antes y después de buscar una coloración, están implementados en Python 3. Mientras que el algoritmo estocástico que resuelve el problema de coloración (al igual que los demás) está implementado en C++.

Las librerías de Python 3 utilizadas para todos los scripts son las que se listan a continuación:

```
import pandas
import numpy as np
```

A.1. Script de preprocesamiento - 1ª Fase

Se lee el archivo "ArchivoULL.csv", que es el grafo proporcionado por Telefónica y se crea "grafonica.txt", el grafo obtenido tras la primera fase, y "asignaciones.txt", que contiene la correspondencia entre los índices de los nodos y su código de célula.

```
1  # Lectura e inicializacion de datos
2  data = pandas.read_csv('./ArchivoULL.csv', header = 0, dtype =
       int)
3
4  coords1 = list(data.Celula)
5  coords2 = list(data.Celula_Colindante)
6  nodes = list(set(coords1 + coords2))
7
8  n_nodes = len(nodes)
9  n_edges = len(coords1)
10
```

```
numbered_nodes = list(enumerate(nodes, 1))
predecessors = [[] for y in range(n_nodes)]
new_coords = np.zeros((n_edges, 2), dtype = int)

# Almacenamiento de las aristas y los predecesores de cada nodo
for i in range(n_edges):
  for j in range(n_nodes):
    if coords1[i] == numbered_nodes[j][1]:
      new_coords[i][0] = numbered_nodes[j][0]
    if coords2[i] == numbered_nodes[j][1]:
      new_coords[i][1] = numbered_nodes[j][0]
  predecessors[new_coords[i][1] - 1].append(new_coords[i][0])

new_coords = list(map(lambda x: list(x), new_coords))

# Generación de las las nuevas aristas (los predecesores de los sucesores pasan
# a ser vecinos también)
for i in range(n_nodes):
  for j in range(len(predecessors[i]) - 1):
    for k in range(j + 1, len(predecessors[i])):
      new_coords.append([predecessors[i][j], predecessors[i][k]])

# Se toma a partir de ahora como grafo no dirigido, y el formato debe ser de
# la forma: primera coordenada mayor que la segunda
for i in range(len(new_coords)):
  if new_coords[i][0] < new_coords[i][1]:
    new_coords[i][0], new_coords[i][1] = new_coords[i][1], new_coords[i][0]

# Se eliminan aristas repetidas y se ordenan de menor a mayor por la primera coordenada
new_coords = list(map(lambda x: list(x), set(map(tuple, new_coords))))
new_coords.sort()

# Escritura en fichero del nuevo grafo ("grafonica.txt") y las correspondencias
# con los valores de las celulas
text = 'c Grafo tipo redes de comunicación\np edge ' + str(n_nodes) + ' ' + str(len(new_coords)) + '\n'
for i in range(len(new_coords)):
```

```
47   text += 'e ' + str(new_coords[i][0]) + ' ' + str(new_coords[
         i][1]) + '\n'
48 file = open('./grafonica.txt', 'w')
49 file.write(text)
50 file.close()
51
52 text = 'Corresponcia entre las células y los nodos tomados (
       enumerados), ' + str(n_nodes) + ' nodos:\n'
53 for i in range(n_nodes):
54   text = text + str(numbered_nodes[i][0]) + ' ' + str(
         numbered_nodes[i][1]) + '\n'
55 file = open('./asignaciones.txt', 'w')
56 file.write(text)
57 file.close()
```

A.2. Script 1 de preprocesamiento - 2ª Fase

Se fijan los nodos a los que no se les puede cambiar el PCI y se almacenan en "nodosfijos.txt", además en "coloresoriginales.csv" se almacena la correspondencia con los colores originales.

```
1  data = pandas.read_csv('./ArchivoULL.csv', header = 0, dtype =
       int)
2
3  coords = []
4  coords.append(list(data.Celula))
5  coords.append(list(data.Celula_Colindante))
6  coords.append(list(data.PCI_Colindante))
7
8  n_directed_edges = len(coords[0])
9
10 fixed_cells = []
11 fixed_cells.append(list(set(coords[1]) - set(coords[0])))
12 fixed_cells[0].sort()
13 colores = []
14 for i in fixed_cells[0]:
15   for j in range(n_directed_edges):
16     if i == coords[1][j]:
17       colores.append(coords[2][j])
18       break
19 fixed_cells.append(colores)
20
21 file_nodes = open('./nodosfijos.txt', 'w')
22 file_colours = open('./coloresoriginales.csv', 'w')
```

```
23
24 file_colours.write('Celula,Color\n')
25
26 used = []
27 for i in range(len(fixed_cells[0])):
28   if fixed_cells[1][i] not in used:
29     used.append(fixed_cells[1][i])
30     file_nodes.write(str(fixed_cells[0][i]))
31     file_colours.write(str(fixed_cells[0][i]) + ',' + str(
           fixed_cells[1][i]) + '\n')
32     for j in range(i + 1, len(fixed_cells[0])):
33       if fixed_cells[1][i] == fixed_cells[1][j]:
34         file_nodes.write(',' + str(fixed_cells[0][j]))
35     file_nodes.write('\n')
36
37 file_nodes.close()
38 file_colours.close()
```

A.3. Script 2 de preprocesamiento - 2ª Fase

Se completa la 2ª fase creando "grafonica2.txt", que contiene el grafo adaptado de acuerdo a las restricciones de nodos fijos.

```
1 # Grafo
2 file = open('grafonica.txt', 'r')
3 graph = file.read()
4 file.close()
5
6 graph = graph.split('\n')
7 description = graph[0]
8 n_nodes = graph[1].split('␣')[2]
9 graph = list(map(lambda x: x.split('␣'), graph[2:-1]))
10
11 # Nodos fijos
12 file = open('nodosfijos.txt' , 'r')
13 fixed_nodes = file.read()
14 file.close()
15
16 fixed_nodes = fixed_nodes.split('\n')[:-1]
17 fixed_nodes = list(map(lambda x: x.split(','), fixed_nodes))
18
19 # Asignaciones
20 file = open('asignaciones.txt' , 'r')
21 assignments = file.read()
22 file.close()
```

```
23
24 assignments = assignments.split('\n')[1:-1]
25 assignments = list(map(lambda x: x.split(' '), assignments))
26 list(map(lambda x: x.reverse(), assignments))
27 dict_assignments = dict(assignments)
28
29 # Unificamos los nodos fijos del mismo color
30 for i in range(len(fixed_nodes)):
31   if len(fixed_nodes[i]) == 1:
32     continue
33   assignment_master = dict_assignments[fixed_nodes[i][0]]
34
35   for t in range(1, len(fixed_nodes[i])):
36     assignment_slave = dict_assignments[fixed_nodes[i][t]]
37     for w in range(len(graph)):
38       for p in range(1, 3):
39         if graph[w][p] != assignment_slave:
40           continue
41         graph[w][p] = assignment_master
42         if p == 1 and int(graph[w][1]) < int(graph[w][2]):
43           graph[w][2], graph[w][1] = graph[w][1], graph[w][2]
44
45 # Unimos todos estos nodos para que no compartan el mismo
     color
46 for i in range(len(fixed_nodes) - 1):
47   for j in range(i + 1, len(fixed_nodes)):
48     graph.append(['e', dict_assignments[fixed_nodes[j][0]],
         dict_assignments[fixed_nodes[i][0]]])
49 graph = sorted(list(map(lambda a: list(a), set(map(lambda a:
     tuple(a), graph)))), key = lambda a: tuple([int(a[1]), int
     (a[2])]))
50
51 # Escritura
52 file = open('./grafonica2.txt', 'w')
53 file.write(description + ' - nodos fijados\n')
54 file.write('p edge ' + n_nodes + ' ' + str(len(graph)) + '\n')
55 for n in graph:
56   file.write(' '.join(n) + '\n')
57 file.close()
```

A.4. Script de postprocesamiento

Se devuelve a las células su PCI inicial, y proporciona la solución final.

```
file = open('solution.txt', 'r')
solution = file.read()
file.close()

file = open('asignaciones.txt', 'r')
assignments = file.read()
file.close()

file = open('nodosfijos.txt', 'r')
fixed_nodes = file.read()
file.close()

data = pandas.read_csv('coloresoriginales.csv', header = 0,
    dtype = str)

solution = solution.split('\n')
n_nodes = solution[0]
solution = solution[1:-1]

assignments = assignments.split('\n')[1:-1]
assignments = list(map(lambda x: x.split(' '), assignments))

fixed_nodes = fixed_nodes.split('\n')[:-1]
fixed_nodes = list(map(lambda x: x.split(','), fixed_nodes))

fixed_cells = list(data.Celula)
original_colours = list(data.Color)

list(map(lambda x: x.reverse(), assignments))
for i in range(len(assignments)):
  assignments[i][1] = solution[i]
dict_colores = dict(assignments)

for i in range(len(fixed_nodes)):
  if len(fixed_nodes[i]) != 1:
    for j in range(1, len(fixed_nodes[i])):
      dict_colores[fixed_nodes[i][j]] = dict_colores[
          fixed_nodes[i][0]]

new_colors = []
for i in range(len(original_colours)):
  new_colors.append(dict_colores[fixed_cells[i]])

for i in range(len(original_colours)):
  for j in dict_colores:
    if dict_colores[j] == original_colours[i]:
```

```
45        dict_colores[j] = new_colors[i]
46      elif dict_colores[j] == new_colors[i]:
47        dict_colores[j] = original_colours[i]
48    for j in range(i + 1, len(original_colours)):
49      if original_colours[i] == new_colors[j]:
50        new_colors[j] = new_colors[i]
51
52 solution = list(map(lambda x: int(x), solution))
53
54 file = open('solucion_final.csv', 'w')
55 file.write('Celula,color,' + n_nodes + ' celulas,' + str(max(
       solution) + 1) + ' PCIs\n')
56 for i in dict_colores:
57   file.write(i + ',' + dict_colores[i] + '\n')
58 file.close()
```

A.5. Script contador de colisiones y confusiones

Se toma el grafo de "ArchivoULL.csv" con la solución inicial proporcionada por Telefónica, y se crea "colisiones.csv" y "confusiones.csv"donde se cuentan y especifican las mismas.

```
1 # Lectura de datos
2 data = pandas.read_csv('./ArchivoULL.csv', header = 0, dtype =
      int)
3 coords = []
4 coords.append(list(data.Celula))
5 coords.append(list(data.Celula_Colindante))
6 nodes = list(set(coords[0] + coords[1]))
7 n_nodes = len(nodes)
8 n_edges = len(coords[0])
9
10 # Se comprueba si hay aristas repetidas.
11 coords = np.asarray(coords)
12 coords = coords.transpose()
13 coords = coords.tolist()
14 aux_cords = []
15 for sublist in coords:
16   if sublist not in aux_cords:
17     aux_cords.append(sublist)
18 if (n_edges != len(aux_cords)):
19   print('Hay ', n_edges - len(aux_cords), ' aristas repetidas.
       ')
20 coords = np.asarray(aux_cords)
```

```
21 coords = coords.transpose()
22 coords = coords.tolist()
23
24 # Incorporamos los PCI de los vertices a la lista de listas.
25 coords.append(list(data.PCI))
26 coords.append(list(data.PCI_Colindante))
27 colours = list(set(coords[2] + coords[3]))
28
29 # Contamos el numero de colisiones y las incorporamos a un
       fichero (ordenándolas y eliminando las repetidas por ser
       dirigido).
30 collisions = []
31 for i in range(n_edges):
32   if coords[2][i] == coords[3][i]:
33     if coords[0][i] > coords[1][i]:
34       collisions.append([coords[0][i], coords[1][i], coords
             [2][i]])
35     else:
36       collisions.append([coords[1][i], coords[0][i], coords
             [2][i]])
37 collisions = list(map(lambda x: list(x), set(map(tuple,
       collisions))))
38 collisions.sort()
39
40 text = 'Celula,Celula_Colindante,PCI,Numero colisiones total: '
       + str(len(collisions)) + ',Numero colores usados: ' +
       str(len(colours)) + '\n'
41 for i in range(len(collisions)):
42   text += str(collisions[i][0]) + ',' + str(collisions[i][1])
         + ',' + str(collisions[i][2]) + '\n'
43 file = open('./colisiones.csv', 'w')
44 file.write(text)
45 file.close()
46
47 # Contamos el numero de confusiones y las incorporamos a un
       fichero (ordenadas) y eliminamos las confusiones repetidas
        por distintas celulas intermedias.
48 numbered_list = list(enumerate(nodes, 1))
49 new_coords = np.zeros((n_edges, 2), dtype =int)
50 predecessor = [[] for y in range(n_nodes)]
51 for i in range(n_edges):
52   for j in range(n_nodes):
53     if coords[1][i] == numbered_list[j][1]:
54       new_coords[i][1] = numbered_list[j][0]
55       break
```

```
56    predecessor[new_coords[i][1] - 1].append([coords[0][i],
          coords[2][i]])
57
58  confusions_total = []
59  for i in range(n_nodes):
60    for j in range(len(predecessor[i]) - 1):
61      for k in range(j + 1, len(predecessor[i])):
62        if predecessor[i][j][1] == predecessor[i][k][1]:
63          if predecessor[i][j][0] > predecessor[i][k][0]:
64            vector = [predecessor[i][j][0], predecessor[i][k
                  ][0], predecessor[i][j][1], numbered_list[i][1]]
65          else:
66            vector = [predecessor[i][k][0], predecessor[i][j
                  ][0], predecessor[i][j][1], numbered_list[i][1]]
67          confusions_total.append(vector)
68  confusions_total.sort()
69
70  confusions = [confusions_total[0]]
71  for i in range(1, len(confusions_total)):
72    if confusions_total[i - 1][0] != confusions_total[i][0] or
          confusions_total[i - 1][1] != confusions_total[i][1]:
73      confusions.append(confusions_total[i])
74
75  file = open ('./confusiones.csv', 'w')
76  text = 'Celula,Celula2,PCI,Celula_intermedia,Numero␣
      confusiones␣total:␣' + str(len(confusions)) + ',Numero␣
      colores␣usados:␣' + str(len(colours)) +  '\n'
77  for i in range(len(confusions)):
78      text += str(confusions[i][0]) + ',' + str(confusions[i
          ][1]) + ',' + str(confusions[i][2]) + ',' + str(
          confusions[i][3]) + '\n'
79  file.write(text)
80  file.close()
```

A.6. Script verificador

Se comprueba si la solución final es factible y no existe ningún conflicto (0 colisiones y confusiones).

```
1  file = open('solution.txt', 'r')
2  solution = file.read().split('\n')
3  file.close()
4  file = open('grafonica.txt', 'r')
5  assignments = list(map(lambda x: x.split('␣'), file.read().
       split('\n')))
```

```
file.close()

k = 0
for i in range(2, len(assignments) - 1):
  if solution[int(assignments[i][1])] == solution[int(
      assignments[i][2])]:
    k = k + 1
print(k, 'conflictos')
```

A.7. Algoritmo CFL

```
#include "stdafx.h"
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
using namespace std;

/* Dado un vector de probabilidades ponderadas, selecciona uno
    aleatoriamente */
int weightedRandomSample(double* weights, int count) {
  int result = -1;
  double max_random_value = 0.0, current_random_value;

  while (result == -1) {
    for (int index = 0; index < count; index++) {
      current_random_value = pow((double)rand() / RAND_MAX,
          1.0 / weights[index]);

      if (current_random_value > max_random_value) {
        max_random_value = current_random_value;
        result = index;
      }
    }
  }
  return result;
}


int main() {
```

```
32  const int num_colo = 115; /* Debe ser entero para ser usado
        como índice de array */
33  const float numero_colo = 115;  /* Debe ser flotante para
        poder operar con él */
34  string linea;
35  string p[4];
36  string palabra;
37  int i = 0;
38  int j = 0;
39  int k = 0;
40  int numv;
41  int numa;
42  int* nodos;
43  int* aristas[2];
44  float beta = 0; // (2*numero_colo + 1)/(2*numero_colo + 2);
45  int num_maxiter = 10000;
46  int conflicto;
47
48  /*
49  Leemos el fichero, de donde la línea que empiece por "p"
        leemos el numero de aristas
50  y vertices, y de las que empiecen por "e" las aristas
51  */
52  ifstream myfile("grafonica.txt");
53  if (myfile.is_open()) {
54    while (getline(myfile, linea)) {
55      if (linea[0] == 'p') {
56        istringstream iss(linea, istringstream::in);
57        while (iss >> palabra) {
58          p[i] = palabra;
59          i++;
60        }
61        numv = atoi(p[2].c_str());
62        numa = atoi(p[3].c_str());
63        for (k = 0; k < 2; k++) {
64          aristas[k] = new int[numa];
65        }
66        nodos = new int[numv];
67      }
68      i = 0;
69      if (linea[0] == 'e') {
70        istringstream iss(linea, istringstream::in);
71        while (iss >> palabra) {
72          if (palabra != "e") {
73            aristas[i][j] = atoi(palabra.c_str());
74            i++;
```

```
75            }
76          }
77          j++;
78        }
79      }
80      myfile.close();
81    }
82    else cout << "Incapaz de abrir el archivo";
83
84    srand(time(NULL)); /* Inicializa una semilla aleatoria */
85
86    /* Inicializa los colores en los nodos donde todos tienen la
          misma probabilidad en cada uno */
87    for (i = 0; i < numv; i++) {
88      nodos[i] = rand() % num_colo;
89    }
90
91    /* Matriz de probabilidades para cada color y nodo */
92    double** probabilidad = new double*[numv];
93    for (i = 0; i < numv; i++) {
94      probabilidad[i] = new double[num_colo];
95    }
96    /* Se inicializa la probabilidad igual para todos los
         colores */
97    for (i = 0; i < num_colo; i++) {
98      for (j = 0; j < numv; j++) {
99        probabilidad[j][i] = 1 / numero_colo;
100     }
101   }
102
103   bool* vecindad = new bool[numv]; /* Almacenará la informació
        n sobre si hay conflicto entre nodos (1) o no (0) */
104   for (i = 0; i < numv; i++) {
105     vecindad[i] = 0;
106   }
107   for (i = 0; i < numa; i++) {
108     if (nodos[aristas[0][i] - 1] == nodos[aristas[1][i] - 1])
          {
109       vecindad[aristas[0][i] - 1] = 1;
110       vecindad[aristas[1][i] - 1] = 1;
111     }
112   }
113
114   for (k = 0; k < num_maxiter; k++) {
115
116     /*
```

```
117     Dependiendo de las probabilidades de cada color anterior,
            de si hay conflicto
118     y del color previo, se establecen las nuevas
            probabilidades
119     */
120     for (i = 0; i < numv; i++) {
121       for (j = 0; j < num_colo; j++) {
122         if (vecindad[i] == 1) {
123           if (nodos[i] == j) {
124             probabilidad[i][j] = (1 - beta)*probabilidad[i][j
                    ];
125           }
126           else {
127             probabilidad[i][j] = (1 - beta)*probabilidad[i][j]
                    + beta / (numero_colo - 1);
128           }
129         }
130         else {
131           if (nodos[i] == j) {
132             probabilidad[i][j] = 1;
133           }
134           else {
135             probabilidad[i][j] = 0;
136           }
137         }
138       }
139     }
140
141     /*
142     Una vez tenemos la probabilidad de cada nodo sobre cada
            color elegimos
143     los nuevos colores de forma aleatoria con sus respectivos
            pesos
144     */
145     for (i = 0; i < numv; i++) {
146       nodos[i] = weightedRandomSample(probabilidad[i],
              num_colo);
147     }
148
149     for (i = 0; i < numv; i++) {
150       vecindad[i] = 0;
151     }
152     for (i = 0; i < numa; i++) {
153       if (nodos[aristas[0][i] - 1] == nodos[aristas[1][i] -
              1]) {
154         vecindad[aristas[0][i] - 1] = 1;
```

```
155         vecindad[aristas[1][i] - 1] = 1;
156       }
157     }
158     conflicto = 0;
159     for (i = 0; i < numv; i++) {
160       conflicto = conflicto + vecindad[i];
161     }
162     if (conflicto == 0) {
163       break;
164     }
165   }
166
167   bool colores_usados[num_colo];
168   for (i = 0 ; i < num_colo; i++) {
169     colores_usados[i] = 0;
170   }
171   for (i = 0; i < numv; i++) {
172     colores_usados[nodos[i]] = 1;
173   }
174   int num_colores_usados = 0;
175   for (i = 0; i < num_colo; i++) {
176     num_colores_usados = num_colores_usados + colores_usados[i
          ];
177   }
178
179   /* Almacenamos los resultados en un fichero */
180   ofstream myfile2("solucion.txt");
181   myfile2 << "Solucion con " << num_colores_usados << "
       colores usados y " << conflicto << " nodos implicados en
       conflicto." << endl;
182   myfile2 << "nodo color conflicto" << endl;
183   for(i=0; i < numv; i++) {
184     myfile2 << i + 1 << " " << nodos[i] << " ";
185     if (vecindad[i] == 0) {
186       myfile2 << "No";
187     }
188     else {
189       myfile2 << "Si";
190     }
191     myfile2 << endl;
192   }
193   myfile2.close();
194
195   delete [] nodos;
196   for (int i = 0; i < 2; i++) {
197     delete[] aristas[i];
```

```
198     }
199     delete [] vecindad;
200     for (int i = 0; i < numv; i++) {
201       delete[] probabilidad[i];
202     }
203     delete[] probabilidad;
204     return 0;
205 }
```

Bibliografía

[1] Bermúdez-de-Andrés, J. (2008-09). Diseño de algoritmos (Algoritmos de Vuelta Atrás). OpenCourseWare (Universidad del País Vasco). URL: https://ocw.ehu.eus/pluginfile.php/2224/mod_resource/content/1/diseno_alg/contenidos/algoritmos-de-vuelta-atras.pdf

[2] Duffy, K.R., O'connell, N. y Sapozhnikov, A. (2008). Complexity analysis of a decentralised graph colouring algorithm. *Information Processing Letters, 107*(2), 60-63.

[3] Galinier, P. y Hao, J.-K. (1999). Hybrid evolutionary algorithms for graph coloring. *Journal of Combinatorial Optimization, 3*, 379–397.

[4] Gayo-Avello, D. Algorítmica y Lenguajes de Programación (Complejidad computacional). Web de Universidad de Oviedo (Departamento de Informática). URL: http://di002.edv.uniovi.es/~dani/asignaturas/transparencias-leccion19.PDF

[5] Gonthier, G. (2008). Formal Proof — The Four-Color Theorem.

[6] Kubale, M. y Jackowski, B. (1985). A generalized implicit enumeration algorithm for graph coloring. *Commun. ACM, 28*(28), 412–418.

[7] Kučera, L. (1977). Expected behavior of graph coloring algorithms (Fundamentals of Computation Theory). *Lecture Notes in Computer Science, 56*, 447-451.

[8] Leith, D. J. y Clifford, P. (2006). Convergence of distributed learning algorithms for optimal wireless channel allocation. *IEEE CDC*, 2980–2985.

[9] Lewis, R. (2015). A Guide to Graph Colouring: Algorithms and Applications. Berlin, Springer. ISBN: 978-3-319-25728-0. http://www.springer.com/us/book/9783319257280

[10] Lewis, R., Thompson, J., Mumford, C., y Gillard, J. (2012). A Wide-Ranging Computational Comparison of High-Performance Graph Co-

louring Algorithms. *Computers and Operations Research, 39*(9), 1933-1950.

[11] Malone, D., Clifford, P., Reid, D. y Leith, D. (2007). Experimental implementation of optimal WLAN channel selection without communication. *IEEE DySPAN*, 316–319.

[12] Nyberg, S.D. (2016). Physical Cell ID Allocation in Cellular Networks.

[13] Sedeño-Noda, A.A. (2017-2018). Apuntes de la asignatura de Optimización del grado de Ingeniería Informática.

Lista de Figuras

Mobile Network Optimization

Daniel Díaz Ríos
Facultad de Ciencias · Sección de Matemáticas
Universidad de La Laguna
alu0100888881@ull.edu.es

Abstract

Within the mobile communications networks, there is an assignment problem with a limited resource called "Physical Cell Id", which identifies the cells that a mobile terminal connects to. This paper tries to model and solve this problem starting out from a real case, finding some similarities with graph colouring problems, and carrying out a bibliographic search to find and use the best algorithms. The aim is to optimize this allocation to improve the behaviour of the LTE communication network.

1. Introduction

We start with the theoretical study side, where we define necessary concepts which will be helpful or used throughout the problem resolution, as well as the explanation of some existing algorithms which we will use. Later, we describe in a general way the real problem, and some processes that must be followed to face it. Finally we deal with our particular problem, where we specify how it has been solved, all the encountered obstacles, what steps have been followed to solve them, and what solutions are finally obtained.

2. Theoretical fundamentals

Down below we show some graph colouring algorithms which could be useful throughout the problem resolution.

Independent Set algorithm:

This algorithm doesn't preset the maximum number of colours to use, but it always gives us a colouring minimizing the number of colours used. Despite being simple, this algorithm usually provides us very good results, it is also always very efficient.

Algorithm 1 Independent Set algorithm

$X \leftarrow$ vertices
$Y \leftarrow X$
$r \leftarrow 1$
while $X \neq \emptyset$ **do**
  while $Y \neq \emptyset$ **do**
    Choose $i \in Y$
    Assign colour r to vertex i
    $X \leftarrow X \setminus \{i\}$
    $Y \leftarrow Y \setminus \{i \wedge j : j \text{ is adjacent to } i\}$
  end while
  $Y \leftarrow X$
  $r \leftarrow r + 1$
end while

Communication-Free Learning:

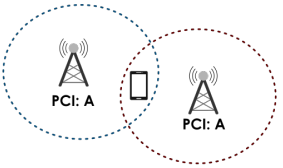
It is a decentralised stochastic colouring algorithm that converges to the optimal solution with high probability. This method chooses the colour of every node ($k \in 1,...,n$) at each instant $t \in \{0,1,...\}$, where it will be represented by $c_k(t)$ which depends on the node and the instant that we're in. The colours to be used in this algorithm is predefined as $\{1,...,C\}$, which are chosen randomly according to the probability distribution $p_k(t)$, which is initialised for $t = 0$ as $p_k(0) = (1/C,...,1/C)$ $\forall k \in \{1,...,n\}$ and for $t \geq 1$ as follows. In addition, this distribution depends on the β value to be chosen in $(0,1)$.

$$p_k(t+1) = \begin{cases} \delta_{c_k(t)} \\ \quad \text{si } c_k(t) \neq c_i(t) \ \forall i \in \Gamma_k \\ (1-\beta)p_k(t) + \frac{\beta}{C-1}\bar{\delta}_{c_k(t)} \\ \quad \text{si } \exists i \in \Gamma_k : c_k(t) = c_i(t) \end{cases}$$

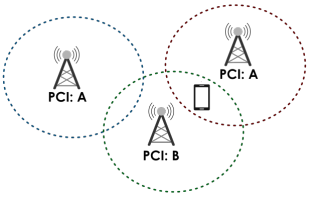
3. Presentation of the real problem

Mobile communications networks are characterized by having a cellular structure that allows the reuse of limited radio resources, to ensure quality and capacity of service to users who access to the network. One of these resources is the Physical Cell Id (PCI), it's the cell identifier which is located in the physical layer of the LTE (Long Term Evolution) network, which aims to identify the cells that a terminal is connected to. But the number of PCIs is finite, by design it's limited to 504. Two unwanted events that may accur are: PCI collision and confusion.

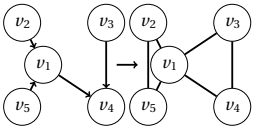
A collision occurs when two neighboring cells have the same PCI:



A confusion occurs when a cell has two adjacent cells with the same PCI:

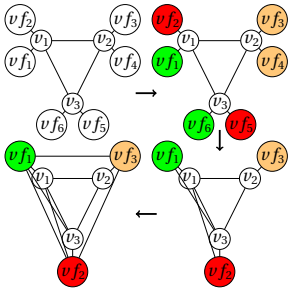


We will use the graph colouring problem to solve the PCI collision and confusion conflicts, where the vertices will represent the cells, the colours, the numbers of their PCIs and the edges the adjacent relationships. This is not exactly a colouring problem, because although collisions can be minimised by following this procedure, we must also minimise confusions. In order to reduce them, the predecessor of the successor nodes of a vertice will become neighbors too, as shown in the directed graph (our case) below:



4. Resolution of the real case

There are several obstacles to the resolution of the real case, one of them is the existence of some cells (fixed nodes) that can't change their initial PCI. Looking to solve this issue, we perform a data preprocessing in which we follow several phases as an adaptation of the graph showed below, to take this into account.



References

[1] Kučera, L. (1977). Expected behavior of graph colouring algorithms (Fundamentals of Computation Theory). *Lecture Notes in Computer Science, 56*, 447-451.

[2] Duffy, K.R., O'connell, N. y Sapozhnikov, A. (2008). Complexity analysis of a decentralised graph colouring algorithm. *Information Processing Letters, 107*(2), 60-63.

[3] Lewis, R., Thompson, J., Mumford, C., y Gillard, J. (2012). A Wide-Ranging Computational Comparison of High-Performance Graph Colouring Algorithms. *Computers and Operations Research, 39*(9), 1933-1950.

[4] Nyberg, S.D. (2016). Physical Cell ID Allocation in Cellular Networks.