UNIVERSIDAD DE ALCALÁ
Escuela Politécnica Superior

GRADO EN INGENIERÍA TELEMÁTICA

Trabajo Fin de Grado

Aplicación de técnicas de negociación a la coloración de grafos y su aplicación en redes cognitivas

Autor: Javier Lorenzo Díaz

Director/es: D. David Orden Martín y D. Iván Marsá Maestre

TRIBUNAL:

Presidente: D. Pedro Ramos Alonso

Vocal 1º: D. Enrique de la Hoz de la Hoz

Vocal 2º: D. David Orden Martín

CALIFICACIÓN: .....

FECHA: .....

Dedicado a mi familia y amigos, por todo su apoyo,
a mis profesores, por su ayuda y dedicación y,
en especial, a Andrea, por hacerme feliz y
enseñarme a no rendirme jamás.

Índice

Índice **i**

Lista de figuras **iv**

Lista de tablas **v**

1 Resumen Extendido **xi**
1.1 Introducción xi
1.2 Generador de grafos euclídeos xi
1.3 Implementaciones de los algoritmos xii

2 Introducción **xiii**

3 Aspectos teóricos **1**
3.1 Teoría de grafos 1
3.1.1 Definiciones 1
3.1.2 Propiedades de los grafos 2
3.1.3 Tipos de grafo 2

4 Generador de grafos euclídeos **7**
4.1 Descripción 7
4.1.1 Posición 7
4.1.2 Interferencia y punto de acceso más cercano 13

5 Algoritmos implementados **17**
5.1 Algoritmo `LCCS` 17
5.1.1 Descripción 17
5.1.2 Simulación 18
5.2 Modificación algoritmo DSATUR 18
5.2.1 Base teórica 19
5.2.2 Descripción DSATUR 19
5.2.3 Descripción modificación DSATUR 21
5.2.4 Simulación 22
5.2.5 Conclusiones 27
5.3 Algoritmo *Accumulated-Weight* (AW) 27

5.3.1 Base teórica . . . 27
5.3.2 Descripción . . . 28
5.3.3 Simulación . . . 32
5.3.4 Conclusiones . . . 37
5.4 Algoritmos Hminmax y Hsum . . . 38
5.4.1 Base teórica . . . 38
5.4.2 Coloración de grafos ponderada . . . 40
5.4.3 Definición formal . . . 40
5.4.4 Descripción . . . 41
5.4.5 Simulación y conclusiones . . . 42

6 Conclusiones generales y trabajo futuro 49

A Manual de usuario 51
A.1 Proyecto mishra: Generador de grafos euclídeos, LCCS, Hminmax y Hsum . . . 51
A.1.1 Generador de grafos euclídeos . . . 51
A.1.2 Algoritmo LCCS . . . 52
A.1.3 Algoritmo HMINMAX . . . 52
A.1.4 Algoritmo HSUM . . . 53
A.1.5 Herramientas . . . 53
A.2 Proyecto dsatur: DSATUR y M-DSATUR . . . 54
A.2.1 Módulo `auto_gen_graph` . . . 54
A.2.2 Módulo `coef_cochannel` . . . 54
A.2.3 Módulo `dsatur` . . . 54
A.2.4 Módulo `gen_generic_graph` . . . 54
A.2.5 Módulo `graphical_util` . . . 55
A.2.6 Módulo `other_coloring_alg` . . . 55
A.2.7 Módulo `read_write` . . . 55
A.3 Proyecto aw y auto_aw: *Accumulated Weight* . . . 55
A.3.1 Módulo `node` . . . 55
A.3.2 Módulo `main` . . . 55
A.3.3 Módulo `execution` . . . 55
A.3.4 Módulo `global_var` . . . 55
A.3.5 Módulo `aux_func_main` . . . 56
A.3.6 Módulo `read_write` . . . 56
A.3.7 Módulo `coef_cochannel` . . . 56
A.4 Organización de los archivos en el CD . . . 56
A.4.1 Proyecto mishra . . . 56
A.4.2 Proyecto dsatur . . . 57
A.4.3 Proyectos auto_aw . . . 58

B Pliego de condiciones 59
B.1 Hardware: Especificaciones técnicas . . . 59

B.1.1 Especificaciones mínimas 60
B.2 Software: Programas utilizados 60

C Presupuesto **63**
C.1 Gastos de mano de obra 63
C.2 Costes materiales 63
C.3 Presupuesto final 63

Referencias **65**

Lista de figuras

2.1 Ej. de distribución frecuencial de canales. xiii

3.1 Ej. de un grafo G 1
3.2 Ej. diferencia entre grafo plano y no plano. 1
3.3 Ej. grafo 2-regular. 2
3.4 Ej. grafo dirigido. 2
3.5 Ej. grafo completo. 2
3.6 Ej. de grafo aleatorio geométrico con 50 nodos y un radio 0.125. 3
3.7 Ej. de grafo 2-regular aleatorio que sigue una distribución uniforme. 4
3.8 Ej. de grafo Erdős–Rényi. 5

4.1 Ej. de distribución uniforme continua. 8
4.2 Ej. de distribución de Kuzmin. 9
4.3 Representación gráfica de la función de densidad de probabilidad de la distribución normal. . 10
4.4 Ej. de distribución normal 10
4.5 Ej. malla cuadrada. 11
4.6 Ej. de casos de mallas triangulares. 12
4.7 Ej. triangulación de Delaunay y NNG modificado. 14
4.8 Ej. de grafo UDG. 15

5.1 Interferencia entre clientes. 18
5.2 Ej. de coloración apropiada de grafos. 19
5.3 Ej. del algoritmo DSATUR 20
5.4 Número de colores medio utilizado, en grafos Erdős–Rényi. 23
5.5 Utilidad media, en grafos Erdős–Rényi. 24
5.6 Número de colores medio utilizado, en grafos Barabási-Albert. 25
5.7 Utilidad media, en grafos Barabási-Albert. 26
5.8 Máquina de estados del algoritmo AW. 28
5.9 Diagrama de flujo de iteración en la implementación del algoritmo AW. 33
5.10 $nn = 10$, $nc = 2$ y $d = 1$. Coloración resultante de algoritmo AW. 35
5.11 $nn = 10$, $nc = 2$ y $d = 2$. Coloración resultante de algoritmo AW. 36

Lista de tablas

5.1 $nn = 10$ . . . 35
5.2 $nn = 20$ . . . 35
5.3 $nn = 40$ . . . 35
5.4 $nn = 80$ . . . 35
5.5 Resultados correspondientes al tiempo total de ejecución de los distintos casos, con un tiempo máximo de convergencia $t = 40000$ . . . 35
5.6 $nn = 10$ . . . 37
5.7 $nn = 20$ . . . 37
5.8 $nn = 40$ . . . 37
5.9 $nn = 80$ . . . 37
5.10 Resultados correspondientes al tiempo total de ejecución de los distintos casos, con un tiempo máximo de convergencia $t = 120000$ . . . 37
5.11 Ej. empírico de valores de *I-factor* . . . 40
5.12 Caso *random-random* . . . 44
5.13 Caso *square-random* . . . 44
5.14 Caso *random-normal* . . . 44
5.15 Caso *square-normal* . . . 44
5.16 Caso *random-random* . . . 45
5.17 Caso *random-normal* . . . 46
5.18 Caso *square-random* . . . 46
5.19 Caso *square-normal* . . . 46

C.1 Costes materiales . . . 64

Resumen

El objetivo del siguiente TFG consiste en implementar en Python una serie de algoritmos ya existentes relacionados con la coloración de grafos, una de las técnicas más utilizadas para mejorar el aprovechamiento del espectro electromagnético en redes inalámbricas, para posteriormente poderlos contrastar con otros que incluyen métodos de negociación automática. Éstos, junto a la incorporación de nuevas capas de coloreado y nuevos pesos, diferentes a los tradicionales entre vértices y aristas, son necesarios para conducir el carácter simplista de los algoritmos ya existentes hacia el caso real de las redes presentes actualmente.

Palabras clave

Coloración de grafos, Redes cognitivas, Espectro, Frecuencias, Negociación.

Abstract

The purpose of the following work is to implement a number of existing algorithms in Python related to graph coloring, one of the most used techniques to improve the electromagnetic spectrum exploitation in wireless networks, so later they can be contrasted with other algorithms based on auto-negotiation techniques. These techniques, together with the addition of new coloring layers and weights, different from the traditional ones between vertex and edges, are necessary to drive the existing algorithms to the real case of nowadays networks.

Keywords

Graph-coloring, Cognitive networks, Spectrum, Frequencies, Negotiation.

Apartado 1

Resumen Extendido

1.1 Introducción

El gran desarrollo de la tecnología inalámbrica y el consiguiente aumento de redes de este tipo en las últimas décadas ha provocado una disminución considerable del espectro electromagnético libre, convirtiendo este recurso en un producto muy cotizado mundialmente.

Debido al alto coste que supone la posesión de una banda de frecuencia, ha surgido con el tiempo una solución rápida orientada a la disminución de los gastos: la **reutilización de frecuencias**. Sin embargo, esta solución acarrea un problema: provoca la aparición de interferencia entre bandas de frecuencia cercanas que depende de la potencia de ambas señales y de la cercanía entre otros múltiples factores. Por lo tanto, surge un problema a solucionar, conocido como problema de **asignación de frecuencias**, el cual como su nombre indica, busca una solución óptima en la que se asignen bandas de frecuencia a los nodos de la red sin que la interferencia entre ellas degrade la calidad del servicio. Aunque es conocido como el problema de asignación de frecuencia, en realidad no se trata de un problema único sino que existen numerosas variaciones dependiendo de varios factores como el tipo de red y sus características.

En este trabajo se ha utilizado el problema de la coloración de grafos para intentar progresar en la búsqueda de una solución a la asignación de frecuencias, utilizando los grafos como aproximaciones de las redes en los cuales, los vértices representan los nodos de la red (clientes y puntos de acceso); las aristas, la interferencia entre nodos y los colores, los canales permitidos.

A pesar de ser una de las técnicas más ampliamente utilizadas, presenta el inconveniente de simplificar el problema, prescindiendo de multitud de parámetros presentes en las redes reales. Por lo tanto, para intentar enfocar la solución al problema real es necesario añadir nuevas capas al coloreado, pesos a las aristas y permitir aristas monocromáticas entre otras mejoras.

1.2 Generador de grafos euclídeos

Para poder realizar simulaciones sobre escenarios orientados al caso real (grafos euclídeos) para las distintas implementaciones de los algoritmos, es necesario en primer lugar implementar un generador de

grafos que permita generar varias configuraciones de las posiciones para los vértices, y también permita generar grafos de **interferencia** y de **punto de acceso más cercano**.

1.3 Implementaciones de los algoritmos

Los algoritmos implementados pueden clasificarse según su nivel de abstracción y complejidad.

En primer lugar, LCCS es un algoritmo muy extendido mundialmente, sin embargo presenta más desventajas que ventajas, lo que se refleja en los malos resultados obtenidos en su comparación con una coloración aleatoria uniforme.

A continuación, los algoritmos DSATUR y su modificación M-DSATUR intentan solucionar el problema de asignación de frecuencia mediante coloreado apropiado de grafos (*proper coloring*). Los resultados de las simulaciones de ambos algoritmos son comparados con los resultados de un algoritmo de coloración aleatoria, obteniendo mejores resultados para DSATUR y M-DSATUR, siendo los resultados de este último los mejores, debido a que toma en cuenta la interferencia cocanal para elegir el color a utilizar.

El algoritmo *Accumulated-Weight* pretende ir más allá que los anteriores en su solución, utilizando comunicación entre nodos y computación distribuida para obtener unos resultados bastante buenos finalmente, con el inconveniente de que, debido a establecerse un número fijo de colores, existen casos en los que el algoritmo no llega a converger.

Para finalizar, los algoritmos *Hminmax* y *Hsum*, presentan otro elemento que podría ser clave en futuros desarrollos: las aristas ponderadas. Se obtienen resultados mejores que en el caso aleatorio en el caso de estos algoritmos, sin embargo, los resultados podrían mejorar si se pudiese disponer de información acerca del procedimiento a la hora de elegir el color de una lista de ellos.

Apartado 2

Introducción

Las redes inalámbricas han experimentado en las últimas dos décadas un rápido desarrollo, dando como resultado una escasez de frecuencias en la franja correspondiente del espectro electromagnético utilizada en esta clase de redes.

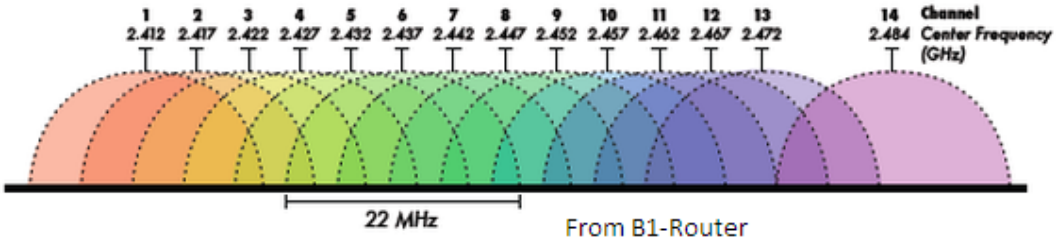


Figura 2.1: Ej. de distribución frecuencial de canales.

Una de las principales consecuencias de esta escasez ha sido la implantación progresiva de una comercialización de frecuencias, la cual ha conducido a su vez a una reutilización de éstas para evitar costes innecesarios. Sin embargo, esta solución es bastante problemática debido al uso de frecuencias similares por diferentes redes, lo que puede provocar interferencias entre ellas y por tanto una disminución de la calidad de las comunicaciones. Es aquí donde surge el problema de la **asignación de frecuencias**, cuya solución es necesaria para paliar estos inconvenientes. En la figura 2.1 se puede observar un ejemplo de interferencia entre canales debido a la cercanía entre ellos al no disponer de un mayor ancho de banda para desplegarlos.

Este problema no es único, sino que depende en gran medida del tipo de red sobre la que se va a aplicar, sus características específicas, la finalidad deseada por los clientes y los proveedores y la subjetividad de los investigadores encargados, provocando un gran número de modelos para el mismo problema.

En el caso de este trabajo, el modelo principal sometido a estudio, es el problema de la **coloración de grafos**, cuya generalización directa es el problema de asignación de frecuencias de orden mínimo (*Minimum Order Frequency Assignment Problem* o MO-FAP), en el cual se persigue evitar interferencias inaceptables y minimizar el número de frecuencias utilizadas durante la asignación de frecuencias.

La coloración de grafos es una de las técnicas más ampliamente utilizadas en la asignación de frecuencias en redes cognitivas. Sin embargo, presenta un inconveniente principal: se trata de una simplificación del problema expuesto, lo que provoca que la mayoría de los algoritmos basados en esta técnica prescindan de gran parte de los parámetros presentes en estas redes. Esto es debido a la gran complejidad que presenta la coloración de grafos, perteneciente a la clase de problemas NP-completos.

La revisión [1] establece la necesidad de incorporar nuevas capas al coloreado así como otros tipos de pesos, diferentes a los tradicionales de vértices y aristas, para modelar adecuadamente el comportamiento y las necesidades de las redes cognitivas.

El propósito del siguiente proyecto comprende varios objetivos a cumplir. En primer lugar, tras la revisión de bibliografía relacionada con asignación de frecuencias, redes cognitivas y coloración de grafos, se pondrá a punto un generador de grafos capaz de modelar diversas estructuras de red que permitan realizar simulaciones de casos reales.

A continuación se implementarán herramientas que permitan dotar a los grafos de nuevas capas de coloreado y nuevos tipos de peso.

Tras cumplir este último objetivo, el siguiente paso será realizar varias implementaciones de algoritmos ya existentes, relacionados con la coloración de grafos, en el lenguaje de programación interpretado **Python** de tal forma que se puedan lanzar conjuntos de experimentos diseñados para satisfacer diversas situaciones.

Por último, se documentarán debidamente las tareas realizadas y se propondrán posibles mejoras futuras.

La estructura de este trabajo es la siguiente: en el capítulo 3, *Aspectos teóricos*, se realizará una introducción al campo clave relacionado con este TFG, la teoría de grafos. En el capítulo 4, se explicará en profundidad el generador de grafos, para dar paso al capítulo 5, en el que se tratarán los distintos algoritmos utilizados. Por último, en el capítulo 6 se realizarán las conclusiones del trabajo en conjunto y las posibles mejoras futuras. Además, existirán varios anexos que contendrán el manual de usuario, el pliego de condiciones y el presupuesto, permitiendo un mayor entendimiento a las implementaciones realizadas y a la forma de utilizarlas.

Apartado 3

Aspectos teóricos

En este capítulo, se realizará una breve introducción teórica a la teoría de grafos, necesaria para la comprensión de este trabajo, al utilizarse grafos en la simulación de redes y escenarios.

3.1 Teoría de grafos

La **teoría de grafos** es un campo perteneciente a las ramas de las matemáticas y las ciencias de la computación, que se encarga del estudio de los **grafos**, diagramas consistentes en un conjunto de puntos (vértices) con líneas uniendo determinados pares de ellos (aristas) [2]. A continuación se definirán formalmente estos conceptos.

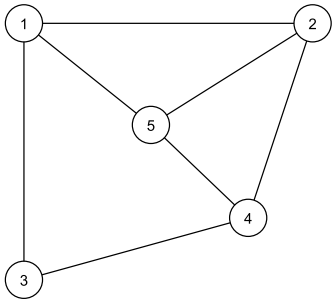


Figura 3.1: Ej. de un grafo G

3.1.1 Definiciones

Un grafo **G**, está formado por dos conjuntos (V, E). V corresponde al conjunto de vértices y E al conjunto de aristas, las cuales son pares de vértices [3].

Como ejemplo, en la figura superior derecha (fig. 3.1) se muestra un grafo G, con

$$V = \{1, 2, 3, 4, 5\}, E = \{(1, 2), (1, 3), (1, 5), (2, 4), (2, 5), (3, 4), (4, 5)\}$$

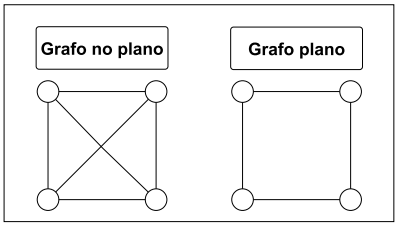


Figura 3.2: Ej. diferencia entre grafo plano y no plano.

Los extremos de una arista (vértices), se dice que son **incidentes** con ella y viceversa. Dos vértices se dice que son **adyacentes** si son incidentes con una misma arista, es decir, si ambos están conectados por una misma arista. Dos aristas que inciden con un mismo vértice, también reciben este nombre. En el contexto de este trabajo, se usará el término análogo **vecinos** en su lugar. Una arista cuyos extremos son el mismo vértice, se denomina **lazo**.

El **grado** de un vértice v, escrito como $d(v)$, es el número de aristas en las que v es un extremo. Un vértice cuyo grado es 1, se denomina vértice **colgante**. Si el grado es 0, pasa a denominarse vértice **aislado**.

Por último, el **orden** de un grafo indica el número de vértices que contiene, y el **tamaño**, el de aristas.

3.1.2 Propiedades de los grafos

Un grafo se dice que es **finito**, si los conjuntos V de vértices y E de aristas, son finitos.

Un grafo es **simple** si no presenta bucles ni más de una arista entre el mismo par de vértices (e.g. fig. 3.1). En el caso de este trabajo, todos los grafos utilizados son de este tipo.

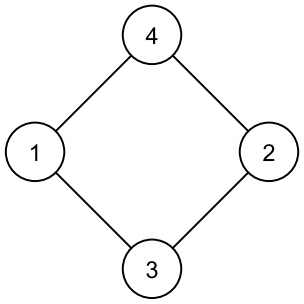


Figura 3.3: Ej. grafo 2-regular.

Se entiende por grafo **plano**, aquel que puede ser representado en un espacio bidimensional o plano, es decir, no puede ser dibujado de forma que algún par de aristas se crucen (e.g. fig. 3.2)

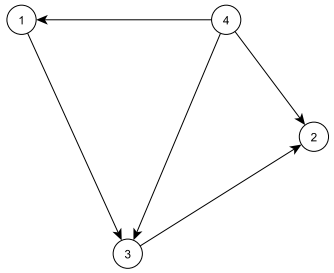


Figura 3.4: Ej. grafo dirigido.

Un grafo **regular**, es aquel en el que el grado es igual para todos sus vértices. Se denomina grafo **n-regular** a aquel cuyo grado equivale a n (e.g. fig.3.3)

Se denomina **grafo dirigido** o **digrafo**, a un grafo en el cual sus aristas tienen una dirección y sentido definidos, a diferencia del **grafo no dirigido** o **grafo propiamente dicho**, cuyas aristas no presentan dirección alguna.

Se dice que un grafo simple es **completo**, si existe una arista entre todos los pares de vértices posibles. El número de aristas presente en grafos con esta característica, puede ser calculado a través de la siguiente ecuación: $n_E = n_V \cdot \frac{n_V - 1}{2}$, donde n_V es el número de vértices.

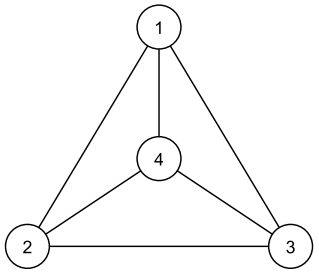


Figura 3.5: Ej. grafo completo.

3.1.3 Tipos de grafo

En teoría de grafos, existe un gran número de modelos de grafo. A continuación, se realizará una introducción a varios de ellos, utilizados cada uno de ellos en al menos uno de los algoritmos implementados.

3.1.3.1 Grafo euclídeo

Un **grafo geométrico** es un grafo cuyos vértices y aristas se asocian con **objetos geométricos** [4].

Un grafo **euclídeo**, subtipo del grafo geométrico, es aquel en el que sus vértices representan puntos en el plano, y sus aristas, la distancia euclídea entre sus extremos, la cual se calcula a través de la siguiente

ecuación, para el caso de 2 dimensiones:

$$d(p,q) = \sqrt{(x_q - x_p)^2 + (y_q - y_p)^2}$$

En el contexto de este trabajo, los grafos utilizados de este tipo tienen las siguientes características comunes: son **finitos**, **simples**, **no planos**, y **no dirigidos**.

A continuación, en la figura 3.6, se muestra un ejemplo de grafo aleatorio geométrico generado mediante el generador de grafos `random_geometric_graph` perteneciente al *package* **NetworkX** [6]. Este modelo de grafo posiciona los nodos mediante una distribución aleatoria uniforme en el intervalo bidimensional $[0, 1] \times [0, 1]$ y añade una arista entre 2 nodos u y v si la distancia euclídea entre ambos es menor que un parámetro r (*radius*).

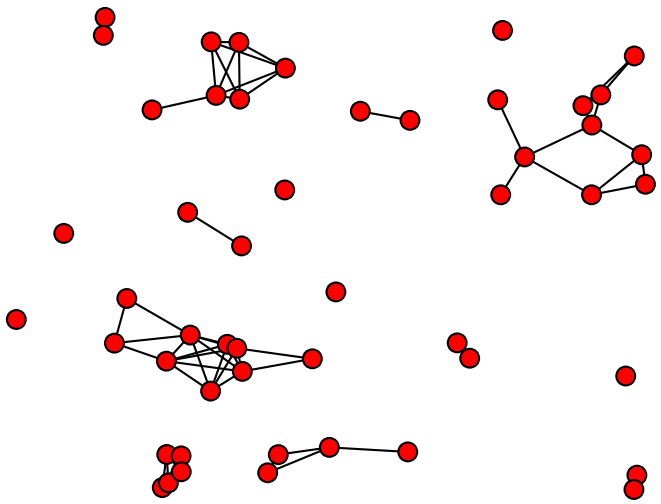


Figura 3.6: Ej. de grafo aleatorio geométrico con 50 nodos y un radio 0.125.

3.1.3.2 Grafo aleatorio

Es un tipo de grafo bastante general que abarca muchos subtipos, entre los que se encuentra el grafo generado a través del modelo **Erdős–Rényi**.

Son grafos generados mediante algún tipo de **proceso aleatorio** o **distribución probabilística**. Como ejemplo, en una de las implementaciones de algoritmos, explicadas en el siguiente capítulo, se utiliza un modelo de grafo regular aleatorio que genera el mismo número de aristas para cada vértice (debido a

la regularidad del grafo) de forma aleatoria, siguiendo una distribución aleatoria uniforme discreta (e.g. fig 3.7)

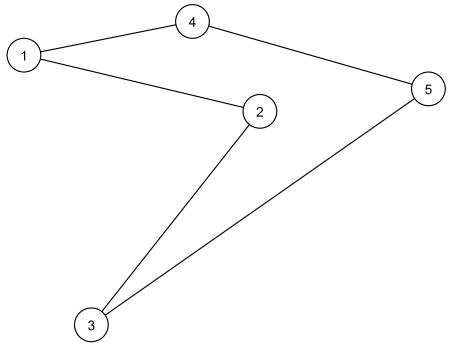


Figura 3.7: Ej. de grafo 2-regular aleatorio que sigue una distribución uniforme.

3.1.3.3 Modelo Barabási-Albert

Este modelo comprende un algoritmo que genera grafos que siguen una **ley potencial** (grafos libres de escala) en la distribución del grado de sus vértices, siguiendo un mecanismo de **conexión preferencial** [23].

El algoritmo sigue el siguiente procedimiento:

1. El grafo comienza con un número pequeño m_0 de vértices.
2. Cada unidad de tiempo se añade un vértice al grafo con m aristas ($m \leq m_0$) conectadas a m vértices diferentes ya existentes en el grafo.
3. La elección de dichos vértices se realiza mediante un mecanismo de conexión preferencial: la probabilidad de que un vértice i se una al nuevo vértice mediante una arista esta relacionada con su grado (d_i) a través de la siguiente ecuación:

$$P(i) = \frac{d_i}{\sum_{j=0}^{|V|} d_j}$$

3.1.3.4 Modelo Erdős–Rényi

El modelo **Erdős–Rényi** [8] es un modelo probabilístico basado en la generación de grafos aleatorios $G(n, p)$, cuya probabilidad de creación de arista entre dos nodos p es fija e **independiente** del resto de aristas.

Un grafo dentro de $G(n, p)$ tiene un número esperado de aristas $\binom{n}{2}p$. La probabilidad de que el grado de cualquier vértice en un grafo de este conjunto sea k se caracteriza por seguir una distribución binomial, como muestra la siguiente ecuación:

$$P(deg(v) = k) = \binom{n-1}{k} p^k (1-p)^{n-1-k}$$

Interpretando la anterior expresión, si en nuestro grafo tenemos n vértices, un vértice v puede unirse mediante una arista a k vértices de los $n-1$ restantes en el grafo. Existen $\binom{n-1}{k}$ combinaciones posibles

de k vértices del total de $n-1$, y la probabilidad de que k vértices concretos se unan al vértice v es de $p^k(1-p)^{n-1-k}$ [9].

Este tipo de modelos son utilizados entre otras aplicaciones para estudiar diversas propiedades de los grafos. Como ejemplo, en la figura 3.8 se muestra un grafo con 30 vértices y probabilidad de arista $p = 0.4$.

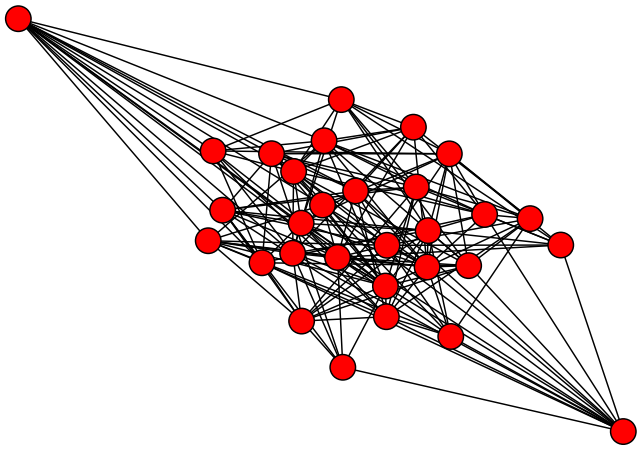


Figura 3.8: Ej. de grafo Erdős–Rényi.

Apartado 4

Generador de grafos euclídeos

Varios de los algoritmos implementados están ideados para ser utilizados en escenarios reales, como es el caso de *Hminmax* y *Hsum*, por lo que una de las primeras herramientas a diseñar ha sido un generador de grafos euclídeos, el cual se describe a continuación.

4.1 Descripción

En esta sección, se tratará en primer lugar la generación de la posición de los nodos para dar paso posteriormente a los tipos de grafos que el generador puede crear.

4.1.1 Posición

Un grafo **euclídeo** está formado por un conjunto de aristas y un conjunto de vértices como en todos los tipos de grafo. Sin embargo, en este caso concreto, los vértices representan puntos en el plano y las aristas, la distancia euclídea entre los vértices de sus extremos. Por lo tanto, el primer paso para crear el generador de grafos es la **generación de la posición** de los vértices.

En este TFG se han utilizado varias distribuciones aleatorias y estructuras geométricas en malla para calcular la posición de los vértices. Debido a la orientación real de dicho generador, se ha distinguido desde un primer momento la posición de los nodos clientes y la de los puntos de acceso (APs). A continuación se listan con una explicación breve:

- **Distribución uniforme continua**. Se denota como $U(a, b)$ la distribución continua en la que los valores reales comprendidos en el intervalo $a \leq x \leq b$ son equiprobables. Su función de densidad de probabilidad, por tanto, corresponde a la siguiente función definida a trozos:

$$f(x) = \begin{cases} \dfrac{1}{b-a}, & \text{para } a \leq x \leq b \\ 0, & \text{para } x < a \text{ o } x > b \end{cases}$$

  En la figura 4.1 se puede observar una representación gráfica de 10000 puntos, cuya posición ha sido generada a partir de una distribución uniforme continua $U(0, 1)$

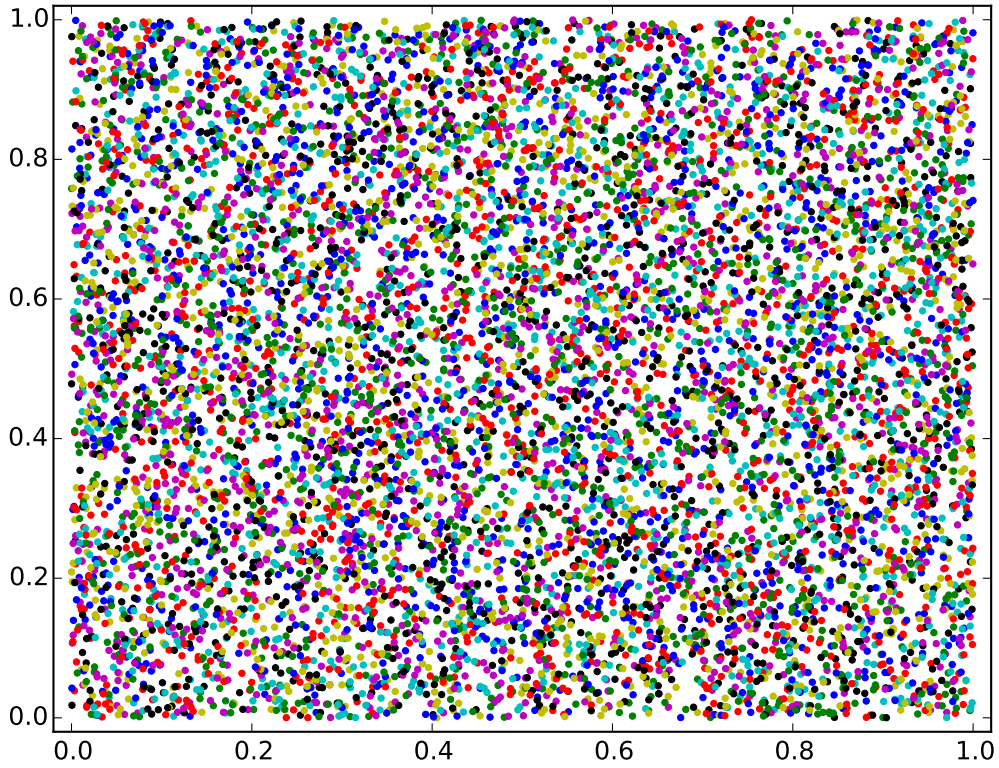


Figura 4.1: Ej. de distribución uniforme continua.

- **Distribución de Kuzmin**. Se trata de una distribución utilizada en el campo de la astrofísica para modelar la distribución de cúmulos estelares. Es radialmente simétrica y su densidad disminuye rápidamente conforme el radio aumenta [12].

  En [13] se explica un procedimiento para generar puntos en dos dimensiones siguiendo la distribución de Kuzmin, que consta de los siguientes pasos:

  1. Generar un punto p cuyos componentes cartesianos, sigan una distribución uniforme $U(-1, 1)$.
  2. Obtener el punto q del modelo Kuzmin de la siguiente manera (escalando p):

  $$q = \frac{p}{|p|}\left[\frac{1}{(1-|p|)^2} - 1\right]$$

  En la figura 4.2, se muestra un ejemplo gráfico de 10000 puntos generados mediante el procedimiento anterior. El ejemplo mostrado, acota los valores máximos y mínimos en las coordenadas x e y a 10 y -10 ($[-10, 10] \times [-10, 10]$), ya que sin realizar esta acotación, algunos puntos surgirían bastante alejados y no se visualizaría correctamente.

  Aquellos puntos tan alejados no interesan a la hora de realizar experimentos de asignación de frecuencia por lo que en el caso de la imagen, el algoritmo trunca las coordenadas de los puntos

fuera de los límites a los valores máximos y mínimos, es decir, mantiene todos los puntos en el intervalo bidimensional $[-10, 10] \times [-10, 10]$. Otra forma también implementada, es generar puntos e incrementar un contador por cada punto dentro del intervalo permitido hasta que ese contador alcance el número de puntos requerido.

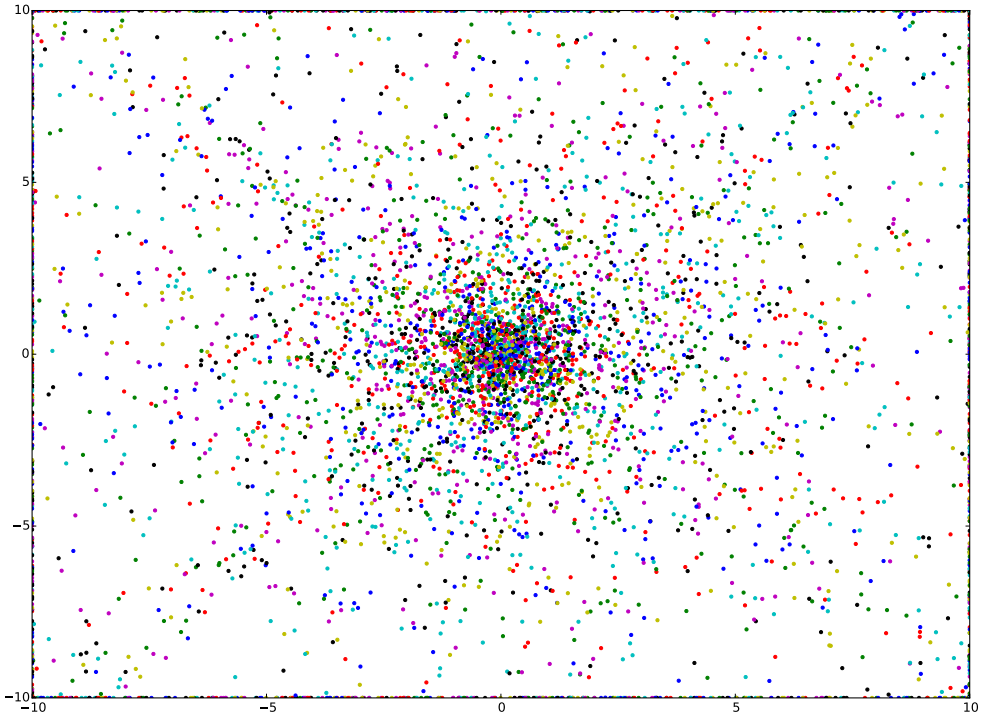


Figura 4.2: Ej. de distribución de Kuzmin.

- **Distribución normal**. Se trata de una distribución radialmente simétrica con densidad central bastante menor que en el caso de la distribución de Kuzmin [12]. Su función de densidad de probabilidad (representada en la fig. 4.3) es la siguiente:

$$f(x \mid \mu\sigma) = \frac{1}{\sigma\sqrt{2\pi}}\, e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

El caso concreto con $\mu = 0$ y $\sigma = 1$ se denomina **distribución normal estándar** y se denota como $N(0, 1)$ [14]. Es muy utilizada en el campo de las ciencias naturales y sociales para realizar estadísticas y representar variables aleatorias cuya distribución no esté clara.

A continuación en la fig. 4.4, se muestra un ejemplo de 1000 puntos cuyas posiciones han sido generadas siguiendo una distribución normal estándar.

- **Malla cuadrada**. Esta estructura consiste simplemente en, dado un número de puntos y un punto de referencia en el plano (en nuestro caso, se utiliza el origen de coordenadas $(0, 0)$ por comodidad), se genera una malla cuadrada en el intervalo 2-dimensional $[0, n] \times [0, n]$ donde n corresponde a un

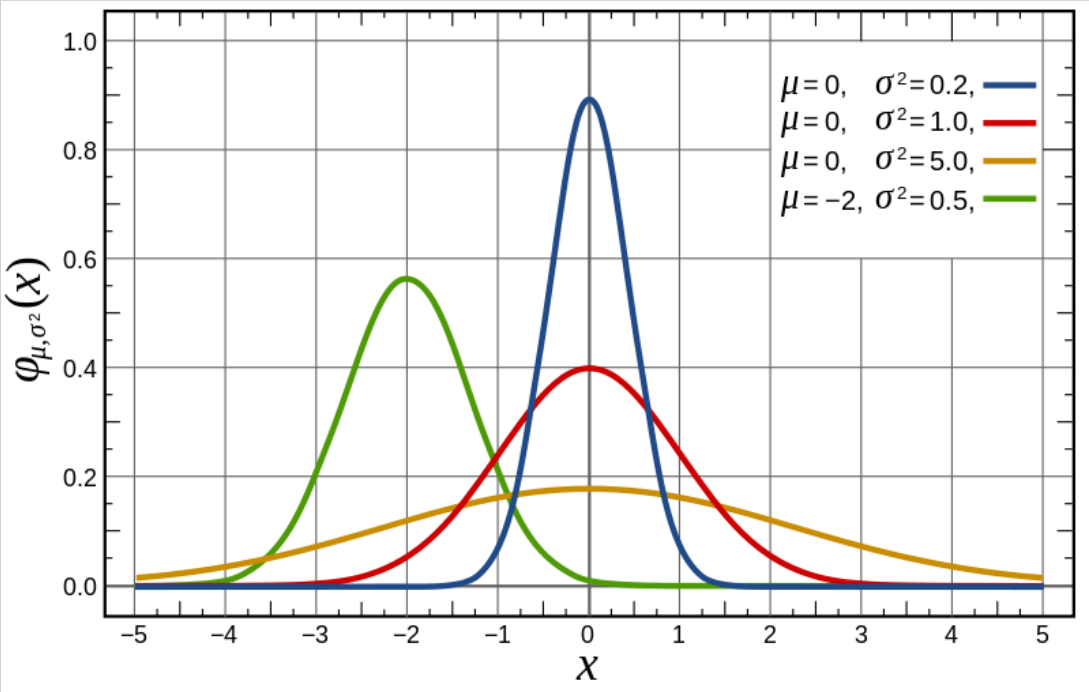


Figura 4.3: Representación gráfica de la función de densidad de probabilidad de la distribución normal.

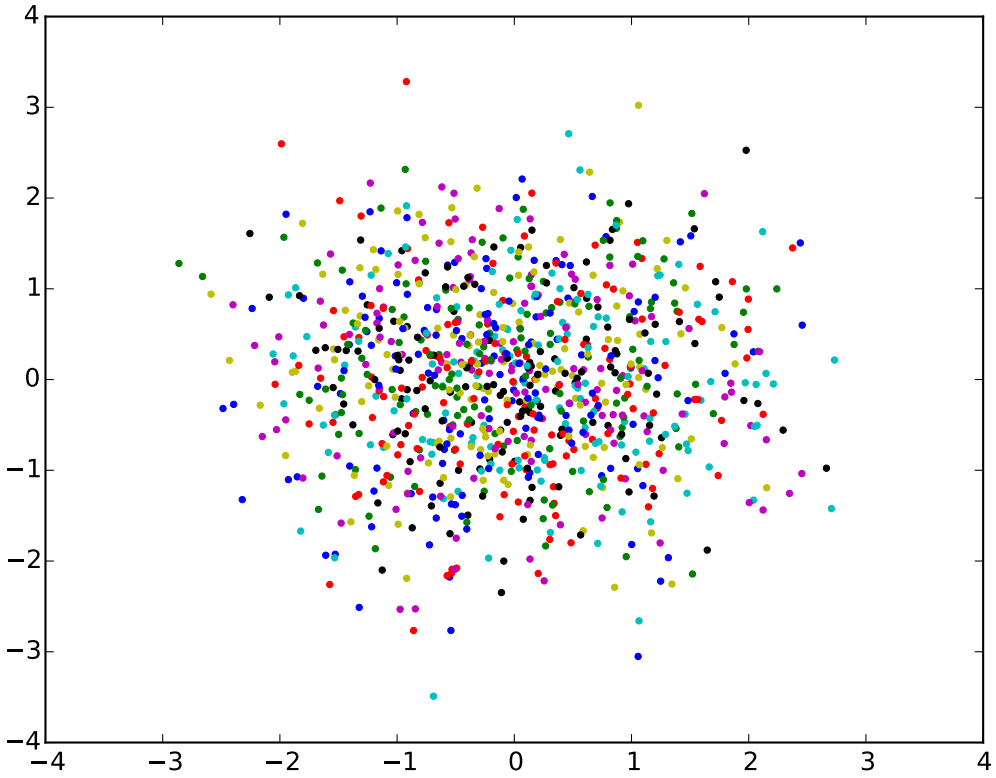


Figura 4.4: Ej. de distribución normal

factor multiplicativo del intervalo. Los puntos son equidistantes entre sí y abarcan todo el intervalo. En la fig. 4.5 se aclara como sería el resultado final con una representación gráfica de 100 puntos en el intervalo 2-dimensional $[0, 1] \times [0, 1]$.

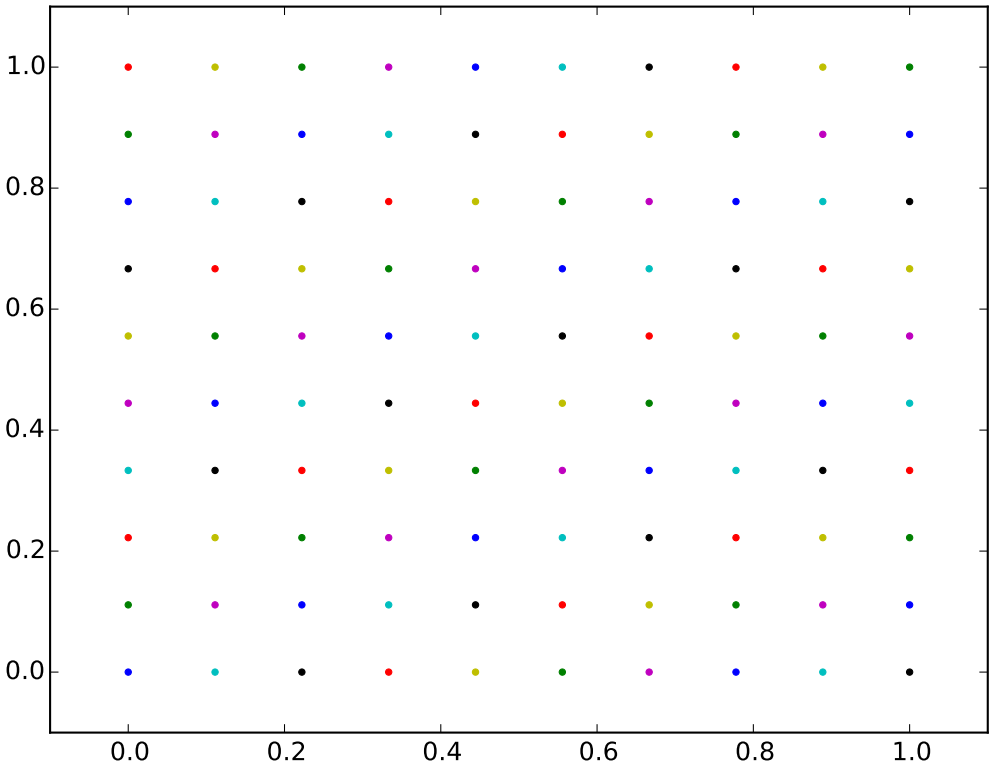


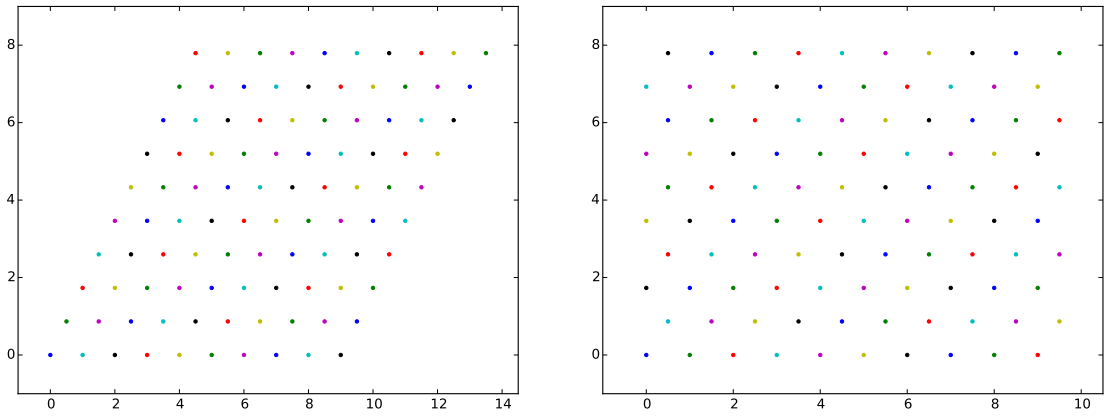
Figura 4.5: Ej. malla cuadrada.

- **Malla triangular**. Esta última estructura es un poco más compleja de generar que la anterior. En ella, todas las posiciones de los puntos, son combinaciones lineales de los vectores $p = (1, 0)$ y $q = \left(\frac{1}{2}, \frac{\sqrt{3}}{2}\right)$ [15]

  En la figura 4.6, se muestran dos ejemplos de malla triangular. El primero (subfig. 4.6a) se trata de una malla triangular "inclinada" formada por 100 puntos y la segunda (subfig. 4.6b) se trata de una malla triangular "vertical", de 100 puntos también. Las distintas configuraciones se consiguen modificando las combinaciones lineales que se realizan de los vectores p y q.

Como se comentaba anteriormente, debido a que el objetivo del generador es crear grafos para poder simular entornos complejos (cercanos al caso real) se ha distinguido desde un primer momento nodos **clientes** y **puntos de acceso o APs**. Por lo tanto, esta distinción ha permitido realizar con los métodos anteriores de generación de posición, una serie de combinaciones que se describen a continuación:

- *random-random*. Es el caso en el que los APs y los clientes se distribuyen por el plano utilizando una distribución uniforme continua. Sin embargo, mientras los APs utilizan una distribución uniforme



(a) Ej. malla triangular "inclinada". (b) Ej. malla triangular "vertical".

Figura 4.6: Ej. de casos de mallas triangulares.

$U(0,1)$ en cada coordenada, los clientes utilizan una $U(0-d_{ap}, 1+d_{ap})$ en cada coordenada, donde d_{ap} corresponde al radio de cobertura de los APs. De esta formal, todos los APs pueden tener clientes en cualquier parte dentro de su radio de cobertura.

- *square-random*. Caso en el que los APs se posicionan creando una malla cuadrada en el intervalo $[0,n] \times [0,n]$ donde n es un factor multiplicativo y los clientes se distribuyen por el plano utilizando una distribución uniforme continua. Los clientes como en el caso anterior, utilizan una $U(0-d_{ap}, n+d_{ap})$ en cada coordenada, para que todos los APs puedan tener clientes en cualquier parte dentro de su radio de cobertura.

- *tri-random*. Caso en el que los APs se posicionan creando una malla triangular y los clientes se distribuyen por el plano, utilizando una distribución uniforme continua. Los clientes, como en el caso anterior, utilizan las coordenadas máximas y mínimas alcanzadas por los puntos, y utilizan $U(x_{min}-d_{ap}, x_{max}+d_{ap})$ para la coordenada x, y $U(y_{min}-d_{ap}, y_{max}+d_{ap})$ para la coordenada y, para que todos los APs, puedan tener clientes en cualquier parte dentro de su radio de cobertura.

- *random-normal*. Es el caso en el que los APs se distribuyen por el plano utilizando una distribución uniforme continua y los clientes, una distribución normal centrada en el AP correspondiente y con desviación estándar configurable.

- *square-normal*. Caso en el que los APs se posicionan creando una malla cuadrada en el intervalo $[0,n] \times [0,n]$ donde n es un factor multiplicativo y los clientes, una distribución normal, centrada en el AP correspondiente y con desviación estándar configurable.

- *tri-normal*. Caso en el que los APs se posicionan creando una malla triangular y los clientes, una distribución normal, centrada en el AP correspondiente y con desviación estándar configurable.

Una vez finalizado el proceso de creación del generador de posición para los nodos, el siguiente paso es modelar un generador de grafos de "interferencia" y de "AP más cercano".

4.1.2 Interferencia y punto de acceso más cercano

Con el objetivo de generar grafos de "interferencia", y grafos de "AP más cercano" se han usado dos implementaciones, que aproximan dichos problemas a modelos existentes de grafos.

4.1.2.1 Punto de acceso más cercano

Para llegar a una solución a este problema, se ha optado por utilizar una modificación del **grafo de vecino más cercano**, cuyas siglas, NNG, corresponden a su nombre inglés *Nearest Neighbour Graph*.

Este grafo, se caracteriza por ser un grafo dirigido y tener n vértices en un espacio métrico (en nuestro caso el plano euclídeo) con aristas dirigidas, del vértice p al q siempre que q sea el vértice más cercano a p [16].

Antes de seguir, es necesario definir el concepto de **triangulación** de un área poligonal, consistente en la división de dicho área en un conjunto de triángulos. Tales triángulos con su unión deben formar el área poligonal; sus vértices deben ser vértices de dicho área y toda pareja de triángulos debe compartir como máximo un lado o un vértice, es decir, ser **disjunta**. Una triangulación, desde el punto de vista de la teoría de grafos, corresponde a un grafo no dirigido y sin aristas múltiples cuyos vértices y aristas equivalen a los vértices de los triángulos y a sus lados respectivamente [17].

El grafo NNG es un subgrafo de la **Triangulación de Delaunay** (DT), que cumple la **condición de Delaunay**: la circunferencia circunscrita de cualquier triángulo no debe contener ningún vértice [18]. La ventaja de esta triangulación es su generación de un número lineal de aristas, permitiendo recorrerlas en tiempo $O(nlogn)$.

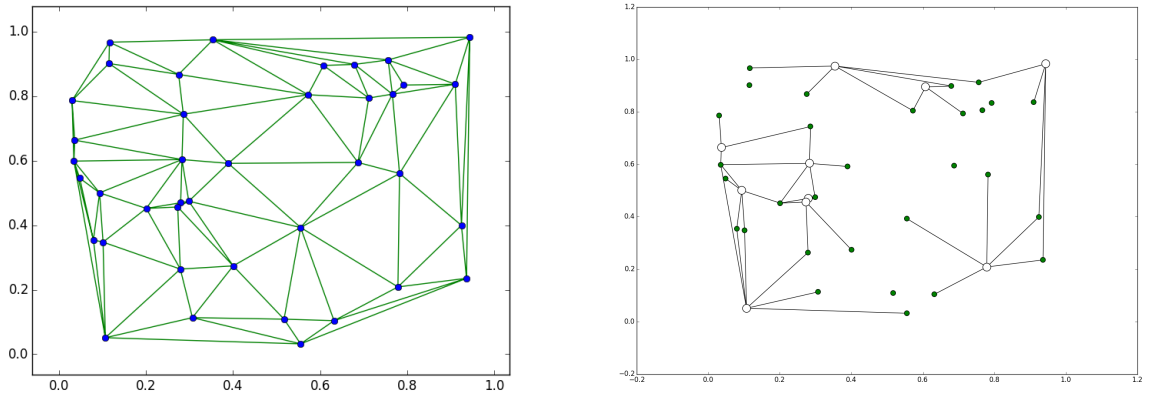
Debido a la diferenciación entre clientes y APs, en la implementación realizada, no se trata de buscar el "punto más cercano" sino el "AP más cercano" a cada uno de los clientes, por lo que una parte de las aristas obtenidas al aplicar la triangulación no serán necesarias. Además, no se ha utilizado un grafo dirigido, sino que, mediante una etiqueta `NearestAP` (AP más cercano) en cada cliente, se identifica el AP más cercano.

Como ejemplo, en la figura 4.7 se muestra una triangulación completa de Delaunay y posteriormente, el grafo NNG modificado obtenido con la implementación.

Es importante indicar que, para adecuar el uso de este generador a varios algoritmos, en la generación de este grafo se toma en cuenta el radio de cobertura del AP, y si un nodo cliente no entra dentro del radio de cobertura de al menos un AP o un nodo AP no tiene clientes, ese nodo será eliminado del grafo. En el caso de la imagen 4.7, se utilizó una versión anterior que no incluye esta modificación.

4.1.2.2 Interferencia

En este caso, se utiliza un grafo del tipo **disco unidad** (*Unit Disk Graph*, o UDG), un grafo de intersección en el que cada vértice presenta un disco o unidad con centro en él mismo y en el que las aristas surgen



(a) Ej. triangulación de Delaunay.

(b) Ej. NNG modificado.

Figura 4.7: Ej. triangulación de Delaunay y NNG modificado.

entre dos vértices si sus circunferencias se intersecan.

Recorrer cada arista, requiere un tiempo cuadrático, característica mejorable utilizando técnicas expuestas en el artículo [19] (esto será comentado en el apartado 6 sobre el trabajo futuro a realizar).

Debido de nuevo a la diferenciación entre clientes y APs, se requerirán dos tipos de grafos UDG: uno con radio $R1$ correspondiente al radio de cobertura de los APs, que simula la comunicación en sentido AP $\rightarrow$ cliente, también conocido como tráfico de bajada, cuyo porcentaje es mucho mayor que el tráfico de subida (cliente $\rightarrow$ AP). Este último tipo de tráfico caracteriza al segundo modelo, con radio $R2$ correspondiente al radio de cobertura del cliente.

En la implementación realizada, los aspectos anteriores han sido tomados en cuenta, sin embargo, a la hora de utilizar el generador de grafos de interferencia para los algoritmos, se ha optado por igualar ambos radios ($R1 = R2$) para simplificar el proceso.

Además, en la implementación ambos modelos de grafo son superpuestos, es decir, como ambos grafos poseen el mismo conjunto de vértices, simplemente se añade en el primero, las aristas extra del segundo. Estas aristas están sujetas a distintas formas de interferencia que se describen a continuación:

- **Interferencia AP-cliente**: Si un AP tiene dentro de su radio de cobertura un cliente no asociado a él, se podrá producir una interferencia por parte del tráfico producido por dicho cliente. Si tomamos que los radios $R1$ y $R2$ son iguales, el AP y el cliente podrán experimentar interferencia. Si el radio del cliente es menor que el del AP, el AP la experimentará pero el cliente no.

- **Interferencia cliente-AP**: Si dentro del radio de cobertura de un cliente hay otros APs además del que está asociado, el cliente puede sufrir una interferencia por parte del tráfico producido por dicho AP. Como en el caso anterior, si igualamos radios, ambos podrán experimentar interferencia, pero si el radio del AP es menor, éste no la experimentará (este último caso es poco común).

- **Interferencia AP-AP**: Si un AP tiene dentro de su radio de cobertura a otro AP, ambos pueden sufrir interferencia producida por el tráfico de ambos canales.
- **Interferencia cliente-cliente**: Si un cliente tiene dentro de su radio de cobertura otro cliente cuyo AP asociado es diferente, se podrá producir interferencia en los canales de subida y de bajada de ambos, si suponemos que tienen el mismo radio de cobertura.

Antes de finalizar este subapartado, hay que aclarar que, en la lista de interferencias anterior, en nigún momento se estableció que la interferencia iba a tener lugar, sino que podría. Esta incertidumbre, es debida a que dependiendo del canal/color en el que esté cada uno, puede que la interferencia se aprecie o sea imperceptible. Además, como último apunte, en los clientes de un mismo AP, aunque sus radios de cobertura colisionen, no existe interferencia entre ellos, debido al uso de protocolos de control de acceso al medio, como por ejemplo CSMA-CA [20].

A continuación, en la figura 4.8 se muestra un ejemplo de grafo UDG con 75 clientes y 25 APs, con posición generada a partir de la configuración *random-random* y con radio de cobertura $R1 = R2 = 0.2$. Se puede observar que no hay en la representación 100 nodos en total y esto es debido a que, aquellos **nodos sin vecinos**, sean APs sin clientes o clientes sin APs asociados, son eliminados del grafo al no ser útiles para la experimentación.

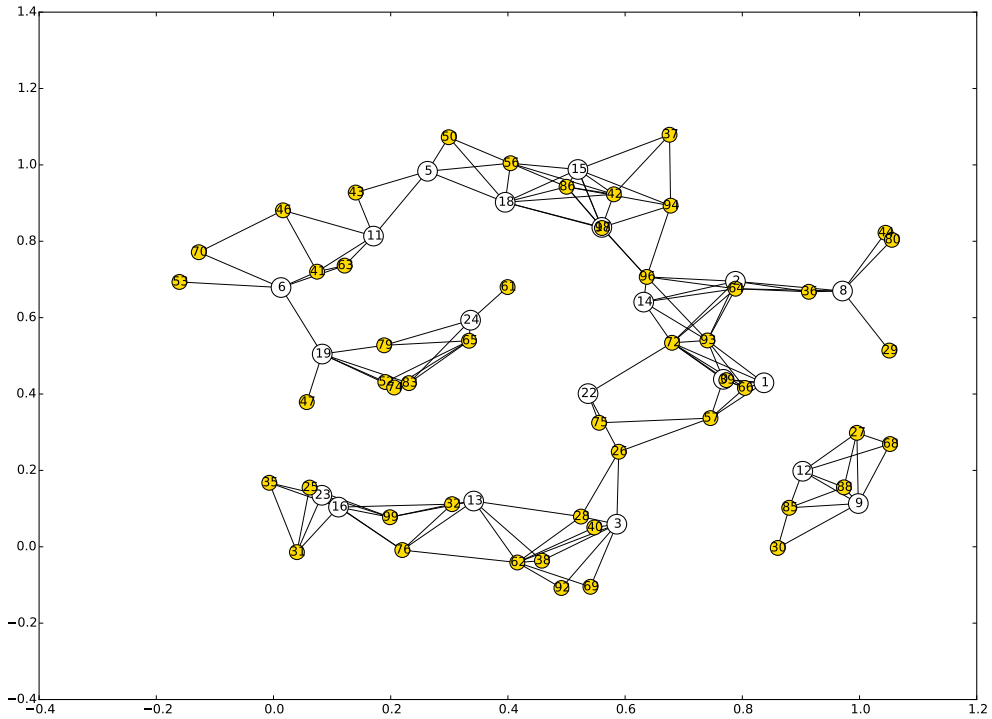


Figura 4.8: Ej. de grafo UDG.

Apartado 5

Algoritmos implementados

Como se comentó anteriormente, no existe el problema de asignación de frecuencia como tal, sino que existen multitud de modelos, que se clasifican según su complejidad y su cercanía a los modelos reales de redes. En el caso de este TFG, se han estudiado e implementado varios algoritmos, que a continuación se pasarán a describir siguiendo un orden de mayor a menor abstracción hacia el caso real.

5.1 Algoritmo `LCCS`

El algoritmo `LCCS` (*Least Congested Channel Search*), es un algoritmo, orientado a ser usado por dispositivos de comunicación (ej. puntos de acceso) en entornos con redes inalámbricas multicanal [21].

5.1.1 Descripción

En el algoritmo `LCCS` cada punto de acceso (AP, *Access Point*) de forma autónoma, busca el canal con menos carga, es decir, con el menor número de clientes. Cuando un punto de acceso termina la búsqueda, pasa a operar en ese canal hasta que la siguiente búsqueda revele otro canal menos congestionado.

Para alcanzar este objetivo, cada AP sigue los siguientes pasos:

1. Búsqueda de balizas (*beacons*) en cada canal, las cuales individualmente representan a un AP vecino.
2. Determinación del número de clientes de cada AP, a través de la información proporcionada por la baliza.
3. Cálculo del número de clientes en cada canal.
4. Selección del canal con menor número de clientes.

Sin embargo, siguiendo este criterio el algoritmo se enfrenta al problema de la afirmación apresurada de que cada cliente genera la misma carga o cantidad de tráfico. Además, LCCS falla a la hora de detectar interferencias entre clientes de dos APs que no interfieren entre ellos y sólo detecta interferencia entre APs, impidiendo reutilizar canales basándose en la interferencia entre clientes [24].

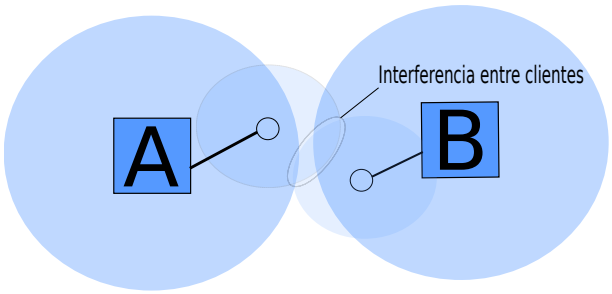


Figura 5.1: Interferencia entre clientes.

5.1.2 Simulación

Debido a que el algoritmo LCCS no está definido ni es parte de ningún estándar, hay ciertas incógnitas, como el orden en el que los APs aparecen en la red que resultan bastante problemáticas. Como se trata de un algoritmo pensado para estar continuamente funcionando y reaccionando a posibles cambios en la topología de red, es necesario solucionar dichas incógnitas de alguna forma.

Para ello, se ha creado una implementación en Python de este algoritmo que funciona de la siguiente manera:

1. En una lista, se guardan los identificadores de los APs y acto seguido, se aleatoriza el orden de éstos (e.g. $[0, 1, 2, 3, 4] \rightarrow [2, 4, 3, 1, 0]$)

2. Se recorre la lista anterior mediante un bucle, y cada iteración consta de los siguientes pasos que afectan a un determinado AP:

   a) Elijo el color/canal con menor número de clientes asignados.

   b) Sumo al número de clientes del canal elegido, el número de clientes del AP en cuestión.

   c) Fin de iteración.

   Al finalizar todas las iteraciones, pueden ocurrir dos eventos:

   a) Se finaliza la ejecución del algoritmo, dejando esta ordenación como la resultante.

   b) Realizamos otra ordenación aleatoria de la lista y volvemos al paso 2. Este último paso se repetirá hasta que se satisfaga una condición (en nuestro caso, cuando llega a 0 un contador que se decrementa 1 cada vez que se termina una ejecución).

La implementación de este algoritmo será utilizada para realizar la coloración inicial de *Hminmax* y *Hsum*, y compararla con los mismos utilizando otras asignaciones iniciales de colores.

5.2 Modificación algoritmo DSATUR

El algoritmo **DSATUR**, está orientado a solucionar el problema de la coloración de grafos de una forma eficiente. La modificación realizada, difiere del algoritmo original en el paso de selección de color para un vértice.

Este caso está bastante orientado al problema matemático de la coloración de grafos y no hacia una aplicación en redes, por lo que esta tratado desde un punto de vista abstracto, sin analogías cliente-vértice e interferencia-arista.

5.2.1 Base teórica

En el artículo de D.Brélaz [25] se proponen varias heurísticas como solución al problema de la coloración apropiada de grafos o *proper graph coloring*, basada en la evitación de aristas monocromáticas.

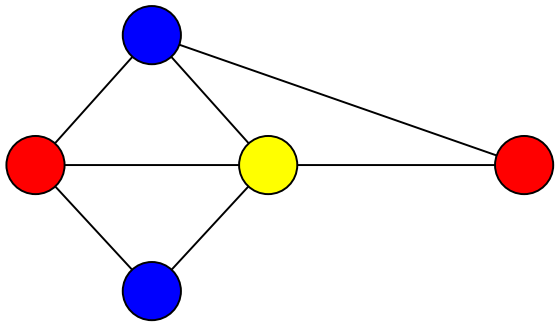


Figura 5.2: Ej. de coloración apropiada de grafos.

La heurística conocida como el algoritmo DSATUR, recibe su nombre del **grado de saturación** de un vértice, definido como el número de colores diferentes a los que el vértice es adyacente, es decir, el número de colores diferentes que tienen sus vecinos. Como se comentó anteriormente, la modificación del algoritmo, solo difiere en el paso de selección de color para el vértice en el cual hace uso de una matriz $k \times k$ de utilidad entre colores denominada W, donde k es el número de colores. Cada elemento de la matriz, representa la utilidad entre los colores i y j y sigue un decaimiento exponencial de base 2 como muestra la siguiente ecuación: $(W_{ij}) = \frac{1}{2^{|i-j|}}$.

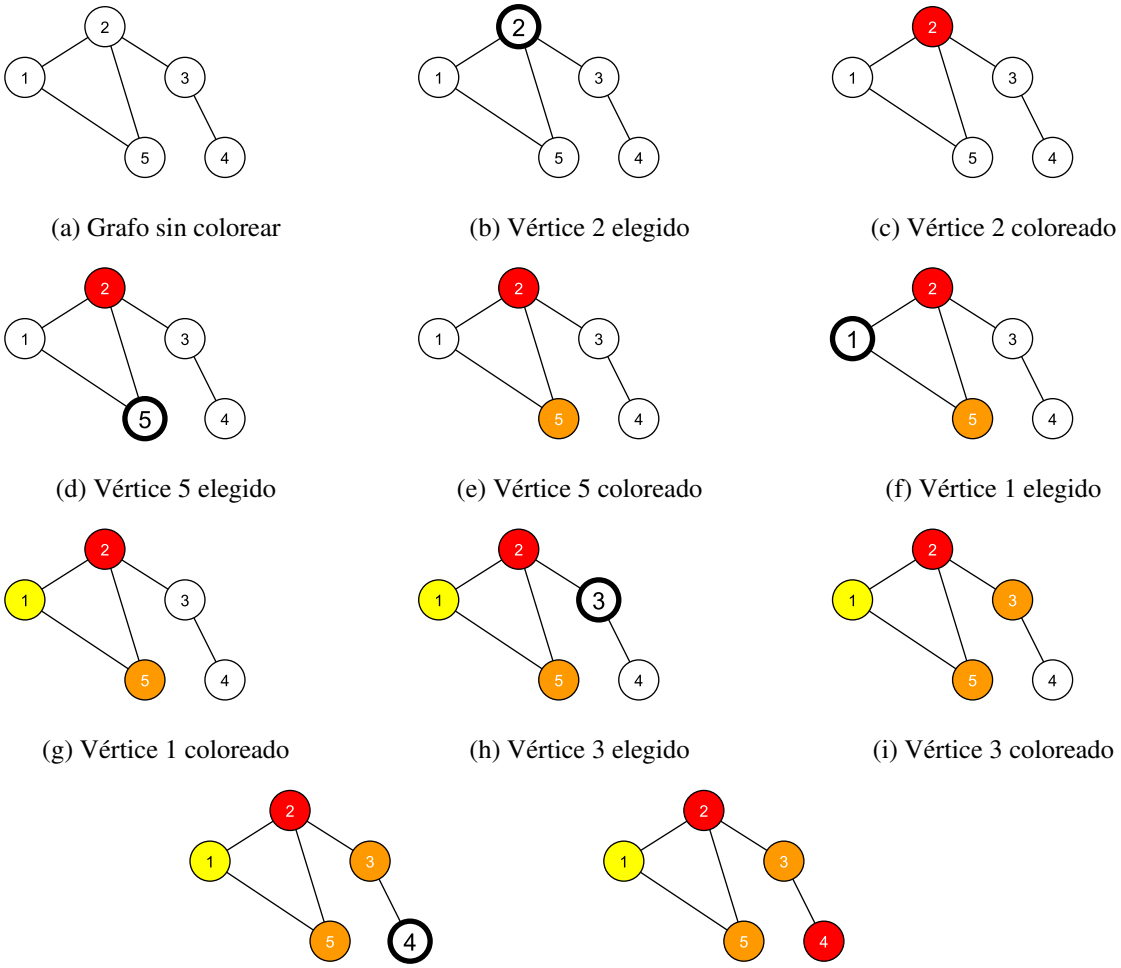
En el siguiente apartado, se procedera a describir el funcionamiento de ambos algoritmos, paso a paso.

5.2.2 Descripción DSATUR

El algoritmo heurístico DSATUR lleva a cabo los siguientes pasos para llevar a cabo una coloración apropiada del grafo en cuestión, al que llamaremos G:

1. Ordenar los vértices según su grado, en orden descendente.
2. Colorear el vértice de mayor grado con el primer color disponible (el numerado con el menor valor).
3. Elegir el vértice con mayor grado de saturación. Si hay varios, se elige cualquier vértice con grado máximo entre los vértices no coloreados.
4. Colorear el vértice elegido con el menor color disponible (si no se puede se añade un color).
5. Repetir los pasos anteriores a partir del 3, hasta que todos los vértices estén coloreados.

A continuación, se muestra un ejemplo simple que permite entender claramente el algoritmo:



(a) Grafo sin colorear (b) Vértice 2 elegido (c) Vértice 2 coloreado

(d) Vértice 5 elegido (e) Vértice 5 coloreado (f) Vértice 1 elegido

(g) Vértice 1 coloreado (h) Vértice 3 elegido (i) Vértice 3 coloreado

(j) Vértice 4 elegido (k) Vértice 4 coloreado

Figura 5.3: Ej. del algoritmo DSATUR

Inicialmente, tenemos un grafo G sin colorear con 5 vértices, con aristas entre ellos como se muestra en la fig. 5.3a. El primer paso, consiste en ordenar los vértices de mayor a menor grado, dando como resultado el siguiente orden: $\{2, 5, 1, 3, 4\}$ (como los nodos 1, 3 y 5 tienen el mismo grado, se han ordenado de esa forma, pero podrían ordenarse de cualquier otra). Se puede observar que el vértice con mayor grado es el 2, por lo que se elige y se colorea con el color de menor índice disponible (fig. 5.3b y fig. 5.3c respectivamente).

Con el vértice 2 coloreado, elegimos el vértice con mayor grado de saturación. Como hay un empate entre los vértices 1, 3 y 5, se elige el vértice no coloreado con grado mayor. Como también hay empate entre los vértices anteriores, se elige uno aleatoriamente (en nuestro caso el 5) y se colorea, con el color de menor índice disponible (fig. 5.3d y fig. 5.3e respectivamente). Como no pueden existir aristas monocromáticas, se debe añadir un color a la lista de colores para colorearlo.

Como siguiente paso, se repite la búsqueda del vértice con mayor grado de saturación, que en este caso resulta ser el 1, por lo que se colorea (fig. 5.3f y fig. 5.3g respectivamente) con un color nuevo debido a la misma razón anterior.

A continuación, hay un empate en la búsqueda del vértice con mayor grado de saturación, entre los vértices 1, 2 y 5, por lo que se elige el vértice no coloreado con mayor grado, que resulta ser el 3, que se colorea con el 2º color de la lista, ya que es el de menor índice disponible (fig. 5.3h y fig. 5.3i respectivamente).

Finalmente, como sigue habiendo un empate en el grado de saturación entre los vértices anteriores, se elige el vértice 4 debido a poseer el máximo grado entre los no coloreados y se colorea con el primer color, ya que es el de menor índice disponible (fig. 5.3j y fig. 5.3k respectivamente). Como todos los vértices han sido coloreados, aquí finaliza la ejecución del algoritmo, dando como resultado que para 5 vértices se utilizan 3 colores.

5.2.3 Descripción modificación DSATUR

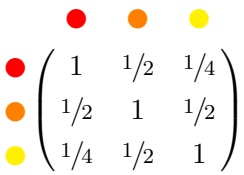
En el algoritmo DSATUR (5.2.2), en su paso 3, se elige el vértice con el grado de saturación mayor y si no hay empate, se le asigna el color de índice más bajo en la lista de colores que, en caso de no existir, provoca la adición de un nuevo color a ésta.

La modificación elegirá el mismo vértice, pero para decidir qué color le asigna tendrá en cuenta la **utilidad**. Es decir:

- Si no puede asignarle ninguno de los colores usados hasta el momento, usará un color nuevo que añadirá a la lista (como en el DSATUR original).
- En caso contrario, para cada uno de los colores que puede usar el nodo calculará qué utilidades tendrían las aristas incidentes a él, y coloreará el nodo con el color que dé la mejor utilidad.

Para el cálculo de la utilidad, se utilizará la matriz W comentada anteriormente y se tomará como mejor utilidad el resultado menor. Para aclarar esta definición, se retoma el ejemplo del caso anterior.

Todo transcurre igual que en el ejemplo anterior hasta la elección del vértice 3. En el caso anterior, este se colorea con el color con menor índice disponible que resulta ser el 2º (naranja). Sin embargo, en la modificación habría que calcular la utilidad de los colores disponibles (el 2º y el 3º, naranja y amarillo respectivamente). Disponemos para este fin de la matriz W siguiente:



$$\begin{pmatrix} 1 & 1/2 & 1/4 \\ 1/2 & 1 & 1/2 \\ 1/4 & 1/2 & 1 \end{pmatrix}$$

Para el color naranja, tenemos una utilidad con respecto al rojo de ½ y para el amarillo de ¼, por lo que el color con menor utilidad, y por tanto el elegido para colorear el vértice 3 será el amarillo.

Finalmente en el caso del vértice 4, se colorea con el color rojo, al igual que en el ejemplo anterior, ya que por casualidad es el color con menor utilidad con respecto al amarillo.

En el siguiente subapartado se explicarán las simulaciones llevadas a cabo con los dos algoritmos.

5.2.4 Simulación

Se han realizado implementaciones en Python de ambos algoritmos, utilizando el módulo `greedy_coloring` de **NetworkX** [26]. Para contrastar los algoritmos con algún método diferente se ha elegido una coloración aleatoria, que elige el número máximo de colores a partir del grado medio del grafo y colorea los nodos siguiendo una distribución uniforme discreta de los colores en el intervalo discreto $[0, col_{max})$.

Para estudiar ambos algoritmos implementados, se ha conducido una serie de experimentos con las siguientes características:

- 2 tipos de grafos:
  - Grafos generados a partir del modelo **Erdős–Rényi** (se ha utilizado la implementación del *package* NetworkX `erdos_renyi_graph`)
  - Grafos generados a partir del modelo **Barabási-Albert** (se ha utilizado la implementación del *package* NetworkX `barabasi_albert_graph`)
- Como parámetros de creación de los grafos, se han utilizado las distintas combinaciones de los siguientes:
  - En el caso general, se ha utilizado un número de nodos $n = \{10, 20, 40, 80\}$
  - Para los grafos **Erdős–Rényi**, se ha utilizado un valor de probabilidad de creación de arista $p = \{0.1, 0.3, 0.5, 0.7, 0.9\}$.
  - Para los grafos **Barabási-Albert**, se ha utilizado un valor para el número de aristas a conectar a un nodo nuevo desde los nodos anteriores $m = \{1, 2, 3, 4, 5\}$

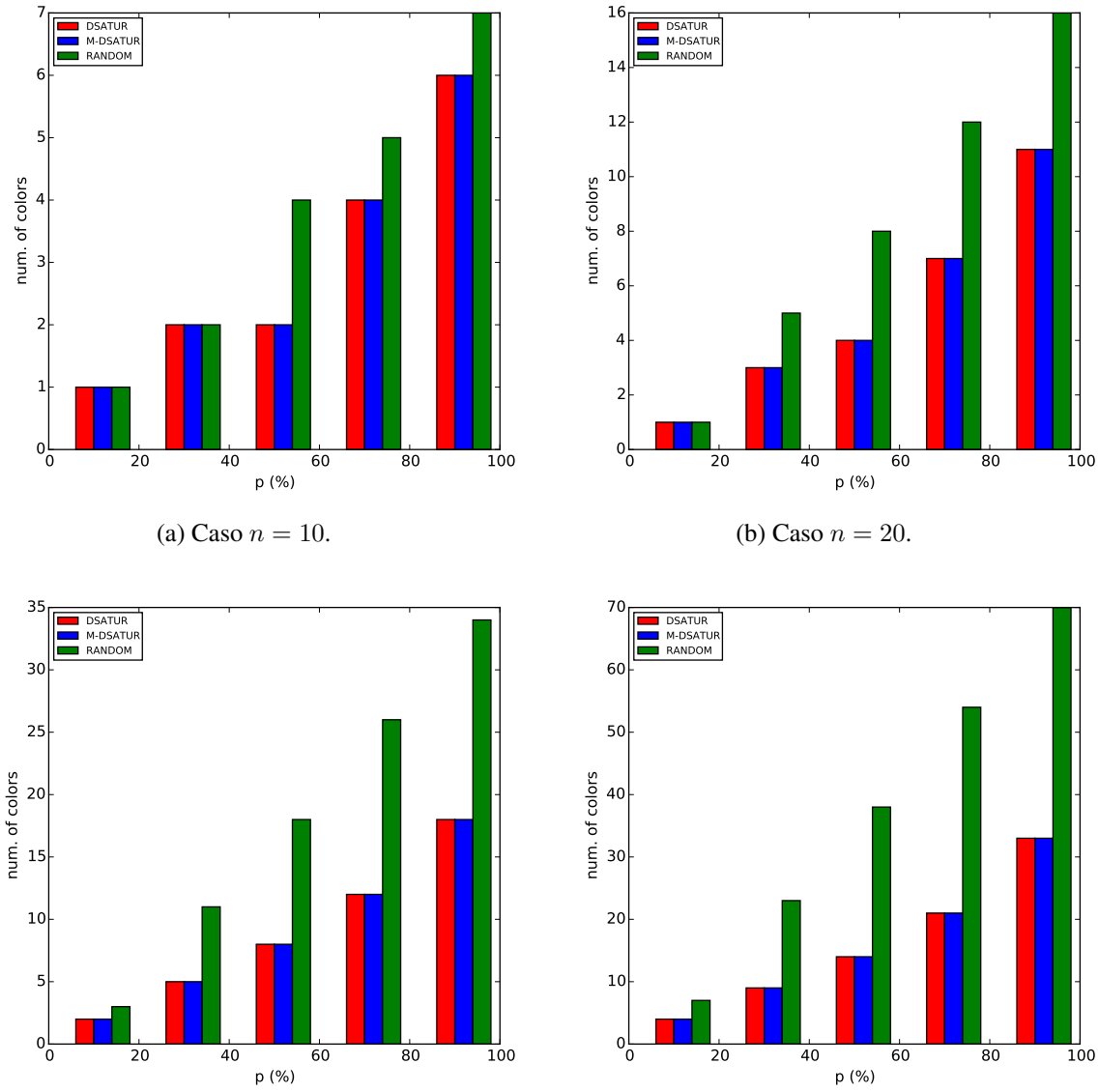
Además, debido a que los algoritmos utilizados para la creación de grafos no son deterministas, se han realizado 15 ejecuciones por combinación. para realizar un estudio más preciso. A continuación se listarán las características a estudio y su notación.

Antes de introducir los parámetros resultantes estudiados, se explicará la función de utilidad usada. Esta, consiste en sumar para cada nodo, la interferencia cocanal con cada uno de sus vecinos y guardar dicho valor. En el caso del algoritmo DSATUR, esta función no tiene ningún papel, sin embargo, en la modificación realizada, es utilizada a la hora de decidir el color de cada nodo en ciertas ocasiones (explicado en el apartado anterior)

- Número medio de colores: $color_{avg}$
- Utilidad media por nodo: $util_{avg}$

5.2.4.1 Grafos Erdős–Rényi

A continuación, se expondrán los resultados obtenidos para grafos generados mediante el modelo Erdős–Rényi.



(a) Caso $n = 10$.

(b) Caso $n = 20$.

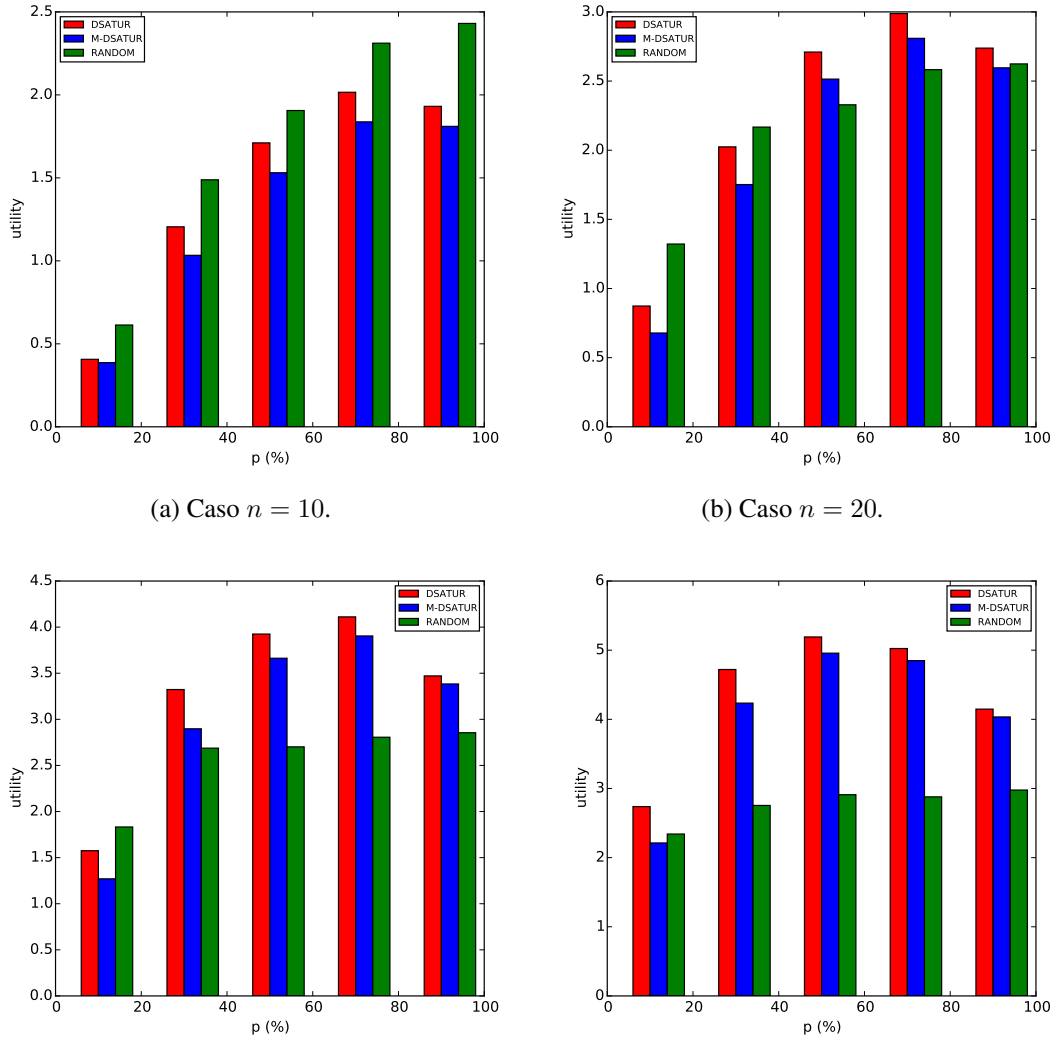
(c) Caso $n = 40$.

(d) Caso $n = 80$.

Figura 5.4: Número de colores medio utilizado, en grafos Erdős–Rényi.

En primer lugar, en la figura 5.4, se muestran 4 diagramas de barras, correspondientes al número medio de colores utilizados para los distintos casos generados, respecto al valor de la probabilidad de creación de arista p en %.

Contrastando los resultados, se puede observar, que el número de colores utilizados, no se diferencia de forma notable en los casos de DSATUR, y M-DSATUR, ya que el verdadero objetivo de la modificación del algoritmo es minimizar la utilidad. El número medio de colores, lógicamente, aumenta con el aumento de la probabilidad de arista p y con el aumento del número de nodos (n) en el grafo. Además, se puede



(a) Caso $n = 10$.

(b) Caso $n = 20$.

(c) Caso $n = 40$.

(d) Caso $n = 80$.

Figura 5.5: Utilidad media, en grafos Erdős–Rényi.

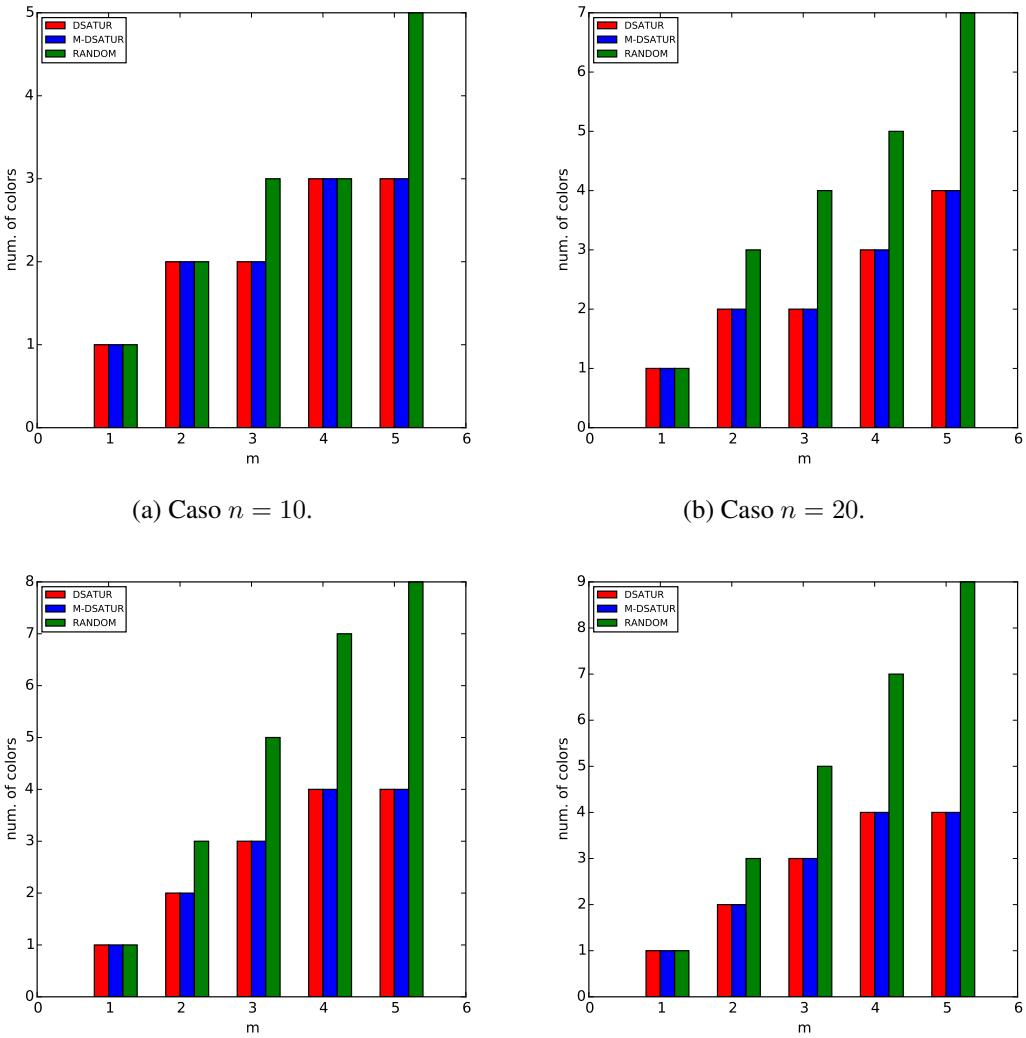
observar como la diferencia en el número medio de colores, con el aumento del número de nodos, aumenta considerablemente en el caso aleatorio con respecto a los algoritmos implementados, cuya diferencia sigue siendo pequeña o nula.

Seguidamente, en la figura 5.5, se puede observar la utilidad media por nodo en los 3 algoritmos. Como es normal, la utilidad media por nodo aumenta con el aumento de p, sin embargo con el aumento de n, la utilidad media por nodo en el caso aleatorio, se va disminuyendo hasta ser menor que en los otros dos algoritmos. Esto es debido a que, como se observa en la figura 5.4, el caso aleatorio, utiliza un número de colores medio bastante mayor al utilizado en los otros algoritmos (en algunos casos el doble incluso), por lo que gracias al uso de más colores, la utilidad es minimizada hasta tal punto. Sin embargo, si se observa el caso de $n = 10$, la utilidad es considerablemente mayor en el aleatorio, y puede ser debido a que en este caso, el número de colores medio es igual en los 3 algoritmos o muy similar. Además, se trata de un modelo aleatorio, por lo que aún realizando 15 ejecuciones, estos resultados pueden no ser los definitivos. A pesar de ello, es suficiente este número de ejecuciones para probar que en todos los casos,

la utilidad es mayor en DSATUR que en M-DSATUR y por lo tanto este último obtiene mejores resultados.

5.2.4.2 Grafos Barabási-Albert

A continuación, a semejanza del subapartado anterior, se expondrán los resultados obtenidos para, en este caso, grafos Barabási-Albert.



(a) Caso $n = 10$.

(b) Caso $n = 20$.

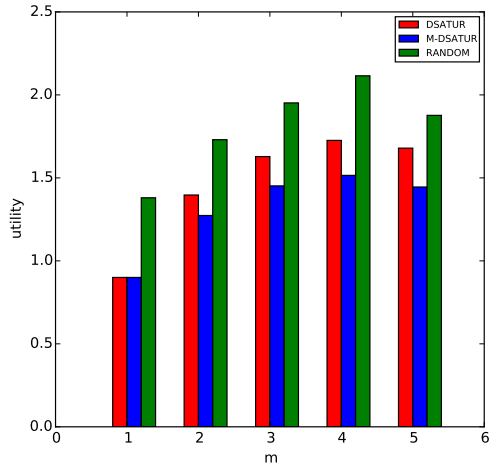
(c) Caso $n = 40$.

(d) Caso $n = 80$.

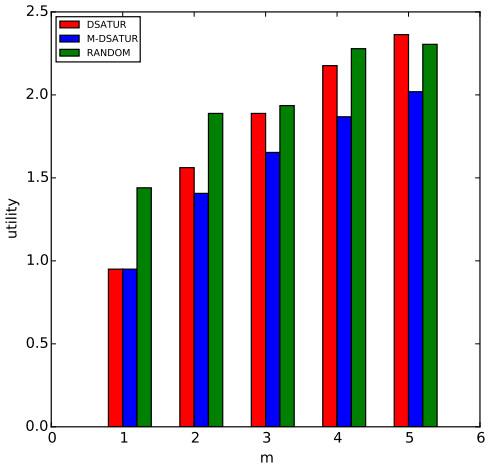
Figura 5.6: Número de colores medio utilizado, en grafos Barabási-Albert.

En primer lugar, en la figura 5.6, se muestran los resultados obtenidos para el número medio de colores, dependiente del número de nodos y el valor del parámetro m, . En el caso de los grafos con 10 vértices (fig. 5.6a), el número de colores es similar en las 3 implementaciones, siendo superior, para el caso aleatorio con $m = 3$ y $m = 5$. Esto puede ser debido al aumento de valor de m, ya que con él, el número de aristas aumenta considerablemente y por consiguiente el grado medio por vértice $\big(num_{aristas} = m(n-m) \big)$.

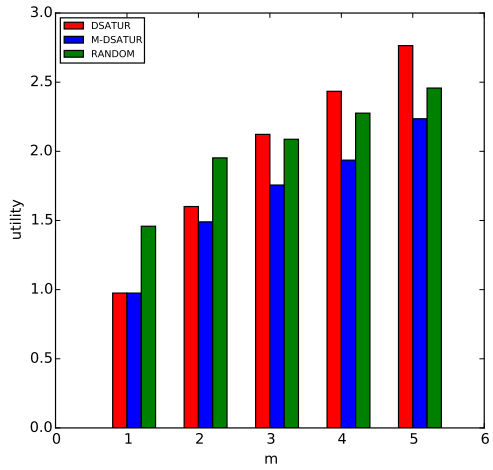
Para los demás valores de n, $color_{avg}$ en DSATUR y M-DSATUR es muy parecido, sin embargo, éste se diferencia en gran medida con el caso aleatorio ya que el número de aristas se incrementa notablemente



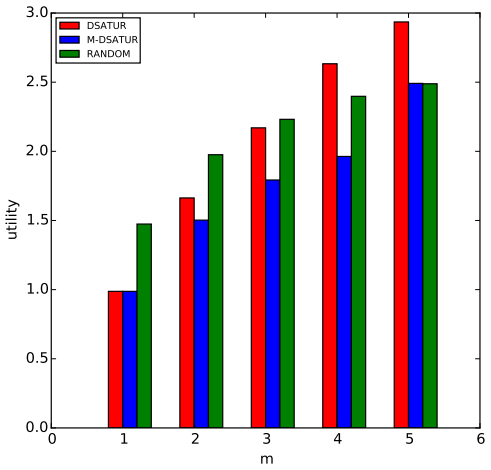
(a) Caso $n = 10$.



(b) Caso $n = 20$.



(c) Caso $n = 40$.



(d) Caso $n = 80$.

Figura 5.7: Utilidad media, en grafos Barabási-Albert.

respecto al valor de $n = 10$ y con él, el grado medio del grafo, parámetro en el que se basa la coloración aleatoria utilizada.

El siguiente parámetro a medir es la utilidad media por nodo ($util_{avg}$) cuyos valores obtenidos se muestran en la figura 5.7. Se puede observar que para el caso $n = 10$, $util_{avg}$ es mayor en el caso aleatorio que en las 2 implementaciones restantes y que entre estas últimas es menor en M-DSATUR que en DSATUR. Estos resultados concuerdan con el objetivo impuesto a la implementación M-DSATUR de **minimizar** este valor.

Para los demás valores de n se obtienen resultados diferentes. Para el caso de $n = 20$, los resultados son iguales que en el caso anterior, salvo para $m = 5$, caso en el que, es menor en el aleatorio que en DSATUR, pero no que en M-DSATUR. Para el resto de casos, el menor sigue siendo M-DSATUR y el de mayor utilidad en casos con valores de m altos, sigue siendo DSATUR. Estos resultados pueden ser debidos a que en el aleatorio se dispone de más colores posibles (en muchos casos, casi el doble) y esto influye en su minimización de la utilidad. Sin embargo, si se observan los casos con $m = 1$ en la fig. 5.6,

todas las implementaciones disponen del mismo número de colores y esto se refleja en la utilidad (fig. 5.7), obteniendo unos valores mayores para el caso aleatorio que para los otros 2, bastante igualados.

5.2.5 Conclusiones

Con los resultados anteriores hemos podido observar que realizando ligeras modificaciones en el algoritmo DSATUR, se puede obtener una mejor utilidad con respecto al algoritmo original en todos los casos. Además, aunque este algoritmo modificado no constituya una solución al problema de asignación de frecuencias en el caso real, debido a su nivel de abstracción, puede utilizarse como base para la implementación de técnicas más complejas.

5.3 Algoritmo *Accumulated-Weight* (AW)

Se trata de un algoritmo distribuido, en el que cada nodo se colorea de forma autónoma, observando sólo a sus vecinos y siguiendo un comportamiento impuesto por una máquina de estados. El algoritmo tiene como objetivo obtener una posible solución al problema de coloración de grafos, para poder aplicarse en redes inalámbricas y habilitar la configuración automática en sus nodos.

5.3.1 Base teórica

Con el fin de alcanzar esta configuración automática de forma robusta y eficiente de manera distribuida, AW se basa en conceptos relacionados con la **emergencia**.

La **emergencia** está inspirada en sistemas naturales distribuidos, como los hormigueros y termiteros, que se basan en la interacción y cooperación de nodos autónomos como método para alcanzar una solución, sin utilizar una estrategia concreta. Los algoritmos emergentes, se caracterizan por una gran **escalabilidad** (habilidad de un sistema para reaccionar y adaptarse sin perder calidad), **robustez** (habilidad de un sistema para hacer frente a errores durante la ejecución) y **descentralización** (cada nodo es independiente y tiene su propia capacidad de procesamiento).

Los algoritmos emergentes son además no deterministas a nivel bajo, debido a las numerosas interacciones aleatorias entre nodos. Sin embargo, a nivel alto, los patrones de comportamiento son predecibles si el algoritmo es estable.

En lo referente a la coloración de grafos, algunas soluciones actuales para grafos en 2D permanecen en la clase de complejidad *NP-hard* (NP-duro o NP-difícil). Para solucionar el problema de esta complejidad, se han desarrollado numerosos algoritmos que utilizan técnicas de computación paralela o concurrente. Sin embargo, todas estas soluciones, aunque en su mayoría eficientes, son **centralizadas** en el sentido de que la topología completa del grafo debe conocerse en cada nodo con capacidad de procesamiento. Esto es posible para topologías estáticas, mediante la recolección de toda la información topológica a priori, pero en el caso de topologías dinámicas, no existe ninguna solución eficiente entre los anteriores algoritmos.

Los algoritmos emergentes, en vez de intentar solucionar el problema utilizando una visión globalizada del grafo, utilizan únicamente la vista local de los nodos, información basada en que cada nodo sólo conoce a sus vecinos (número de vecinos, distancia a cada uno de ellos, etc), obtenida mediante técnicas de difusión, conexión directa, o simplemente ”escuchando” los mensajes mandados por sus vecinos. Esta característica permite a estos algoritmos adaptarse mejor en escenarios dinámicos. Además, como la computación está distribuida entre los diferentes nodos, el tiempo de convergencia del algoritmo y la complejidad de la comunicación disminuyen potencialmente [27].

5.3.2 Descripción

AW es un algoritmo emergente basado en la máquina de estados de la fig. 5.8. A continuación, paso a describir dicha máquina de estados y el funcionamiento del algoritmo:

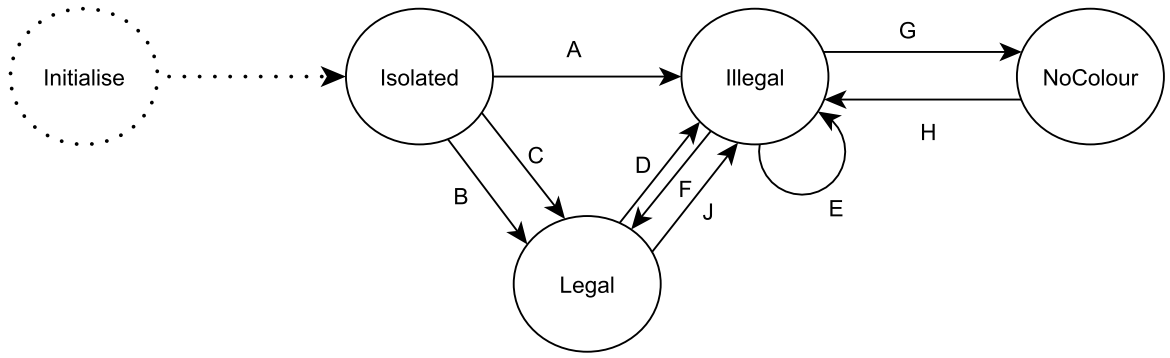


Figura 5.8: Máquina de estados del algoritmo AW.

5.3.2.1 Estados

La máquina de estados citada anteriormente presenta 5 estados diferentes:

- **Initialise**. Estado que no constituye parte del algoritmo, sino que sirve para aplicarle al nodo un retardo inicial aleatorio para aumentar la precisión en la simulación.

- **Isolated**. Estado en el que se encuentra un nodo tras recibir su primer mensaje de estado (*Status message* en el texto original). Dependiendo del resultado de un enfrentamiento, tratado en el siguiente apartado, el nodo pasará a alguno de los estados restantes.

- **Legal**. Estado en el que se encuentra un nodo que gana en un enfrentamiento.

- **Illegal**. Estado en el que se encuentra un nodo que pierde en un enfrentamiento.

- **NoColour**. Estado en el que se encuentra un nodo que pierde en un enfrentamiento, y que no puede colorearse sin interferir con sus vecinos. Se explicará más extensamente en el siguiente apartado.

5.3.2.2 Funcionamiento

El algoritmo tiene 4 estados activos como se indica en la fig. 5.8. Básicamente, el algoritmo consiste en el intercambio de mensajes de estado entre nodos, independientemente del estado en el que se encuentren.

La transmisión de estos mensajes se realiza de forma periódica, con un periodo máximo **M** (1000 ms), valor elegido para evitar exceso de intercambios entre nodos sin comprometer el tiempo de respuesta del sistema a cambios topológicos. Además, un componente aleatorio *r*, donde $0 \leq r \leq 0.1M$, es utilizado para cada nodo con el fin de evitar la simetría en el sistema e imitar el comportamiento natural de los sistemas emergentes, estableciendo el periodo de transmisión (*tx*) continuamente variable en cada nodo, en el intervalo $0.9M \leq tx \leq M$.

El estado **Initialise** no forma parte del algoritmo, como se comentaba anteriormente, por lo que el estado del que parte cada nodo es el estado **Isolated**, en el cual las únicas acciones realizadas son el coloreado con el color más bajo, para asegurar el uso eficiente de los colores, y la espera a recibir un mensaje de estado. Un mensaje de estado, contiene la siguiente información relacionada con su emisor:

- **ID**: Identificador del nodo emisor.
- **Colour**: Color del nodo emisor.
- **State**: Estado del nodo emisor.
- **AccumulatedWeight**: Peso acumulado, factor crucial en la ejecución de enfrentamientos.
- **NeighbourIDs_Array**: Lista que contiene los identificadores de los vecinos del nodo emisor.
- **Neighbour_AgeAdjustedWeights_Array**: Lista que contiene los llamados **vectores de peso** (*weight vectors*, en el texto original) de cada uno de los vecinos del emisor.

Los **vectores de peso** de un nodo, son listas que contienen información de los vecinos de éste. Cuando un nodo recibe un mensaje de estado de otro, el receptor guarda el contenido del mensaje en el vector correspondiente al emisor y establece una marca de tiempo en dicho vector, considerando obsoletos aquellos con una antigüedad de más de $2M$. La información contenida es la siguiente:

- **NeighbourID**: Identificador del vecino.
- **Weight**: Peso del vecino.
- **Colour**: Color del vecino.
- **State**: Estado del nodo emisor.
- **NumberOfNeighbours**: Número de vecinos del nodo vecino.
- **ForceMessageLastSentTimestamp**: Marca de tiempo que indica el instante de tiempo en el que se mandó el último **mensaje de fuerza** (*Force message* en el texto original) a este vecino, relacionado con el estado **NoColour**.
- **Timestamp**: Marca de tiempo.

El peso de un nodo está presente en todos los mensajes que envía. Éste es calculado cada vez que el nodo manda un mensaje de estado o lo recibe, de la siguiente forma:

1. Si el nodo está en estado **Legal**:

    a) Se cuenta el número de vecinos cuyo vector no haya expirado de forma temporal.

    b) Se suma el valor anterior al **peso local aleatorio** (*random local weight* en el texto original), valor real entre 0 y 1.

2. Si el nodo no está en estado **Legal**, simplemente no se toma en cuenta a los vecinos, es decir, el peso será únicamente el peso local aleatorio.

En cuanto un nodo recibe un mensaje de estado, en primer lugar comprueba si el color del emisor coincide con el suyo y si es así, se produce un **enfrentamiento** (*clash* en el texto original), en el cual el nodo compara su **peso acumulado** con el **peso** del emisor. Si el peso del receptor es mayor que el del emisor, el estado del nodo receptor pasará a ser **Legal**; en caso contrario, pasará a ser **Illegal** (transiciones A y B en la fig. 5.8).

Además, si el mensaje recibido no provoca un enfrentamiento, el nodo receptor cambia su estado directamente a **Legal** (transición C).

Una vez en el estado **Illegal**, el nodo cambia su color al más bajo disponible y reinicia un contador inicializado con valor $3M$ que mide el tiempo gastado en dicho estado, desde que se recibió el último mensaje de estado que provoco enfrentamiento.

A partir de este estado pueden ocurrir varias transiciones:

1. El nodo recibe un mensaje de enfrentamiento, con un peso del emisor mayor que el suyo $\rightarrow$ Vuelta al estado **Illegal** (transición E), vuelve a colorearse con el color más bajo disponible y reinicia el contador anterior.

2. El temporizador llega a 0 $\rightarrow$ Paso al estado **Legal** (transición F en la fig. 5.8).

3. El nodo, no puede colorearse sin interferir con sus vecinos (no hay colores disponibles) $\rightarrow$ Paso al estado **NoColour** (transición G).

Además, existe un caso bastante especial en el cual, dos nodos se encuentran en estado **Legal** a la vez, procesando un enfrentamiento recibido anteriormente, con el nodo contrario en estado **Illegal**. En este caso, ambos nodos se proclaman ganadores del enfrentamiento, debido a que en el cálculo del peso en estado **Legal** incluye el número de vecinos, impidiendo que uno de ellos cambie de color. Para asegurar que este problema no persiste durante mucho tiempo, cada nodo llevará un contador de mensajes que implican enfrentamiento. Si este contador es mayor que 2 en un periodo de 3M, el nodo pasará a estado **Illegal** (transición J en la fig. 5.8).

Una vez en el estado **NoColour**, el nodo busca el color más adecuado para elegir, refiriéndose con adecuado a aquel que cause menos problemas en el entorno que le rodea. Esta búsqueda es posible, gracias

al uso de una función de utilidad.

El valor de la utilidad es calculado para cada color, siguiendo las siguientes reglas que en conjunto forman un procedimiento sencillo:

1. Si más de 1 vecino tiene ese color, sumar 5 a la utilidad por cada nodo adicional, es decir, si por ejemplo tenemos 5 vecinos con el mismo color, habrá que sumar a la utilidad de dicho color $4 \cdot 5 = 20$. Usamos esta regla para intentar afectar al menor número de vecinos posible.

2. Por cada vecino que posea el color, se sumará a la utilidad su número de vecinos, para ayudar a minimizar los efectos de este proceso al resto del grafo, ya que si un nodo tiene 100 vecinos y otro 10, es preferible obligar a cambiar de color al que posee 10 vecinos para lograr el objetivo anterior.

3. Por cada vecino en estado **Legal** con el color en cuestión, se sumará 2 a la utilidad, permitiendo preservar la estabilidad en zonas locales.

4. Por cada vecino en estado **NoColour** con el color en cuestión, se sumará 2 a la utilidad, evitando seleccionar dichos objetivos, ya que no mejorarían la situación del nodo.

5. Por cada nodo vecino con el color, si su campo **ForceMessageLastSentTimestamp** del vector de peso correspondiente es menor que M, se sumará 50 a la utilidad; si es menor que 2M, 25 y si es menor de 3M, 10. Esto permite asegurar que el mismo vecino, en un periodo corto de tiempo, no es reiteradamente hecho objetivo de mensaje de fuerza, explicados a continuación.

Tras el cálculo de todas las utilidades, se colorea con el color con utilidad mínima y se envía un mensaje de fuerza a cada vecino con este color. Éstos, al recibir el mensaje de fuerza, pasan al estado **Illegal** y realizan una difusión de mensajes de estado a sus vecinos con su nuevo peso.

Un nodo en estado **NoColour** permanece en dicho estado hasta que un temporizador (con valor indeterminado por el texto original) expira, momento en el cual, el nodo pasa a estado **Illegal** (transición H en la fig. 5.8).

Finalmente, la **convergencia** del algoritmo es alcanzada cuando todos los nodos están en estado **Legal** o **Isolated** y no se produce ninguna transición durante un periodo mayor de 3M.

El establecimiento de esta convergencia esta orientado a poder evaluar el algoritmo mediante una implementación en código. Sin embargo, el algoritmo aplicado a un escenario real y dinámico, puede alcanzar la anterior situación estable y verse seguidamente interrumpida por la llegada o desplazamiento de un nodo.

En el siguiente subapartado, se hablará de la implementación del algoritmo y de las distintas pruebas que se han llevado a cabo, junto a los resultados obtenidos.

5.3.3 Simulación

Con el objetivo de estudiar el algoritmo en cuestión, se ha realizado una implementación de éste en el lenguaje de programación interpretado Python.

Se trata de una implementación en tiempo simulado representado por una variable `tim_tot` perteneciente a la función principal, cuyo valor se incrementa con cada iteración del bucle principal. Una de estas iteraciones se representa de forma esquemática en la figura 5.9. A continuación, se procede a describir más detalladamente esta iteración:

1. El paso **INICIO** en el diagrama representa la personalización de la ejecución, consistente en la inicialización de variables necesarias, la elección de distintos parámetros vía teclado, la generación de los grafos y la creación de la lista de objetos de clase `node`, a la que se añaden n objetos, donde n es igual al número de nodos en el grafo.

2. En el siguiente paso, la ejecución entra en el bucle principal del algoritmo en el cual, mientras no sea alcanzada la convergencia, se comprobará para cada nodo lo siguiente:

   a) ¿El instante de transmisión de mensajes del nodo ha llegado (*t* = *n.Interval*)? Si no es así, se pasará al **paso c**. Si la respuesta es afirmativa, se pasará al **paso b**.

   b) Se calcula un nuevo intervalo de transmisión para el nodo y éste envía los mensajes que correspondan a su estado.

   c) Se comprueba el estado del nodo. Si se encuentra en estado **Illegal** o **NoColour**, se procede a comprobar si ha expirado el temporizador que permite la transición a estado **Legal** (`n.ILT`) y a **Illegal** (`n.NCIT`) respectivamente.

   d) El nodo procesa los mensajes que haya recibido y se incrementan las variables encargadas de simular el tiempo.

   e) ¿El algoritmo ha llegado a converger? Si no es así, se vuelve al paso a. Si ha convergido, se finaliza la ejecución (**FIN**).

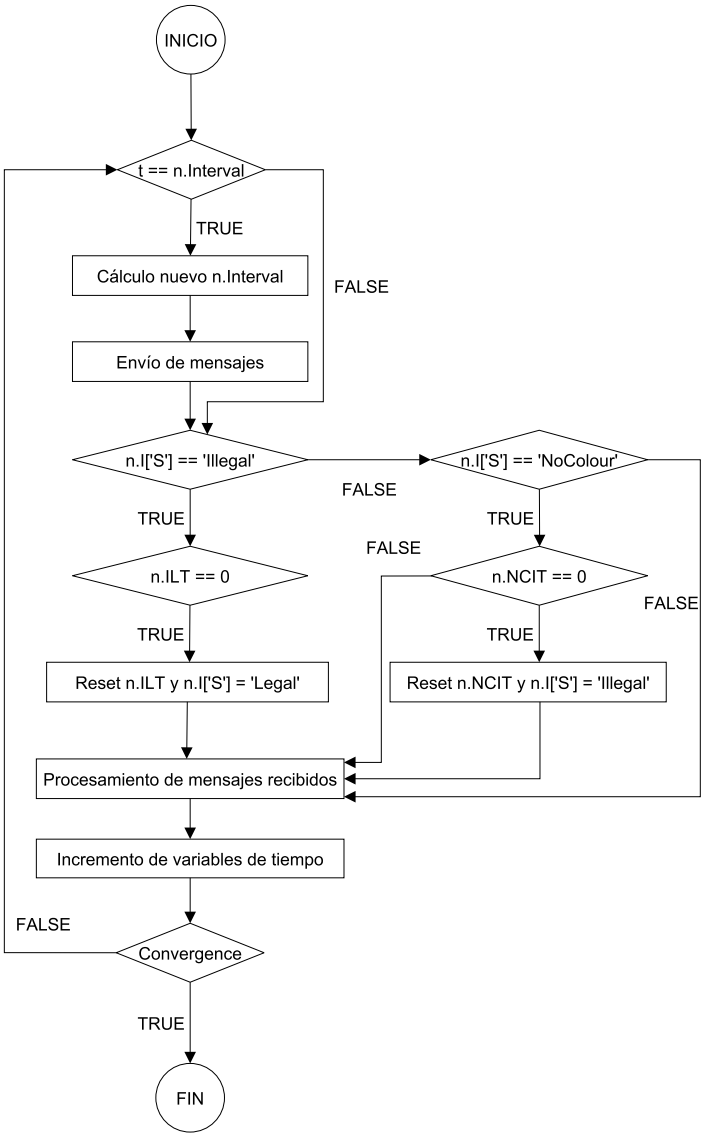


Figura 5.9: Diagrama de flujo de iteración en la implementación del algoritmo AW.

Además, el envío de mensajes, se ha realizado de forma simplificada utilizando dos pilas de mensajes comunes para todos los nodos, una para mensajes de estado (`message_list`) y otra para mensajes de fuerza (`force_message_list`). A los mensajes se les han añadido 2 campos nuevos, `IDo` e `IDd` que contienen el identificador de nodo origen y destino respectivamente, lo que permite a los nodos identificar que mensaje es dirigido hacia ellos. Una vez procesado un mensaje, este es eliminado de la lista, para optimizar el uso de recursos.

En la implementación realizada, dentro de la personalización de las ejecuciones, se han permitido 3 tipos de grafo: el grafo regular aleatorio, obtenido mediante el generador de grafos `random_regular_graph`; el grafo generado utilizando el modelo Barabási-Albert, obtenido mediante el generador `barabasi_albert_graph`; y el grafo resultante del modelo Erdős–Rényi, obtenido mediante el generador `erdos_renyi_graph`. Todos los generadores de grafos citados, pertenecen al *package* NetworkX 1.9.1 [10].

Las simulaciones realizadas tienen como objetivo estudiar la escalabilidad y la convergencia del algoritmo AW. Para ello, se han utilizado grafos regulares aleatorios, ya que permiten observar como se comporta el algoritmo cuando es incapaz de realizar una coloración en el tiempo dado. Para estudiar la escalabilidad, se utilizan varios valores de grado d del grafo, para observar la semejanza en los resultados, debido a que el algoritmo es **distribuido**. A continuación, se resume el conjunto de pruebas realizadas:

- Un grado por vértice $d \in \{1, 2, 3, 4\}$
- Un número de nodos $nn \in \{10, 20, 40, 80\}$
- Un número de colores máximo $nc \in \{2, 4, 8, 10\}$

El tiempo máximo de convergencia se ha establecido para un primer experimento en 40000. Si una ejecución no converge en ese tiempo se tomará como una **ejecución fallida**, debido probablemente a la falta de colores a la hora de evitar aristas monocromáticas. Otro conjunto de experimentos ha sido realizado, utilizando un tiempo de 120000, para observar la diferencia, ya que puede variar el número de casos que convergen, al tener más tiempo para alcanzar este objetivo. Hay que tener en cuenta que hay algunos casos en los que no influye el tiempo máximo de convergencia, debido a la naturaleza del grafo y del algoritmo ($\nexists$ aristas monocromáticas).

Por lo tanto, realizando 10 ejecuciones de cada conjunto de parámetros, tenemos para cada modelo un total de ejecuciones $total_{exec} = 4 \cdot 4 \cdot 4 \cdot 10 \cdot 2 = 1280$, con un total de 640 grafos generados.

Los grafos generados, son grafos aleatorios regulares, es decir, cumplen las siguientes características:

- Debido a su carácter **regular**, cada vértice en el grafo tiene el mismo grado.
- Debido a que se trata de un modelo **aleatorio**, las aristas entre vértices, se establecen de forma aleatoria siguiendo una distribución concreta (en este caso, internamente, `random_regular_graph` utiliza la función `shuffle` del *package* `random`, la cual se basa en la función `random` del mismo *package* que genera un valor aleatorio uniforme en coma flotante entre 0.0 y 1.0 con 53 bits de precisión [31]).

En los experimentos, como se comentó anteriormente, se estudian la convergencia y la escalabilidad. En las tablas y gráficas utilizadas para mostrar los resultados se utiliza la siguiente notación:

- nn: Número de nodos.
- d: Grado por vértice.
- tt: Tiempo total transcurrido hasta converger.

Para el estudio de la convergencia, se ha medido el tiempo medio de ejecución de todos los casos. En la tabla 5.5 se muestran los resultados. Cada subtabla corresponde a un valor de nn.

Para el caso de $nn = 10$ (5.1) se puede observar que con un $nc = 2$ y un $d = 1$, tt no llega al límite (40000) sino que se fija en 6954. Es claro que el algoritmo converge en todos los casos ya que cada nodo sólo tiene un vecino y hay dos colores. Un ejemplo de este caso se puede observar en la figura 5.10.

Tabla 5.1: $nn = 10$

nc	d	tt
2	1	6954
2	2	30381
2	3	40000
2	4	40000
4	1	6944
4	2	6944
4	3	6952
4	4	10243
8	1	6957
8	2	6947
8	3	6949
8	4	6935
10	1	6645
10	2	6650
10	3	6945
10	4	6947

Tabla 5.2: $nn = 20$

nc	d	tt
2	1	6968
2	2	40000
2	3	40000
2	4	40000
4	1	6962
4	2	6949
4	3	6954
4	4	10263
8	1	6964
8	2	6960
8	3	6959
8	4	6951
10	1	6954
10	2	6948
10	3	6963
10	4	6959

Tabla 5.3: $nn = 40$

nc	d	tt
2	1	6970
2	2	40000
2	3	40000
2	4	40000
4	1	6974
4	2	6970
4	3	6973
4	4	33691
8	1	6969
8	2	6972
8	3	6972
8	4	6966
10	1	6972
10	2	6971
10	3	6971
10	4	6973

Tabla 5.4: $nn = 80$

nc	d	tt
2	1	6981
2	2	40000
2	3	40000
2	4	40000
4	1	6983
4	2	6972
4	3	6976
4	4	20477
8	1	6985
8	2	6972
8	3	6973
8	4	6979
10	1	6981
10	2	6974
10	3	6976
10	4	6974

Tabla 5.5: Resultados correspondientes al tiempo total de ejecución de los distintos casos, con un tiempo máximo de convergencia $t = 40000$

Ahora bien, observando los resultados para $d = 2$, surge un resultado extraño con valor 30381. Este valor simplemente indica que, al recopilar los valores de las 10 ejecuciones correspondientes a este caso, alguna entre ellas logró converger en el tiempo dado y las demás no. Esto es posible ya que, aunque se trate de un grafo de grado 2 en cada nodo y 2 colores posibles, hay una probabilidad de que el nodo se coloree con un color y sus vecinos con el otro, cumpliendo en este caso la evitación de aristas monocromáticas (véase fig. 5.11). Es necesario comentar que como se mostrará en los casos con $nn > 10$, estos casos serán mucho menos probables ya que por ejemplo un número impar de nodos conectados entre sí formando un polígono (e.g. 3 nodos conectados entre sí mediante aristas formando un triángulo) aparecerán con más probabilidad de forma que no sea posible colorear con 2 colores dichos nodos.

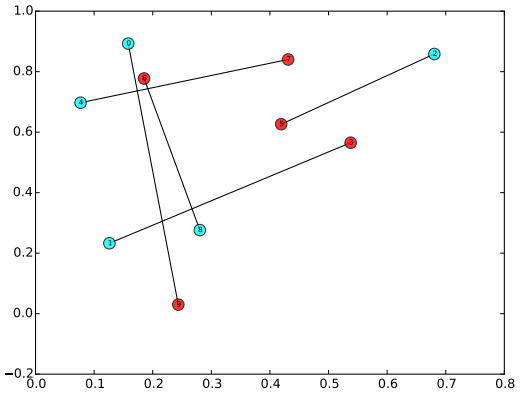


Figura 5.10: $nn = 10$, $nc = 2$ y $d = 1$. Coloración resultante de algoritmo AW.

Para el resto de valores de d, no se da ningún caso en el que converja el algoritmo, ya que aumenta el número de aristas, por lo que la probabilidad de que la colocación de las aristas permita la convergencia disminuirá, ya que la probabilidad de que aparezca un número impar de vértices conectados entre sí aumentará. Además, con mayor d, aparecen casos nuevos en los que se impediría la convergencia (e.g. subgrafos completos de 4 nodos, en el caso de $d = 3$).

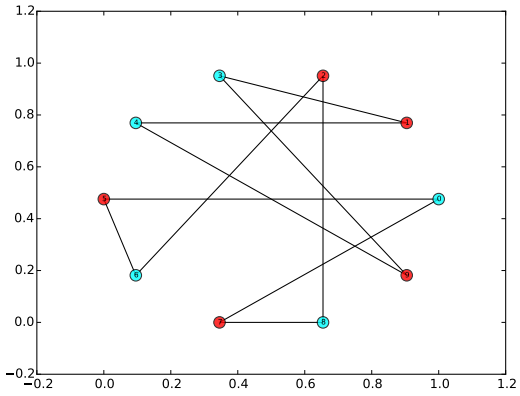


Figura 5.11: $nn = 10$, $nc = 2$ y $d = 2$. Coloración resultante de algoritmo AW.

Pasando al siguiente número de colores, para $nc = 4$, el algoritmo converge en casi todos los casos, ya que con ese número de colores y tan pocos nodos, con un grado menor que 4, hay colores suficientes. Sin embargo, en el caso de $d = 4$, hay varias ejecuciones que no convergen. Sin embargo, observando el valor de tt en proporción con las que convergen son pocas ($tt = 10243$). Esto puede ser debido a que es más difícil crear, de forma uniformemente aleatoria subgrafos que impidan la convergencia en estos grafos con 4 colores (e.g. un subgrafo completo de 5 nodos) que un subgrafo completo de un número impar de nodos.

Los valores de nc restantes, simplemente han sido utilizados para comprobar que el algoritmo funciona correctamente ya que, como es de esperar, si usamos $nc = 8$ o $nc = 10$ para los anteriores valores de d, convergerá siempre, ya que como se explicó en el párrafo anterior, aunque se de un subgrafo completo de 5 nodos (peor caso posible), habrá un color por nodo mínimo. Explicado desde el interior del algoritmo, un nodo nunca podrá entrar en estado **NoColour** (siempre habrá colores disponibles).

En los demás valores de nn $(20, 40, 80)$ la explicación para los distintos resultados según el número de colores es similar ya que, como se comentó anteriormente, el aumento del número de nodos conlleva un mayor número de aristas, lo que se traduce en una mayor probabilidad de que los subgrafos completos aparezcan, provocando que ningún caso con $nc = 2$ y $d = 2$ converja y que en los casos con $nc = 4$ y $d = 4$ haya menos casos en los que se llegue a converger. En este último caso, se puede observar que no hay una relación aparente entre el tiempo medio de convergencia y el número de nodos. La afirmación dada anteriormente de que el aumento de este último valor puede provocar que el algoritmo no converja está ligada al hecho de que, en la creación de los grafos hay un gran factor probabilístico, al distribuirse sus aristas de manera uniformemente aleatoria, por lo que las ejecuciones pueden converger o no.

Con respecto a la escalabilidad, el algoritmo está claro que se comporta de manera distribuida, ya que independientemente del número de nodos y del número de aristas, salvo que no converja, el tiempo de convergencia es muy similar, estableciéndose en el intervalo $[6645, 6985]$.

Además de estas ejecuciones, se han realizado, como se estableció previamente, otro conjunto de experimentos idénticos salvo en el tiempo de convergencia máximo, que se establece en el triple del anterior (120000). Los resultados, mostrados en la tabla 5.10, han sido similares a los anteriores por lo que es posible que el tiempo elegido no haya sido suficiente o simplemente que, debido a que los grafos son generados aleatoriamente, el número de grafos sin casos que impidan la convergencia ha sido similar.

Tabla 5.6: $nn = 10$

nc	d	tt
2	1	6941
2	2	86377
2	3	120000
2	4	120000
4	1	6959
4	2	6949
4	3	6951
4	4	6951
8	1	6949
8	2	6960
8	3	6950
8	4	6951
10	1	6942
10	2	6944
10	3	6944
10	4	6952

Tabla 5.7: $nn = 20$

nc	d	tt
2	1	6956
2	2	120000
2	3	120000
2	4	120000
4	1	6972
4	2	6965
4	3	6959
4	4	29564
8	1	6971
8	2	6953
8	3	6961
8	4	6956
10	1	6960
10	2	6959
10	3	6946
10	4	6946

Tabla 5.8: $nn = 40$

nc	d	tt
2	1	6966
2	2	120000
2	3	120000
2	4	120000
4	1	6976
4	2	6960
4	3	6965
4	4	86090
8	1	6970
8	2	6977
8	3	6969
8	4	6969
10	1	6973
10	2	6971
10	3	6974
10	4	6969

Tabla 5.9: $nn = 80$

nc	d	tt
2	1	6980
2	2	120000
2	3	120000
2	4	120000
4	1	6979
4	2	6980
4	3	6975
4	4	52478
8	1	6985
8	2	6971
8	3	6976
8	4	6976
10	1	6978
10	2	6983
10	3	6979
10	4	6977

Tabla 5.10: Resultados correspondientes al tiempo total de ejecución de los distintos casos, con un tiempo máximo de convergencia $t = 120000$

Para finalizar y como una aclaración, cabe indicar que para generar los experimentos se ha utilizado una versión automatizada de la implementación (denominada `auto_aw`), modificada de forma que no se realice la introducción de parámetros de personalización por la entrada estándar, sino mediante bucles utilizando listas.

5.3.4 Conclusiones

La implementación del algoritmo ha permitido mostrar su escalabilidad y convergencia, mostrando por un lado las ventajas y desventajas de éste en una aplicación real.

En primer lugar, es ventajoso el hecho de que, en los casos que puede converger, lo hace en poco tiempo y sabiendo que se trata de un algoritmo que está ejecutándose continuamente, es buena esta presteza a la hora de llegar a una solución, para así poder enfrentarse más eficientemente a cambios en la red. Además, como se ha observado en el apartado anterior, es escalable y lo muestra con unos tiempos muy similares en casos con número de nodos diferente, permitiendo que la red en la que se use pueda crecer libremente sin ocasionar gran problema en este sentido.

Sin embargo, existen varios problemas que se traducen a desventajas al utilizar este algoritmo en un caso real. El primer problema es que el algoritmo AW realiza un uso "eficiente" de los canales/colores, intentando utilizar el menor número posible. Esta propiedad sería beneficiosa en un caso real si el algoritmo tomase en cuenta la interferencia entre canales como realizan algoritmos más complejos (véase 5.4), sin embargo, al no tomarla en cuenta, los colores utilizados son consecutivos y por tanto los coloreados finales de este algoritmo provocan grandes interferencias. Aparte, existe otro problema, que trata directamente con la convergencia. Existen multitud de casos, como algunos de los que hemos visto

antes, en los que es imposible que converja si se intenta realizar una coloración en la que no existan aristas monocromáticas.

5.4 Algoritmos Hminmax y Hsum

Hminmax y **Hsum** son dos algoritmos orientados a mejorar la utilización del espectro electromagnético en las redes de área local inalámbricas (WLANs, *Wireless Local Area Networks*), basándose en una **variación con peso** del conocido problema de coloración de grafos, que toma en cuenta la interferencia real observada entre canales en estos entornos inalámbricos y su impacto en los usuarios.

Debido a que el problema de la coloración de grafos ponderada es **NP-duro**, ambos algoritmos, presentan una naturaleza distribuida y escalable, que les permite alcanzar un mejor rendimiento que anteriores implementaciones.

5.4.1 Base teórica

Debido a la creciente popularidad de las WLANs, el problema de la **compartición de recursos** o **asignación de canales** entre los distintos APs (puntos de acceso, *Access Points* en inglés), ha adquirido bastante protagonismo, en especial en redes WLAN basadas en el estándar del IEEE 802.11, conocidas generalmente como redes Wi-Fi [28].

Las redes basadas en el estándar 802.11 o redes Wi-Fi, operan en bandas de frecuencia no autorizadas (2.4 y 5 GHz). Cada estándar WLAN (802.11b/g/n) define un número fijo de canales, los cuales representan la frecuencia central. Entre canales, existe una separación de 5 MHz. En el estándar 802.11b, como ejemplo, se definen 14 canales (11 permitidos en EE.UU) cada uno de ellos ocupando alrededor de 30 MHz, 15 a cada lado de la frecuencia central. Esta característica, provoca que existan sólo 3 canales que puedan ser utilizados simultáneamente sin causar interferencia: el 1, 6 y 11.

Hoy en día, en el proceso de asignación de frecuencia en este tipo de redes, se sigue el siguiente procedimiento:

1. Cada AP, comprueba periódicamente el volumen de tráfico en el canal que está utilizando, provocado por otros APs y sus clientes.

2. Comprueba si el volumen es mayor que un umbral.

    a) Si es así, el AP cambia a otro canal menos congestionado.

    b) Si no es así, se vuelve al paso 1.

Esta técnica es conocida como **búsqueda del canal menos congestionado** (*Least Congested Channel Search*, `LCCS`) y es bastante utilizada hoy en día. Sin embargo, con la naturaleza frecuencial **no autorizada** y el **abaratamiento** de los APs, las redes WLANs y por lo tanto el número de APs en un vecindario físico ha ido aumentado progresivamente a gran velocidad y por consiguiente, convirtiendo esta solución al problema de la asignación de frecuencia en una poco eficiente, ya que, con gran número

de APs, la dificultad para resolver este problema aumenta considerablemente.

Los algoritmos **Hminmax** y **Hsum**, ambos escalables y distribuidos, intentan solucionar este problema de asignación de frecuencia, intentado solucionar dos subproblemas, derivados del anterior, que según los autores no han sido tratados en la literatura anterior y que resultan bastante importante a la hora de conseguir una asignación de canales eficiente: **reutilización dinámica de canales** y **asignación de canales ponderada**.

5.4.1.1 Reutilización dinámica de canales

El clásico **problema del terminal escondido**, debido a la presencia de clientes dentro de la región de transmisión de dos APs diferentes, puede ocasionar una disminución del régimen binario de estos clientes. Los algoritmos *Hminmax* y *Hsum*, capturando dinámicamente la distribución de clientes en la red toman ventaja de la ausencia de este problema, ya que si no existen usuarios en la región de transmisión de dos APs diferentes, se puede fomentar la reutilización de un mismo canal.

Lamentablemente, las técnicas utilizadas actualmente, como `LCCS` o estudios de radiofrecuencia, no permiten tomar en cuenta estos casos.

5.4.1.2 Asignación de canales ponderada

El problema de asignación de frecuencia, es típicamente solucionado tomando como modelo el **problema de coloración de grafos mínimo**, en el que el objetivo es colorear todos los APs (vértices) con el mínimo número de colores de forma que no existan aristas monocromáticas. Esta aproximación es correcta cuando el número de posibles canales/colores es infinito o muy alto, de tal forma que se puedan utilizar sin tener en cuenta un límite concreto, sin embargo, en redes inalámbricas reales como por ejemplo en las WLANs definidas por el estándar 802.11b en EE.UU, se comentaba anteriormente que sólo existen 3 canales (1, 6 y 11) que no se superponen, por lo que en este caso es problemático usar el modelo anterior, ya que limitaría el tamaño máximo de una red (e.g. un AP sólo podría estar en el rango de transmisión de otros 2 APs sin que se produjese interferencia alguna).

Los algoritmos estudiados en este apartado, intentan solucionar el problema anterior, utilizando una **variante con peso del problema de la coloración de grafos**, que permite utilizar otros canales aparte de los anteriores, aunque exista interferencia apreciable entre ellos. Para próximos apartados, en los que se tratará más extensamente esta variante, se utilizará el término **canales independientes** para aquellos canales que no interfieren entre sí (e.g. canal 1 y 6) y se denominará interferencia entre canales adyacentes (o cocanal) a aquella causada por la adyacencia de dos APs con canales **vecinos** o **adyacentes** Todo esto, permite a esta solución la configuración automática del proceso de asignación de canales a los APs, al contrario que en estudios de radiofrecuencia, permitiendo enfrentarse a redes dinámicas, en las cuales las condiciones se ven modificadas continuamente.

Posteriormente en la descripción de los algoritmos, se hablará del propósito de utilizar esta variante, que no es otro que minimizar el impacto de poder utilizar canales que interfieren entre sí en APs vecinos.

Tabla 5.11: Ej. empírico de valores de *I-factor*

canal/color	1	2	3	4	5	6	7	8	9	10	11
I-factor	0	0.22	0.60	0.72	0.77	1.0	0.96	0.77	0.66	0.39	0

En el siguiente apartado, se describirá más detallada y formalmente la variación con peso del problema de coloración de grafos mínimo.

5.4.2 Coloración de grafos ponderada

Como se indicaba en el apartado anterior, el problema de asignación de canales o frecuencia para WLANs, puede modelarse como un problema de coloración de grafos mínimo, estableciendo los APs como vértices y la interferencia entre ellos, debido a su proximidad, como aristas. Sin embargo, también se comentaba que, debido al objetivo de colorear el grafo, evitando las aristas monocromáticas y además utilizando el menor número de colores, el problema se complicaba bastante, hasta el punto de considerarse NP-Duro para grafos generales. Además, con el incremento del número de APs, si el número de colores (canales) independientes disponibles no crecía, llegaba un momento en el que no eran suficientes.

Para solventar este problema, y permitir una solución eficiente al problema de asignación de frecuencia, en los algoritmos, se utiliza una **variación ponderada** de la coloración de grafos mínima, que conlleva la aparición de un **peso** en cada arista y una **función objetivo concreta**, es decir, se persigue maximizar o minimizar un valor numérico [11]. En nuestro caso, la contribución a dicha función objetivo está a cargo de las aristas y es mayor conforme su peso aumenta.

A continuación, se formalizará con notación matemática, la variación ponderada de la coloración de grafos y las funciones objetivo utilizadas en los algoritmos.

5.4.3 Definición formal

Se define un grafo de interferencia $G = (V, E)$ como sigue: denotamos con $V = \{ap_1, ap_2, \ldots, ap_n\}$ el conjunto de n APs y (ap_i, ap_j) como la arista entre los APs i y j con $ap_i \neq ap_j$.

Tenemos W como la función de peso en G; $W(ap_i, ap_j)$ indica el peso normalizado de la arista entre dichos APs, que se calcula como el número de clientes asociados a los dos APs (i y j) que se ven afectados si a estos APs se les asignan canales que interfieren entre sí.

El problema de coloración de grafos ponderada se establece como sigue: $C(ap_i), ap_i \in V$ es un mapeo $C : V \rightarrow \{1 \ldots k\}$, donde k indica el número máximo de colores en el estándar 802.11 utilizado. Definimos el término *I-factor* (del inglés *Interference-factor*, factor de interferencia) como la interferencia entre los colores asignados a ambos APs y se denota como $I(ap_i, ap_j)$. Este parámetro puede ser medido empíricamente y en la tabla 5.11, se puede observar los resultados obtenidos para este valor utilizando en el experimento dos APs, uno establecido en el canal 6 fijo y otro recorriendo los canales desde el 1 hasta el 11 [24].

Al producto de $W(ap_i, ap_j) \cdot I(ap_i, ap_j)$ se le conoce como *I-value* (valor I en español) y representa la interferencia total de los clientes atrapados en la zona de solapación entre ambos APs

Existen 2 funciones objetivo, L_{max} y L_{sum}, que utilizarán los algoritmos para mejorar la interferencia cocanal.

- L_{max}. Función objetivo llamada así por el parámetro que se encarga de minimizar, que corresponde al máximo *I-value* en el grafo, es decir, el máximo impacto de interferencia en el grafo:

$$L_{max}(G, C) = \max_{\forall e=(ap_i,ap_j)\in E} I(ap_i, ap_j)W(ap_i, ap_j)$$

- L_{sum}. En este caso el parámetro que se encarga de minimizar es la suma de pesos en todas las aristas, cuyos colores/canales interfieren. Dicho de otra manera, esta función objetivo se encarga de minimizar el efecto total de la interferencia en el grafo a causa de la asignación de frecuencia:

$$L_{sum}(G, C) = \sum_{\forall e=(ap_i,ap_j)\in E} I(ap_i, ap_j)W(ap_i, ap_j)$$

5.4.4 Descripción

A continuación, se realizará la descripción del funcionamiento de ambos algoritmos distribuidos para la asignación de canales en redes inalámbricas WLAN. El primero de ellos, **Hminmax**, no requiere de colaboración entre APs, y tiene como objetivo minimizar la interferencia en el área de cobertura de cada AP (L_{max}). **Hsum**, además, reduce en gran medida el número total de clientes que sufren interferencia en la totalidad de la red (L_{sum}), requiriendo para ello la colaboración entre APs.

5.4.4.1 Hminmax

El primer algoritmo *Hminmax*, no requiere de comunicación entre los APs e intenta reducir la función objetivo $L_{max}(G, C)$ pero de manera local a cada AP, es decir, el algoritmo se ejecuta de forma distribuida en cada uno de ellos y se encarga de minimizar la interferencia en su arista más conflictiva ($L_{max}(G, C, ap_i)$). El procedimiento que sigue para este fin, se resume en el siguiente pseudocódigo:

1. **Inicialización**: En este paso se realizará la coloración inicial del grafo, que podrá ser cualquiera, desde una coloración uniforme de todos los vértices con el mismo color, hasta la utilización de LCCS para este fin.

2. **Optimización**: En este paso se mejora el coloreado de una forma incremental, intentando para cada AP minimizar la interferencia en su arista de mayor conflicto, es decir, minimizando ($L_{max}(G, C, ap_i)$).

   a) Para cada color, se calcula $H(c)$ que corresponde al peso máximo entre las aristas de ap_i en las que ap_j tiene el color c. Se define de la siguiente manera:

$$H(c) = \max_{\forall e=(ap_i,ap_j)\in E \wedge C(ap_j)=c} I(ap_i, ap_j)W(ap_i, ap_j)$$

   b) Se elige el color c' tal que $H(c') = \min_{c=1\ldots k} H(c)$.

   c) Se establece finalmente c' como el nuevo color de ap_i.

5.4.4.2 Hsum

Hsum es el último algoritmo de este apartado y se caracteriza por ser un algoritmo más completo que el anterior, ya que reduce L_{max} y adicionalmente minimiza L_{sum} sin comprometer a la anterior función objetivo. Todo esto, sin embargo, requiere alguna coordinación entre los APs para comunicar el valor de L_{max}. El procedimiento seguido en el caso de este algoritmo, se resume en el siguiente pseudocódigo:

1. **Inicialización**: En este paso se realizará la coloración inicial del grafo, que podrá ser cualquiera, desde una coloración uniforme de todos los vértices con el mismo color, hasta la utilización de LCCS para este fin.

2. **Optimización**: En primer lugar, se define w_{max} como el valor actual de L_{max} con la coloración actual (denotado $L_{max}(C)$).

   a) Para cada color, se calcula $H(c)$ que corresponde al peso máximo entre las aristas de ap_i en las que ap_j tiene el color c. Se define de la siguiente manera:

   $$H(c) = \max_{\forall e=(ap_i,ap_j)\in E \wedge C(ap_j)=c} I(ap_i, ap_j)W(ap_i, ap_j)$$

   b) Se comprueba si $H(c) \geq w_{max}$ y si es así $M(c) = 1$ y si no es así, $M(c) = 0$. Con este último parámetro M, evitamos poder elegir los colores, cuyo valor $M(c) = 1$.

   c) Definimos $S(c)$ como el sumatorio de los pesos de todas las aristas de ap_i en las que su vecino (ap_j) tiene el color c:

   $$L_{sum}(G, C) = \sum_{\forall e=(ap_i,ap_j)\in E \wedge C(ap_j)=c} I(ap_i, ap_j)W(ap_i, ap_j)$$

   d) Se elige c' de forma que $S(c') = \min_{1\leq c\leq k \wedge M(c)=0} S(c)$.

   e) Se selecciona c' como el color de ap_i.

5.4.5 Simulación y conclusiones

Este apartado trata de mostrar, en primer lugar, mediante una serie de experimentos con distintos tipos de grafo y parámetros el comportamiento del algoritmo LCCS con respecto al caso aleatorio. Después de este contraste, los algoritmos *Hminmax* y *Hsum* se compararán con los anteriores. Además, para finalizar, utilizando varias coloraciones iniciales para los algoritmos *Hminmax* y *Hsum* se observará como varía la utilidad en cada caso.

5.4.5.1 Simulación

En primer lugar, es necesario aclarar que la matriz de interferencia cocanal utilizada ha sido generada siguiendo el procedimiento expuesto en [29].

Antes de realizar un estudio de los algoritmos *Hminmax* y *Hsum*, es necesario contrastar el algoritmo LCCS con respecto al caso aleatorio para observar las carencias de este algoritmo ampliamente utilizado en multitud de redes en todo el mundo.

Los escenarios o grafos a utilizar en esta comparación, y en el resto de las simulaciones dentro de este apartado, son los siguientes:

- El radio de cobertura de los nodos d_{ap} (APs y clientes indistintamente) se establece en 0.25.
- Un número de clientes $n_{c/ap}$ correspondientes a cada punto de acceso de 1, 5 y 10. La correspondencia de éstos, no implica su conexión con dicho punto de acceso sino que simplemente se utiliza para calcular el número de clientes final, que será el número de APs multiplicado por este valor: $n_c = n_{ap}^2 \cdot n_{c/ap}$.
- Un número n_{ap} correspondiente a la raíz cuadrada del número de APs, por lo que el número de APs corresponderá al cuadrado de este valor. Los valores establecidos para este campo serán los siguientes: 4, 7 y 10
- Como generadores de posición se establecen 4: *random-random*, *random-normal*, *square-random* y *square-normal*. *tri-random* y *tri-normal* no se utilizarán ya que a pesar de ser configuraciones interesantes, sólo para casos con los valores anteriores muy altos presentan un número de aristas apreciable, siendo en los demás casos inapreciable e inútil para realizar experimentos.
- Para los casos *random-normal* y *square-normal* se establece un parámetro $divisordap$ que se utiliza para calcular la desviación típica de la siguiente forma: $\sigma = d_{ap}/divisordap$. Los valores utilizados en estos experimentos son 2 y 3.
- Se realizarán 15 ejecuciones de cada combinación de experimentos resultando en un total de 2160 grafos, ya que el generador de grafos crea 2 grafos para cada escenario.

Tras generar los grafos necesarios, se ha calculado la utilidad total del grafo para la coloración mediante LCCS y mediante el algoritmo aleatorio como la suma de las utilidades de todos los APs del grafo. En las tablas 5.12, 5.13, 5.14 y 5.15 se muestran los resultados obtenidos para cada generador de posición utilizado.

En general, los resultados son mejores en el caso aleatorio, lo que implica que el algoritmo LCCS, utilizado globalmente en numerosas redes, es muy similar al caso aleatorio atendiendo a la minimización de la interferencia. Sin embargo, LCCS tiene varias ventajas sobre el caso aleatorio. La primera es la fiabilidad: en el caso aleatorio, debido a su comportamiento estocástico, pueden darse resultados muy opuestos, al contrario que en LCCS, ya que en este último no influye ningún factor aleatorio. Además, LCCS realiza un uso más eficiente de los colores, sin utilizar todos ellos si no es necesario, al contrario que el caso aleatorio.

A continuación, es el turno de la comparación de los algoritmos *Hminmax*, *Hsum* y el caso aleatorio. No se realiza una comparación con el algoritmo LCCS ya que el algoritmo aleatorio utilizado presenta mejores resultados.

Como coloración inicial de los nodos en *Hminmax* y *Hsum* se han utilizado 3 variantes: coloración fija a un color (*fixed*), coloración mediante LCCS (*lccs*) y coloración aleatoria uniforme (*random*).

Tabla 5.12: Caso *random-random*

nap	nc	$lccs$	$random$
4	1	3.7534	3.1304
4	5	25.0528	22.6812
4	10	40.9573	32.6751
7	1	37.0677	35.9168
7	5	289.8665	272.1010
7	10	520.7370	484.7567
10	1	123.3846	148.5873
10	5	1146.8587	1152.5204
10	10	2326.8521	2375.0991

Tabla 5.13: Caso *square-random*

nap	nc	$lccs$	$random$
4	1	2.2318	1.6668
4	5	17.2409	7.3151
4	10	33.5706	20.3405
7	1	47.8928	49.0536
7	5	321.9336	302.5163
7	10	553.8868	507.1059
10	1	164.1394	211.0456
10	5	1300.3487	1355.6173
10	10	2367.7086	2420.1246

Tabla 5.14: Caso *random-normal*

nap	nc	dd	$lccs$	$random$
4	1	2	8.7060	8.9857
4	1	3	11.4194	10.5494
4	5	2	50.0150	49.2097
4	5	3	52.0212	45.0802
4	10	2	92.4291	91.0019
4	10	3	110.2823	98.9785
7	1	2	90.6924	94.6433
7	1	3	104.4812	96.3344
7	5	2	540.3763	542.4549
7	5	3	584.7098	544.4261
7	10	2	996.2615	1001.3992
7	10	3	1064.7723	1010.9257
10	1	2	339.4081	372.1670
10	1	3	364.1744	417.8891
10	5	2	2070.0458	2230.0647
10	5	3	2363.4624	2327.9116
10	10	2	4026.4793	4099.1710
10	10	3	4148.0608	4246.8139

Tabla 5.15: Caso *square-normal*

nap	nc	dd	$lccs$	$random$
4	1	2	7.2604	7.9262
4	1	3	12.5613	13.4904
4	5	2	42.6643	47.6951
4	5	3	43.7974	41.5787
4	10	2	68.8417	68.9937
4	10	3	75.7529	72.1428
7	1	2	84.1413	84.0959
7	1	3	100.0613	97.63657
7	5	2	524.8209	481.3443
7	5	3	558.1682	465.9707
7	10	2	933.4835	900.6306
7	10	3	938.4852	939.4004
10	1	2	307.9390	371.4571
10	1	3	333.6347	407.0453
10	5	2	2106.8667	2118.0563
10	5	3	2222.8695	2257.8759
10	10	2	3955.1393	3991.2458
10	10	3	4022.9920	4148.1334

A la hora de realizar la implementación de los algoritmos, en la información incluida en el artículo sobre ellos, falta una aclaración bastante importante acerca de la forma de proceder en la elección de color si resulta haber varios colores a la hora de elegir. Para clarificar, el siguiente ejemplo muestra un caso con esta ocurrencia:

1. Tenemos un AP 1 con color 1 (coloración inicial fija). Suponemos que es el primero en ser coloreado, ya que los algoritmos funcionan de una manera incremental.

2. Para cada AP vecino n, calculamos el peso como el productor de *I-factor* con W (W se calcula dividiendo el número de nodos cliente dentro de la zona de cobertura de ambos APs entre el total de la suma de los nodos clientes de ambos APs). Si ese valor obtenido es mayor que el mayor obtenido anteriormente para el color del vecino n, se establece como el nuevo valor para ese color.

3. Se continúa así hasta que no quedan más APs vecinos y entonces empieza el problema de la falta de aclaración.

4. Los valores calculados en el paso 2 se guardan en un diccionario. Para simplificar, suponemos que todos los nodos tienen el color 1 y suponemos entonces un valor de dicho diccionario `{1: 1.0, 2:0, ... , 11: 0}`. Sabemos que el que no hay que elegir es el 1, ya que tiene el mayor valor, sin embargo tenemos diez colores entre los que elegir ya que tienen el mismo valor (en este caso 0).

La estrategia seguida hasta el momento en la implementación consiste en elegir el primero, es decir, el color 2. Esto permite un mayor aprovechamiento de los colores, sin embargo, presenta un problema: el aumento de la interferencia.

Para intentar solucionar este problema, a falta de datos necesarios para este paso, se ha pensado una estrategia diferente: elegir aleatoriamente uno entre los posibles colores. Esta solución permitirá disminuir la interferencia, sin embargo, en un futuro habrá que utilizar otros métodos más elaborados que permitan obtener mejores resultados. En las tablas 5.16, 5.17, 5.18 y 5.19 se muestran los resultados obtenidos.

En primer lugar, comparando *Hminmax* y *Hsum*, podemos observar que, en el caso *random-random*, para una coloración inicial fija o mediante LCCS, la interferencia es menor en *Hsum* en la mayoría de los casos. Esto es normal ya que *Hsum* es un algoritmo orientado a disminuir L_{max} y L_{sum} (minimiza la interferencia a nivel de la totalidad del grafo) al contrario que *Hminmax* el cual sólo minimiza el primero de ellos. En casos para un número de nodos mayor, se nota más la diferencia (por ejemplo, el caso $nap = 10, nc = 10$).

Respecto al caso aleatorio, es notable observar que en el caso de grafos con pocos nodos ($nap = 4, nc = 1, 5, 10$) el caso aleatorio obtiene mejores resultados que *Hminmax* en 2 de 3 casos y que *Hsum* en un caso. Sin embargo, para casos con mayor número de nodos, el caso aleatorio se ve superado siempre por los dos algoritmos.

En el caso *random-normal* podemos observar que para casos con número de nodos bajo, los 3 algoritmos están bastante igualados. Ahora bien, con el aumento del número de nodos, *Hminmax* y *Hsum* obtienen mejores resultados que el caso aleatorio y entre ellos dos, *Hsum* obtiene los mejores.

Por último, en los casos *square-random* y *square-normal* los resultados son muy similares a los obtenidos anteriormente por lo que resultaría repetitiva la explicación.

Tabla 5.16: Caso *random-random*

nap	nc	hmm_{fix}	hmm_{lccs}	hmm_{ran}	hs_{fixed}	hs_{lccs}	hs_{ran}	ran
4	1	2.343	3.001	2.830	1.831	1.991	3.277	3.130
4	5	18.875	18.642	19.348	18.761	17.467	18.521	22.681
4	10	31.332	34.539	39.322	33.986	35.791	39.176	32.675
7	1	30.193	31.132	29.349	29.265	32.476	27.836	35.917
7	5	222.763	226.898	230.661	221.299	229.728	220.013	272.101
7	10	425.798	408.918	415.899	421.955	415.555	403.339	484.757
10	1	109.633	109.968	111.944	113.445	119.110	113.730	148.587
10	5	920.583	924.760	918.924	911.334	918.911	908.295	1152.520
10	10	1874.631	1906.379	1840.979	1815.268	1842.744	1830.594	2375.099

Tabla 5.17: Caso *random-normal*

nap	nc	dd	hmm_{fix}	hmm_{lccs}	hmm_{ran}	hs_{fixed}	hs_{lccs}	hs_{ran}	ran
4	1	2	10.075	10.305	8.052	9.133	8.063	10.867	8.986
4	1	3	11.291	11.805	11.991	10.238	9.143	11.388	10.549
4	5	2	45.785	46.419	46.274	54.922	43.289	44.515	49.210
4	5	3	44.118	40.253	43.810	52.070	43.285	45.790	45.080
4	10	2	82.846	86.888	80.860	79.330	78.897	89.953	91.002
4	10	3	74.702	81.827	97.274	85.398	94.102	87.604	98.979
7	1	2	72.388	72.102	77.082	81.332	76.500	73.669	94.643
7	1	3	81.069	83.081	82.230	80.244	80.928	83.241	96.334
7	5	2	422.984	436.288	430.132	421.934	427.580	427.644	542.455
7	5	3	451.727	453.289	450.542	461.946	458.343	464.651	544.426
7	10	2	763.678	763.254	774.886	762.309	782.608	745.711	1001.399
7	10	3	821.100	822.265	793.522	825.997	797.771	824.066	1010.926
10	1	2	256.878	264.216	261.536	269.828	255.197	254.806	372.167
10	1	3	300.990	286.324	300.101	292.742	304.793	289.686	417.889
10	5	2	1781.445	1755.168	1768.039	1736.071	1763.894	1757.210	2230.065
10	5	3	1900.304	1902.050	1885.651	1906.769	1880.215	1896.410	2327.912
10	10	2	3391.008	3439.184	3373.035	3358.737	3315.100	3302.627	4099.171
10	10	3	3616.865	3570.408	3610.047	3519.564	3513.635	3472.086	4246.814

Tabla 5.18: Caso *square-random*

nap	nc	hmm_{fix}	hmm_{lccs}	hmm_{ran}	hs_{fixed}	hs_{lccs}	hs_{ran}	ran
4	1	2.253	3.062	2.612	1.971	3.342	2.645	1.667
4	5	16.464	17.091	14.411	14.318	10.870	15.248	7.315
4	10	26.195	23.016	22.062	23.701	23.873	24.956	20.341
7	1	34.151	38.407	39.916	34.104	38.182	36.704	49.054
7	5	255.056	258.739	246.639	248.651	252.099	254.661	302.516
7	10	432.469	446.403	443.261	419.851	433.880	413.627	507.106
10	1	143.330	134.586	140.648	140.644	150.145	147.300	211.046
10	5	991.246	1014.888	1010.711	1014.017	998.390	1009.749	1355.617
10	10	1799.196	1776.478	1799.581	1777.135	1800.921	1815.884	2420.125

Tabla 5.19: Caso *square-normal*

nap	nc	dd	hmm_{fix}	hmm_{lccs}	hmm_{ran}	hs_{fixed}	hs_{lccs}	hs_{ran}	ran
4	1	2	6.125	6.432	7.860	7.324	6.829	6.890	7.926
4	1	3	9.627	10.894	10.415	9.970	10.845	10.051	13.490
4	5	2	40.642	37.144	38.940	34.268	35.821	40.458	47.695
4	5	3	36.151	38.076	34.960	32.541	43.052	35.818	41.579
4	10	2	54.854	63.641	52.935	58.098	54.624	57.450	68.994
4	10	3	60.583	66.477	64.887	55.835	70.068	61.430	72.143
7	1	2	68.718	70.786	71.543	71.093	68.483	75.095	84.096
7	1	3	82.498	82.776	76.657	83.040	87.570	82.187	97.637
7	5	2	405.353	401.971	405.833	399.595	390.676	414.555	481.344
7	5	3	428.503	433.287	417.748	424.202	415.149	431.351	465.971
7	10	2	687.100	717.998	729.839	708.401	710.303	701.962	900.631
7	10	3	708.428	745.133	728.790	730.276	763.380	725.011	939.400
10	1	2	257.384	261.916	259.587	252.129	267.111	263.999	371.457
10	1	3	280.038	276.606	275.490	285.091	279.082	268.685	407.045
10	5	2	1657.282	1670.282	1664.779	1654.857	1626.171	1675.416	2118.056
10	5	3	1740.032	1743.608	1736.781	1718.972	1725.342	1720.177	2257.876
10	10	2	2982.178	3056.708	3062.224	2978.010	2979.351	2943.967	3991.246
10	10	3	3158.424	3199.299	3172.607	3138.827	3113.486	3127.213	4148.133

5.4.5.2 Conclusiones

Tras los resultados obtenidos, se ha podido comprobar que LCCS no resulta ser un buen algoritmo a la hora de intentar minimizar la interferencia entre canales, ya que se ve superado por el caso aleatorio. Sin embargo, observando los resultados de los algoritmos *Hminmax* y *Hsum* se puede observar que, añadiendo características al algoritmo, como aristas ponderadas y comunicación entre nodos, se pueden mejorar bastante estos resultados hasta el punto en el que en la gran mayoría de los casos, los algoritmos obtienen mejores valores para la utilidad que en el caso aleatorio y por extensión, que en LCCS.

Apartado 6

Conclusiones generales y trabajo futuro

La asignación de frecuencias es un problema bastante complejo que afecta a las redes inalámbricas en la actualidad. En este TFG, se ha estudiado la utilización de la coloración de grafos en el intento de dar una solución a este problema.

Debido a la complejidad del problema real (asignación de frecuencias en una red inalámbrica real), existen varios grados de abstracción dentro de las posibles soluciones que además coinciden con sus grados de dificultad.

En primer lugar, las soluciones más abstractas, intentan solucionar el problema de coloración apropiada de grafos (*proper coloring problem*) para alcanzar una solución. Entre las implementaciones realizadas, el algoritmo *Accumulated Weight*, el algoritmo DSATUR y su modificación M-DSATUR son de este tipo.

Las tres implementaciones llegan a una solución en un tiempo razonable y, al evitar aristas monocromáticas, la interferencia cocanal se reduce, lo cual se aprecia sobre todo en M-DSATUR que hace uso de la matriz de interferencias para elegir el coloreado. En el caso de éste último, en la totalidad de los casos obtiene una mejor utilidad que DSATUR y en gran cantidad de ellos presenta una mejor utilidad que el caso aleatorio y un menor número de canales utilizados. Sin embargo, en el caso de AW, presenta peor utilidad que en el caso aleatorio ya que, al trabajar con un número fijo de canales, el algoritmo está ideado de tal forma que piense en esos canales sin interferencia entre ellos y utilice el menor número posible y de menor índice, es decir, si por ejemplo tiene que utilizar 4 colores, elegirá el 1, 2, 3 y 4 antes que cualquier otro.

Además de lo comentado en el párrafo anterior, tener como objetivo la evitación de aristas monocromáticas acarrea una serie de desventajas. La primera y más importante, es que si nos acercamos al caso real y utilizamos la interferencia cocanal para elegir el color de los nodos, en este caso, solo podrán elegirse canales que no interfieran, reduciendo la cantidad utilizable. Además, si el número de canales es limitado (característica del caso real) puede darse el caso de que el algoritmo no converja en algunos casos (caso AW) al no poder realizar una coloración apropiada.

Siguiendo con los tipos de soluciones, los algoritmos propuestos por Mishra presentan un planteamiento novedoso respecto a los anteriores al utilizar aristas ponderadas (cuyo peso utiliza una matriz de inter-

ferencias en su cálculo) para realizar el coloreado y además presentar el objetivo de minimizar una determinada función. Ambos algoritmos sin embargo, al no dejar claro el autor la elección de color en el caso de tener varias opciones, no devuelven unos resultados satisfactorios con respecto a la utilidad si se elige el primero de entre ellos, pero sí si se elige de forma aleatoria siguiendo una distribución uniforme.

A pesar de ello, se trata de un enfoque novedoso orientado al caso real que puede servir de base para futuras implementaciones de nuevos algoritmos orientados cada vez más al caso real, que es el realmente importante a la hora de intentar buscar una aplicación práctica. Además, los algoritmos anteriores presentan características bastante interesantes como la comunicación entre nodos en el caso de AW, que pueden también ser útiles en futuros desarrollos.

Como tareas futuras en el desarrollo del tema tratado en este TFG existen un conjunto bastante interesante de ellas que podría permitir avanzar en la solución a la asignación de frecuencias.

Inicialmente y orientada solo al trabajo realizado en este TFG, estaría el caso de la optimización y la mejora de los algoritmos implementados, pudiendo mejorarse desde distintos puntos. Internamente, podría utilizarse programación multihilo para mejorar el uso de la capacidad del procesamiento y así multiplicar la velocidad de ejecución de las simulaciones. Además, se podrían intentar implementar los algoritmos en C++, ya que este lenguaje de programación, permite realizar una mejor gestión de memoria, necesaria si no se dispone de una memoria suficiente, ya que Python, al ser de más alto nivel, es bastante impredecible en su uso de memoria. Otra forma de solucionar este aspecto podría ser realizar *scripts* en *bash* para que el sistema operativo se encargue de liberar la memoria al finalizar la ejecución del *script* de Python. Como último aspecto y simplemente para simplificar el uso de las implementaciones, se podría realizar una interfaz gráfica que permitiese un uso más visual y sencillo de la implementación, otorgando a usuarios con poco conocimiento de Python utilizarlas sin problemas.

Fuera de la mejora de las implementaciones realizadas, la creación de nuevas propuestas de algoritmo, utilizando características propias de los implementados en este trabajo (comunicación entre nodos, aristas ponderadas, matriz de interferencias, etc) y añadiendo técnicas de negociación automática podría ser un buen punto de partida a la hora de innovar en este campo y acercarse a mejores soluciones para la asignación de frecuencia, que, como se comentaba al principio, es un problema serio que afecta a infinidad de redes inalámbricas en la actualidad.

Apéndice A

Manual de usuario

En este apartado se tratarán todos los aspectos necesarios para permitir el mantenimiento y desarrollo de mejoras en un futuro por personas ajenas a este proyecto.

En primer lugar se explicarán brevemente todos los módulos de cada implementación para finalmente proceder a explicar la organización de los archivos del CD y sus contenidos. Para un mayor detalle acerca de las implementaciones, se incluirá un manual de la API en versión electrónica en el CD, generado mediante la herramienta **Epydoc**.

A.1 Proyecto mishra: Generador de grafos euclídeos, LCCS, Hminmax y Hsum

Las 4 implementaciones nombradas en el título forman parte del mismo proyecto llamado **mishra** (`/code/mishra`). Esto ha sido decidido así ya que, en el artículo escrito por A. Mishra et al. se compara LCCS con Hminmax y Hsum. El generador de grafos también está en este proyecto ya que los dos últimos algoritmos hacen uso de él, y por extensión, para poder compararse LCCS con ellos, este también hace uso.

A.1.1 Generador de grafos euclídeos

Esta implementación, dentro del proyecto **mishra** (`/code/mishra`), se encuentra repartida entre los módulos `pos_generator`, `graph_generator`, `auto_graphs` y `auto_read_graphs`.

A.1.1.1 Módulo `pos_generator`

El módulo `pos_generator` (`/code/mishra/pos_generator.py`) contiene, implementados, todos los generadores de posición de los nodos: *random-random*, *random-normal*, *random-kuzmin*, *square-random*, *square-normal*, *square-kuzmin*, *tri-random*, *tri-normal* y *tri-kuzmin*. Además incluye un generador muy simple `random_position` que genera una posición aleatoriamente uniforme para todos los nodos.

A.1.1.2 Módulo `graph_generator`

En este módulo (`/code/mishra/graph_generator.py`) se encuentran todas las funciones necesarias para crear grafos de interferencia y de AP más cercano.

A.1.1.3 Módulo `auto_graphs`

Módulo (`/code/mishra/auto_graphs.py`) que contiene la automatización, en forma de bucles, utilizada para generar todos los grafos necesarios para realizar las simulaciones.

A.1.1.4 Módulo `auto_read_graphs`

Módulo (`/code/mishra/auto_read_graphs.py`) que contiene la automatización, en forma de bucles, utilizada para leer todos los grafos generados con anterioridad mediante el módulo `auto_graphs`.

A.1.2 Algoritmo LCCS

Dentro del proyecto **mishra** (`/code/mishra`), forman parte de esta implementación los módulos `lccs` y `auto_lccs`.

A.1.2.1 Módulo `lccs`

Módulo (`/code/mishra/lccs.py`) que contiene las funciones necesarias para utilizar la implementación del algoritmo LCCS.

A.1.2.2 Módulo `auto_lccs`

Módulo (`/code/mishra/auto_lccs.py`) que contiene la automatización, en forma de bucles, utilizada para ejecutar el algoritmo LCCS en todos los grafos generados con anterioridad con el módulo `auto_graphs`.

A.1.3 Algoritmo HMINMAX

Dentro del proyecto **mishra** (`/code/mishra`), forman parte de esta implementación los módulos `hminmax` y `auto_hminmax`.

A.1.3.1 Módulo `hminmax`

Módulo (`/code/mishra/hminmax.py`) que contiene las funciones necesarias para utilizar la implementación del algoritmo Hminmax.

A.1.3.2 Módulo `auto_hminmax`

Módulo (`/code/mishra/auto_hminmax.py`) que contiene la automatización, en forma de bucles, utilizada para ejecutar el algoritmo Hminmax en todos los grafos generados con anterioridad con el módulo `auto_graphs`.

A.1.4 Algoritmo HSUM

Dentro del proyecto **mishra** (`/code/mishra`), forman parte de esta implementación los módulos `hsum` y `auto_hsum`.

A.1.4.1 Módulo `hsum`

Módulo (`/code/mishra/hsum.py`) que contiene las funciones necesarias para utilizar la implementación del algoritmo Hsum.

A.1.4.2 Módulo `auto_hminmax`

Módulo (`/code/mishra/auto_hsum.py`) que contiene la automatización, en forma de bucles, utilizada para ejecutar el algoritmo Hsum en todos los grafos generados con anterioridad con el módulo `auto_graphs`.

A.1.5 Herramientas

Dentro del proyecto **mishra** (`/code/mishra`), existen también varios módulos que contienen funciones usadas por los distintos algoritmos, además de aquellas para generar las coloraciones aleatorias utilizadas en la comparación con estos algoritmos.

A.1.5.1 Módulo `automatization`

Módulo (`/code/mishra/automatization.py`) que contiene un conjunto de variables utilizadas por todas las automatizaciones en su mayoría,además de una función utilizada para generar una cadena de texto, necesaria también en las automatizaciones.

A.1.5.2 Módulo `graph_tools`

Módulo (`/code/mishra/graph_tools.py`) que contiene 3 funciones necesarias para los algoritmos Hminmax y Hsum además de para el generador de grafos. La primera sirve para calcular la distancia entre dos nodos, la segunda para obtener el peso W de las aristas entre APs de un grafo y la tercera para realizar una representación gráfica y geométrica de un grafo euclídeo.

A.1.5.3 Módulo `ifactor`

Módulo (`/code/mishra/ifactor.py`) que contiene la matriz de interferencias cocanal utilizada en los algoritmos Hminmax y Hsum y una función de cálculo de utilidad para un grafo.

A.1.5.4 Módulo `initial_coloring`

Módulo (`/code/mishra/initial_coloring.py`) que contiene las funciones que permiten generar la coloración inicial (fija o aleatoria) para los algoritmos Hminmax y Hsum, además de una función que permite colorear los clientes con el color de sus APs.

A.1.5.5 Módulo `read_write`

Módulo (`/code/mishra/read_write.py`) que contiene las funciones que permiten leer y escribir grafos y listas de parámetros en ficheros .gml y .pkl (pickle) respectivamente.

A.1.5.6 Módulo `random_coloring`

Módulo (`/code/mishra/random_coloring.py`) que contiene la función necesaria para realizar coloraciones aleatorias de los distintos grafos.

A.1.5.7 Módulo `auto_random`

Módulo (`/code/mishra/auto_random.py`) que contiene la automatización, en forma de bucles, utilizada para ejecutar el algoritmo aleatorio implementado en todos los grafos generados con anterioridad con el módulo `auto_graphs.py`.

A.2 Proyecto dsatur: `DSATUR` y M-`DSATUR`

En el proyecto **dsatur** (`/code/dsatur`), están reunidos todos aquellos módulos utilizados a la hora de generar coloración mediante DSATUR o M-DSATUR.

A.2.1 Módulo `auto_gen_graph`

Módulo (`/code/dsatur/auto_gen_graph.py`) a cargo de la automatización de los algoritmos DSATUR, M-DSATUR y aleatorio.

A.2.2 Módulo `coef_cochannel`

Módulo (`/code/dsatur/coef_cochannel.py`) que contiene la matriz de interferencia cocanal utilizada en los algoritmos DSATUR, M-DSATUR y aleatorio para el cómputo de la utilidad, cuya función principal también pertenece a este módulo.

A.2.3 Módulo `dsatur`

Módulo (`/code/dsatur/dsatur.py`) que contiene las funciones necesarias para ejecutar los algoritmos DSATUR y M-DSATUR.

A.2.4 Módulo `gen_generic_graph`

Módulo (`/code/dsatur/gen_generic_graph.py`) que contiene las funciones necesarias para crear grafos de los distintos tipos utilizados en los experimentos.

A.2.5 Módulo `graphical_util`

Módulo (`/code/dsatur/graphical_util.py`) que contiene las funciones usadas en la representación gráfica con colores de los grafos.

A.2.6 Módulo `other_coloring_alg`

Módulo (`/code/dsatur/other_coloring_alg.py`) que contiene la implementación de una coloración aleatoria creada para compararse con los algoritmos DSATUR y M-DSATUR.

A.2.7 Módulo `read_write`

Módulo (`/code/dsatur/read_write.py`) que contiene las funciones que permiten leer y escribir grafos y listas de parámetros en ficheros .gml y .pkl (pickle) respectivamente.

A.3 Proyecto aw y auto_aw: *Accumulated Weight*

En el proyecto **aw** (`/code/aw`), están reunidos todos aquellos módulos utilizados a la hora de generar coloraciones personalizadas por el usuario mediante AW. El proyecto **auto_aw** (`/code/auto_aw`), posee unos módulos similares al anterior, pero enfocados a generar gran número de experimentos de forma automática.

A.3.1 Módulo `node`

Módulo (`/code/aw/node.py` y `/code/auto_aw/node.py`) en el que la clase *node* está definida, la cuál provee a cada nodo de toda la lógica necesaria para poder ejecutar el algoritmo.

A.3.2 Módulo `main`

Módulo (`/code/aw/main.py` y `/code/auto_aw/main.py`) que contiene la función principal de la implementación del algoritmo AW.

A.3.3 Módulo `execution`

Módulo (`/code/aw/execution.py`) que se encarga de ejecutar la función principal de la implementación del algoritmo AW.

A.3.4 Módulo `global_var`

Módulo (`/code/aw/global_var.py` y `/code/auto_aw/global_var.py`) que se encarga de ejecutar la función principal de la implementación del algoritmo AW.

A.3.5 Módulo `aux_func_main`

Módulo (`/code/aw/aux_func_main.py` y `/code/auto_aw/aux_func_main.py`) que contiene funciones auxiliares, utilizadas por otros módulos, dentro de la implementación del algoritmo AW (Accumulated Weight).

A.3.6 Módulo `read_write`

Módulo (`/code/auto_aw/read_write.py`) que contiene las funciones que permiten leer y escribir grafos y listas de parámetros en ficheros .gml y .pkl (pickle) respectivamente.

A.3.7 Módulo `coef_cochannel`

Módulo (`/code/auto_aw/coef_cochannel.py`) que contiene la matriz de interferencia cocanal utilizada en el algoritmo AW para el cómputo de la utilidad, cuya función principal también pertenece a este módulo.

A.4 Organización de los archivos en el CD

Como se comentaba en la sección anterior en el CD adjunto a este TFG, existe una organización dentro de la carpeta `code`. Las rutas a los distintos módulos de las implementaciones han sido aclaradas en el apartado anterior por lo que en este apartado se tratará el formato de los archivos producto de las simulaciones.

A.4.1 Proyecto mishra

En este proyecto, existen dos carpetas `/code/mishra/graphs` y `/code/mishra/parameters` en las que se guardan los grafos en formato .gml y la lista de parámetros utilizada en formato pickle respectivamente.

En el primer caso, para guardar un grafo en formato .gml, se sigue la siguiente configuración para el nombre del fichero:

$$tipo_{pos} + " - " + p_1 + [" - " + p_n] + "\_" + tipo_{grafo}$$

- $tipo_{pos}$: representa el tipo de algoritmo utilizado para generar las posiciones de los nodos del grafo. e.g. *random-random*, *square-normal*.
- p_n: representa el parámetro n característico del grafo. Dependiendo del tipo de posición utilizado, el número y tipo de parámetros característicos será diferente. Así, si se trata de un grafo *random-random* tendrá 3 parámetros característicos: raíz cuadrada del número de APs, número de nodos por AP y número de ejecución, en este orden. Al contrario, si se trata por ejemplo de un grafo *square-random*: divisor del radio de cobertura de los APs (utilizado en el cálculo de la desviación estándar), raíz cuadrada del número de APs, número de nodos por AP y número de ejecución, en este orden.

- $tipo_{grafo}$: los grafos podrán ser de dos tipos: "interferencia" (*udg*) y "AP más cercano" (*nng*).

En el caso de la lista de parámetros, sigue un formato muy similar al anterior que es el siguiente:

$$tipo_{pos} + "-" + p_1 + ["-" + p_n] + "-" + tipo_{algoritmo} + "\_" + [col_{inicial}]$$

- $tipo_{pos}$: representa el tipo de algoritmo utilizado para generar las posiciones de los nodos del grafo. e.g. *random-random*, *square-normal*.

- p_n: representa el parámetro n característico del grafo. Dependiendo del tipo de posición utilizado, el número y tipo de parámetros característicos será diferente. Así, si se trata de un grafo *random-random* tendrá 3 parámetros característicos: raíz cuadrada del número de APs, número de nodos por AP y número de ejecución, en este orden. Al contrario, si se trata por ejemplo de un grafo *square-random*: divisor del radio de cobertura de los APs (utilizado en el cálculo de la desviación estándar), raíz cuadrada del número de APs, número de nodos por AP y número de ejecución, en este orden.

- $tipo_{algoritmo}$: los algoritmos pertenecientes a este proyecto que podrán presentarse en este campo son los siguientes: Hminmax (*hminmax*), Hsum (*hsum*), LCCS (*lccs*) y aleatorio (*rand*).

- $col_{inicial}$: este campo sólo es aplicable a los casos en los que el algoritmo utilizdo sea Hminmax o Hsum y podrá tener 3 valores: coloración fija a un color (*fixd*), coloración mediante LCCS (*lccs*) y coloración aleatoria (*rand*).

A.4.2 Proyecto dsatur

Este proyecto es similar al anterior. Existen dos carpetas `/code/dsatur/graphs` y `/code/dsatur/params` en las que se guardan los grafos en formato .gml y la lista de parámetros utilizada en formato pickle respectivamente.

En el primer caso, para guardar un grafo en formato .gml, se sigue la siguiente configuración para el nombre del fichero:

$$tipo_{grafo} + "\_" + n_{nodos} + "\_" + param_{grafo} + "\_" + n_{ejec}$$

- $tipo_{grafo}$: representa el tipo de grafo. En este caso sólo hemos utilizado dos tipos de grafo y el valor de este campo en esos casos es el siguiente: *ba* para el modelo Barabási-Albert y *er* en el caso Erdős–Rényi.

- n_{nodos}: representa el número de nodos en el grafo.

- $param_{grafo}$: representa el valor del parámetro característico del grafo utilizado (parámetro p en el caso Erdős–Rényi y parámetro m en Barabási-Albert).

- n_{ejec}: representa el número de ejecución.

En el caso de la lista de parámetros, sigue un formato muy similar al anterior que es el siguiente:

$$tipo_{grafo} + "\_" + n_{nodos} + "\_" + param_{grafo} + "\_" + n_{ejec} + "\_" + tipo_{algoritmo}$$

- $tipo_{grafo}$: representa el tipo de grafo. En este caso sólo hemos utilizado dos tipos de grafo y el valor de este campo en esos casos es el siguiente: *ba* para el modelo Barabási-Albert y *er* en el caso Erdős–Rényi.
- n_{nodos}: representa el número de nodos en el grafo.
- $param_{grafo}$: representa el valor del parámetro característico del grafo utilizado (parámetro p en el caso Erdős–Rényi y parámetro m en Barabási-Albert).
- n_{ejec}: representa el número de ejecución.
- $tipo_{algoritmo}$: los algoritmos pertenecientes a este proyecto que podrán presentarse en este campo son los siguientes: DSATUR (*dsatur*), M-DSATUR (*mdsatur*) y coloración aleatoria (*random*).

A.4.3 Proyectos auto_aw

Este proyecto es similar al anterior. Existen dos carpetas `/code/auto_aw/node_information` y `/code/auto_aw/param_information` en las que se guardan los grafos en formato .gml y la lista de parámetros utilizada en formato pickle respectivamente.

En el primer caso, para guardar un grafo en formato .gml, se sigue la siguiente configuración para el nombre del fichero:

$$tipo_{grafo} + "\_" + n_{nodos} + "\_" + n_{colores} + "\_" + param_{grafo} + "\_" + n_{ejec}$$

- $tipo_{grafo}$: representa el tipo de grafo. En este caso sólo hemos utilizado dos tipos de grafo y el valor de este campo en esos casos es el siguiente: *BA* para el modelo Barabási-Albert, *ER* en el caso Erdős–Rényi y *R* para el caso de grafos regulares aleatorios.
- n_{nodos}: representa el número de nodos en el grafo.
- $n_{colores}$: representa el número de colores máximo permitido para realizar la coloración.
- n_{ejec}: representa el número de ejecución.

En el caso de la lista de parámetros, sigue un formato idéntico al caso anterior.

Apéndice B

Pliego de condiciones

En este anexo se procederá a explicar todas las condiciones, relacionadas con el *software* y *hardware* utilizados en el desarrollo de este trabajo, para que pueda ser reproducido, mantenido u objeto de mejoras en un futuro por una persona ajena al proyecto, si fuese necesario.

B.1 Hardware: Especificaciones técnicas

En este TFG, se han utilizado 2 PCs de sobremesa y un PC portátil. A continuación, se resumen las características:

- **PC sobremesa clónico (laboratorio)**: PC utilizado en el laboratorio con las siguientes especificaciones:
  - **Procesador**: Intel® Core™ 2 Quad Q9300 @2.50 GHz
  - **Memoria RAM**: 4 GB
  - **Disco Duro**: 500 GB
  - **Sistema Operativo**: Windows 7 Enterprise (64 bits)
- **PC sobremesa personal HP Pavilion**: PC personal con las siguientes especificaciones:
  - **Procesador**: Intel® Core™ 2 Quad Q9300 @2.50 GHz
  - **Memoria RAM**: 5 GB
  - **Disco Duro**: 1000 GB
  - **Sistema Operativo**: Windows 7 Professional (64 bits)
- **PC portátil personal Acer Aspire 5542**: PC portátil personal con las siguientes especificaciones:
  - **Procesador**: AMD Athlon II M300 @ 2 GHz (Dual-core)
  - **Memoria RAM**: 6 GB
  - **Disco Duro**: 320 GB
  - **Sistema Operativo**: Windows 7 Home Premium (64 bits)

Ambos PCs de sobremesa han sido utilizados para realizar todos los experimentos en este TFG además de la memoria de éste. El PC portátil ha sido utilizado a la hora de programar varias implementaciones y realizar pequeñas pruebas, además de para realizar la memoria.

B.1.1 Especificaciones mínimas

- **Procesador**: todas las implementaciones han sido realizadas sin utilizar programación **multihilo**, por lo que un procesador de 1 sólo núcleo será suficiente para realizar los experimentos expuestos en este TFG.

- **Memoria RAM**: hoy en día, es común encontrar una memoria RAM base de 4 GB en la mayoría de los PCs del mercado, por lo que se establecerá como memoria mínima. Sin embargo, en la ejecución de los algoritmos se han llegado a medir hasta 600 MB utilizados como máximo, por lo que un PC con 2 GB podría establecerse como el mínimo (1 GB sería demasiado arriesgado, ya que parte de la memoria RAM es consumida por el sistema operativo).

- **Disco Duro**: como se comentó con respecto a la memoria RAM, hoy en día, es común encontrarse con PCs cuyos discos duros rondan los cientos de GB, incluso los TB (1TB = 1000 GB). Esto, sumado a que las implementaciones junto a las ejecuciones ocupan una cantidad ínfima, del orden de los MB, no hay una especificación mínima en este caso.

- **Sistema Operativo**: aunque el sistema operativo sea más parte del *software* que del *hardware*, se ha decidido comentar en este apartado ya que de él depende todo el apartado siguiente acerca del *software* utilizado. Los sistemas operativos utilizados han sido todos ediciones más o menos potentes de Microsoft Windows 7 de 64 bits. Se recomienda el uso de este, sin embargo, la mayoría del *software* utilizado (salvo Microsoft Office) está disponible para sistemas operativos basados en UNIX como Linux, OS X, etc.

B.2 Software: Programas utilizados

Dentro del *software* utilizado, además del sistema operativo, comentado en la sección anterior, han sido necesarios varios programas que se especifican a continuación:

- **Anaconda**: *Anaconda Scientific Python Distribution* es una distribución de Python creada por la empresa *Continuum Analytics* formada por numerosos *packages* ampliamente conocidos y usados en el mundo de la computación científica. Es gratuito para uso académico y se puede descargar en su página oficial `https://store.continuum.io/cshop/anaconda/`.

  Entre los *packages* pertenecientes a esta distribución, en este TFG han sido utilizados los siguientes:

  - `NetworkX`. Contiene todas las funciones necesarias para la manipulación de grafos de toda clase de naturaleza. En este TFG se ha utilizado la versión 1.9.1 aunque el 2 de agosto de 2015 se lanzó la versión 1.10.

  - `matplotlib`. Necesario para la representación gráfica de los distintos grafos y diagramas. Se ha utilizado la última versión lanzada (1.4.3).
  - `numpy`. Utilizado simplemente para la manipulación de arrays. Se ha utilizado la versión 1.9.2 utilizada.
  - `scipy`. De este *package* ha sido utilizado simplemente aquellas funciones necesarias para generar triangulaciones de Delaunay de conjuntos de nodos en casos concretos. Se ha trabajado con la versión 0.15.1.
  - `epydoc`. Herramienta necesaria para generar el *API manual* o manual de interfaz de programación, incluido en el CD en formato electrónico. Se ha utilizado la versión 3.0.1.

  Para finalizar las especificaciones acerca de Anaconda, cabe informar que se ha utilizado la versión de Anaconda para Python 2.7.9 y el intérprete utilizado ha sido el propio de esta distribución.

- **JetBrains PyCharm Community Edition**: Este IDE (Entorno de Desarrollo Integrado, *Integrated Development Environment*) ha sido utilizado para generar y probar todo el código utilizado en las implementaciones. Se trata de un IDE muy potente y gratuito en su edición de la comunidad y puede descargarse directamente desde la página de *JetBrains* `https://www.jetbrains.com/pycharm/`. Para la realización de este TFG se ha utilizado la versión 4.5.3.

- **Microsoft Office 2013**: Ha sido utilizado Microsoft Excel simplemente para facilitar el manejo de grandes cantidades de datos. Es un programa de pago y puede ser obtenido en su versión para estudiantes en la `http://www.microsoftstore.com/store/mseea/es_ES/pdp/Office-Hogar-y-Estudiantes-2013/productID.263157600`.

- **TeXstudio**: Entorno integrado de escritura para crear documentos LaTeX. Es gratuito y se puede obtener en su página oficial `http://www.texstudio.org/`. Ha sido utilizada la versión 2.10.

- **Google Chrome**: Navegador web gratuito utilizado. Puede ser descargado en la página oficial de Google `https://www.google.es/chrome/browser/desktop/index.html`.

- **Sublime Text 2**: Editor de texto sofisticado y potente cuya licencia no es gratuita, pero permite una evaluación durante tiempo ilimitado. Puede descargarse en su página oficial `http://www.sublimetext.com/2`.

- **Sumatra PDF Reader**: Lector de archivos PDF sencillo, ligero y gratuito utilizado para la lectura de los diferentes artículos incluidos en la bibliografía. Puede descargarse desde su página oficial `http://www.sumatrapdfreader.org/free-pdf-reader-es.html`.

Apéndice C

Presupuesto

En este anexo se va a describir el proyecto de una manera económica explicando los gastos generales que comprenden la mano de obra y los costes materiales.

C.1 Gastos de mano de obra

Según la fuente [32], el sueldo medio de un ingeniero de telecomunicaciones es de 30000 € al año, que en €/h corresponde al resultado del siguiente cálculo:

$$euros_{hora} = \frac{30000}{217 \cdot 8} = 17.28€/h$$

El número 217 corresponde al número de días laborales, resultado de restar a 365, 148 días no laborales: 52 sábados, 52 domingos, 16 festivos, 3 días libres y 25 días de vacaciones (estas dos últimas, son una media).

Han sido necesarias 550 horas para la realización de este trabajo, por lo que el gasto de mano de obra ha sido: $mano_{obra} = 17.28€/h \cdot 550h = 9504.60€$

C.2 Costes materiales

A continuación se muestra una tabla (C.1) con el coste de los recursos empleados, tanto *software* como *hardware*:

C.3 Presupuesto final

Finalmente, tras calcular todos los gastos y costes, el presupuesto de este trabajo se estima en 9883.60 €.

Recurso	Precio (€)	Duración (años)	Tiempo de uso (meses)	Coste (€)
PC de sobremesa con licencia Windows 7 Professional	600	5	10	100
Licencia Windows 7 Professional	160	-	10	160
Anaconda Scientific Python Distribution	0	-	-	0
JetBrains PyCharm Community Edition	0	-	-	0
Office Hogar y Estudiantes 2013	119	-	-	119
Epydoc	0	-	-	0
TeXstudio	0	-	-	0
Navegador Google Chrome	0	-	-	0
Sublime Text 2	0	-	-	0
Sumatra PDF reader	0	-	-	0
				379

Tabla C.1: Costes materiales

Referencias

[1] E.Z. Tragos et al. "Spectrum Assignment in Cognitive Radio Networks: A Comprehensive Survey", *IEEE Commun. Surveys and Tutorials*, vol. 15, no. 3, pp. 1100-1135, 2013.

[2] J.A. Bondy and U.S.R. Murty, "Graphs and subgraphs," in *Graph Theory with Applications*, 5th ed., New York, Elsevier Science Publishing Co. Inc., 1982, ch. 1, sec. 1, pp 001-005

[3] Keijo Ruohonen, "Definitions and Fundamental Concepts," in *Graph Theory*, 2014, ch. 1, sec. 1, pp 001-002. [Online]. Available: `http://math.tut.fi/~ruohonen/GT_English.pdf`

[4] `https://en.wikipedia.org/wiki/Geometric_graph_theory`

[5] `http://math.stackexchange.com/questions/1056744/what-is-a-euclidean-graph\-can-edges-be-negative-in-a-euclidean-graph`

[6] `https://networkx.github.io/documentation/latest/reference/generated/networkx.generators.geometric.random_geometric_graph.html`

[7] A.L. Barabási and R. Albert, “Emergence of scaling in random networks”, *Science*, vol. 286, no. 5439, pp. 509-512, 1999.

[8] P. Erdős and A. Rényi, "On random graphs I", *Publ. Math. Debrecen*, vol. 6, pp. 290-297, 1959.

[9] `https://en.wikipedia.org/wiki/Erd%C5%91s%E2%80%93R%C3%A9nyi_model`

[10] `https://networkx.github.io/documentation/latest/reference/generators.html`

[11] `https://www.courses.psu.edu/for/for466w_mem14/Ch11/HTML/Sec1/ch11sec1_ObjFn.htm`

[12] G.E. Blelloch et al. "Design and implementation of a practical parallel Delaunay algorithm", *Algorithmica*, vol. 24, no. 3-4, pp. 243-269, 1999.

[13] B. Horowitz and R. Vuduc, "Well-Separated Pair Decomposition", 1997.

[14] `https://en.wikipedia.org/wiki/Normal_distribution`

[15] C. McDiarmid and B. Reed. "Channel assignment and weighted coloring" *Networks*, vol. 36, no. 2, pp. 114-117, 2000.

[16] `https://en.wikipedia.org/wiki/Nearest_neighbor_graph`

[17] `https://en.wikipedia.org/wiki/Polygon_triangulation`

[18] `https://en.wikipedia.org/wiki/Delaunay_triangulation`

[19] J.L. Bentley et al. "The complexity of finding fixed-radius near neighbors", *Information processing letters*, vol. 6, no 6, pp. 209-212, 1977.

[20] https://es.wikipedia.org/wiki/Carrier_sense_multiple_access_with_collision_avoidance

[21] M. Achanta, “Method and Apparatus for Least Congested Channel Scan for Wireless Access Points,” US Patent No. 20060072602, Apr. 2006

[22] S. Chieochan et al. "Channel assignment schemes for infrastructure-based 802.11 WLANs: A survey." *IEEE Commun. Surveys and Tutorials*, vol. 12, no. 1, pp. 124-136, 2010.

[23] R. Albert and A. Barabási. "Statistical mechanics of complex networks". *Reviews of modern physics*, vol. 74, no. 1, pp. 047-097, 2002.

[24] A. Mishra, et al. "Weighted coloring based channel assignment for WLANs." *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 9, no. 3, pp. 019-031, 2005.

[25] D. Brélaz, "New methods to color the vertices of a graph", *Communications of the ACM* vol. 22, no.4, pp. 251-256, 1979.

[26] http://networkx.github.io/documentation/development/_modules/networkx/algorithms/coloring/greedy_coloring.html

[27] R. Anthony. "Scalable and efficient graph colouring in 3 dimensions using emergence engineering principles." *in Proc. 2nd IEEE Int. Conf. Self-Adapt. Self-Organizing Syst.*, pp.370-379 2008.

[28] https://en.wikipedia.org/wiki/Wi-Fi

[29] S.W.K. Ng and T.H. Szymanski, "Interference measurements in an 802.11n wireless mesh network testbed", *Proc. 25th IEEE Canadian Conf. Elect Comput Eng.*, Montreal, Canada, 2012, pp 001-006.

[30] https://networkx.github.io/documentation/latest/reference/generated/networkx.generators.random_graphs.random_regular_graph.html#random-regular-graph

[31] https://docs.python.org/2/library/random.html#module-random

[32] http://empleo.trovit.es/3309/salarios-ingeniero-telecomunicaciones