PRÁCTICA 02

ESQUEMA GREEDY O VORAZ

Instalación de fibra óptica

Autores
Antonio José Jiménez Luque
Adrián Jiménez Benítez

Asignatura
Estructura de Datos y Algoritmos II

Titulación
Grado en Ingeniería Informática

Índice.

Índice de ilustraciones.

Índice de tablas.

1. Antecedentes.

El desarrollo de redes de telecomunicaciones eficientes ha sido una preocupación constante en la era de la información, en la que la conectividad se ha convertido en una necesidad primordial. En este contexto, los algoritmos Greedy o Voraces se presentan como una solución eficaz para la optimización de recursos en la implementación de infraestructuras de comunicación. Esta práctica se centra en la aplicación de dichos algoritmos al problema específico de la actualización de una red de telecomunicaciones mediante la instalación de fibra óptica en EDAland, un escenario ficticio que simula las complejidades y desafíos reales enfrentados por las empresas en el sector.

La motivación detrás de este estudio radica en la necesidad de maximizar la eficiencia en la distribución de recursos limitados, en este caso, el presupuesto para la actualización de la red de telecomunicaciones. La actual crisis económica exige soluciones que no solo sean técnicamente viables sino también económicamente sostenibles. El esquema Greedy se presenta como una estrategia prometedora al permitir la selección de las opciones más coste-efectivas en cada paso del proceso de decisión, con el objetivo de encontrar una solución global óptima para el problema planteado.

En esta práctica, se abordará el desafío de determinar los trazados de fibra óptica más eficientes para conectar grandes núcleos de población en EDAland, garantizando la optimización del presupuesto y la cobertura. Se analizarán los algoritmos de Prim y Kruskal, ambos ejemplos de esquemas Greedy, aplicados a la construcción de árboles de recubrimiento mínimo, y se explorará también el problema del viajante de comercio (TSP) para optimizar las rutas de mantenimiento de la red. La elección de estos algoritmos y problemas refleja un enfoque práctico y teórico para abordar los retos inherentes a la planificación y ejecución de infraestructuras de telecomunicaciones en un entorno de recursos limitados.

Este estudio pretende no solo aplicar conceptos teóricos aprendidos en el aula sino también contribuir al desarrollo de soluciones prácticas en el campo de las telecomunicaciones, destacando la relevancia de los algoritmos Greedy en la resolución de problemas de optimización en el mundo real. A través del análisis teórico y experimental de los algoritmos implementados, esta práctica ofrece una oportunidad para evaluar la eficacia de estas estrategias algorítmicas en contextos específicos, proporcionando así insights valiosos para futuras aplicaciones y desarrollos en el sector.

2. Objeto.

El principal propósito de esta práctica es aplicar y profundizar en el conocimiento de los esquemas algorítmicos Greedy, con un enfoque particular en su utilidad para resolver problemas complejos de optimización que surgen en la implementación de redes de telecomunicaciones. Los objetivos específicos se pueden detallar de la siguiente manera:

Implementación de Soluciones Greedy: Desarrollar soluciones prácticas a un problema específico de actualización de la red de telecomunicaciones utilizando el método algorítmico Greedy. Este objetivo incluye la implementación de algoritmos conocidos como el de Prim y Kruskal, adaptados a las necesidades del problema propuesto, para determinar la forma más eficiente de conectar núcleos de población mediante fibra óptica.

Análisis Cualitativo y Cuantitativo: Comparar las soluciones obtenidas mediante los diferentes algoritmos Greedy implementados, tanto desde un punto de vista cualitativo como cuantitativo. Esto implica evaluar la efectividad, eficiencia, y el costo de las soluciones propuestas en diferentes escenarios, utilzando para ello tanto redes de telecomunicaciones reales como sintéticas.

Evaluación de la Eficiencia: Realizar un análisis detallado sobre la eficiencia de los algoritmos implementados. Este análisis deberá abarcar tanto aspectos teóricos, basados en la complejidad algorítmica, como prácticos, mediante la ejecución de pruebas sobre redes de diferentes tamaños y complejidades.

Comparativa Teórica y Práctica: Establecer una comparativa entre los resultados teóricos y los obtenidos experimentalmente. Este objetivo busca validar las hipótesis teóricas con respecto a la eficiencia y efectividad de los algoritmos Greedy en el contexto de las redes de telecomunicaciones, ajustando las expectativas con respecto a los resultados reales.

Aplicación a Problemas Concretos: Aplicar los algoritmos Greedy para resolver problemas específicos identificados en el escenario de EDAland, incluyendo la optimización de la red de telecomunicaciones existente y la planificación de rutas de mantenimiento eficientes. Esto implica una adaptación y aplicación práctica de los conceptos teóricos a situaciones reales y concretas.

Para este proyecto seguidamente detallaremos el líder y las tareas que ha realizado cada uno de los componentes del equipo.

El líder para esta práctica es Adrián Jiménez Benítez.

En la siguiente tabla se recogerá cada una de las tareas a implementar por los distintos miembros del equipo.

Tareas	***Miembro del equipo***
Ejercicio 1	Antonio José Jiménez Luque
Ejercicio 2	Adrián Jiménez Benítez
Ejercicio 3	Adrián Jiménez Benítez
Ejercicio 4	Antonio José Jiménez Luque
Ejercicio 5 (opcional)	Antonio José Jiménez Luque
Menú	Adrián Jiménez Benítez Antonio José Jiménez Luque
Mantenimiento del Repositorio	Antonio José Jiménez Luque
Realización de la Memoria	Adrián Jiménez Benítez Antonio José Jiménez Luque

Tabla 1. Organización de las tareas.

Para la organización de la realización de la práctica se ha desarrollado un Excel donde se recoge más detalladamente como se ha organizado el equipo para la realización de la práctica.

A continuación, se detallarán algunas pautas que se han seguido para la elaboración de las pruebas experimentales.

2.1. Realización de las pruebas experimentales.

Para la realización de las pruebas experimentales, se va a utilizar un equipo portátil. Debido a errores por el tamaño de memoria asignado en el almacenamiento dinámico de Java se ha modificado el tamaño de almacenamiento dinámico de Java, con referente en *-Xms* (para establecer el tamaño de almacenamiento dinámico inicial) y *-Xmx* (para establecer el tamaño máximo de almacenamiento dinámico) en los siguientes parámetros:

- Xms28GB
- Xmx28GB

Con esta configuración podremos realizar pruebas con variables de mayor tamaño. Con respecto al equipo utilizado, se ha utilizado para las pruebas un equipo (portátil) de un miembro del equipo, debido a que para estas pruebas en concreto no sería lo idóneo mezclar ejecuciones en equipos diferentes. Las propiedades del equipo se detallan en la siguiente tabla:

Equipo	**Portátil**
Sistema Operativo (SO)	Windows 11 Pro
Modo	Rendimiento
Estado (Batería / Corriente)	Corriente
CPU	12th Gen Intel i7-12700H 14 Núcleos

	- 6 Perfomance-cores - 8 Efficient-cores Total de subprocesos: 20 Frecuencia turbo máxima 4.70 GHz
GPU	Nvidia GeForce RTX 3060 Laptop GPU
Memoria RAM	32GB DDR5
Aplicación Utilizada	Eclipse IDE for java Developers Versión 2023-12
JDK	Java SE 17

Tabla 2. Propiedades equipo utilizado.

Las pruebas se han realizado como se muestra en la tabla con el perfil de Rendimiento y el equipo siempre conectado a corriente. Para las pruebas se han ejecutado el algoritmo 10 veces en las cuales hemos sacado el tiempo medio para así poder suavizar las irregularidades que hemos podido tener en cada una de las ejecuciones por motivos del sistema operativo u motivos externos.

2.2. Realización de las pruebas.

Para la realización de las pruebas experimentales de rendimiento del Ejercicio 1, Ejercicio 2 y del Ejercicio 3 se han realizado sobre las redes reales de EDAland (reducido y completo), en el caso del Ejercicio 4 se han generado nuevas redes aleatorias con el generador de grafos implementado. Para estas pruebas se han elegido como vértices 500 (24.950 aristas), 1000 (99.900 aristas), 1500 (224.850 aristas), 2000 (399.800 aristas), 2500 (624.750 aristas) y 3000 (899.700 aristas). Estos valores proporcionarán una densidad constante del 20% en comparación con un grafo completamente denso para cada tamaño de vértices y son adecuados para evaluar la eficiencia y el comportamiento de los distintos algoritmos implementados en escenarios de densidad moderada. Se ha elegido esta representación manteniendo una densidad del 20% calculada de la manera siguiente:

$$Número\ de\ aristas = d * \frac{n * (n-1)}{2}$$

2.3. Test.

Para la comprobación del correcto funcionamiento de los algoritmos desarrollados se ha creado la clase *GreedyTest. En* estas pruebas realizamos varias pruebas para comprobar sobre todo que la carga de los archivos es la correcta y que no se cargan los diferentes grafos de manera errónea para evitar en el procesado que ocurra algún error. Para la visualización de los distintos métodos empleados se pueden visualizar en el anexo (véase A.1 Diseño del código). A continuación, se ilustra un ejemplo de test y la correcta ejecución de todos los test.

```
@Test
public void testLoadVertex() {
    RoadNetwork roadNetwork = new RoadNetwork();
    roadNetwork.loadNetwork(roadNetwork.getCarpeta() + "graphEDAland.txt")

    assertEquals(roadNetwork.containsVertex("Almeria"), true);

    assertEquals(roadNetwork.containsVertex("Cordoba"), false);

    assertEquals(roadNetwork.containsVertex("123"), false);
}
```

Ilustración 1. Ejemplo de test.

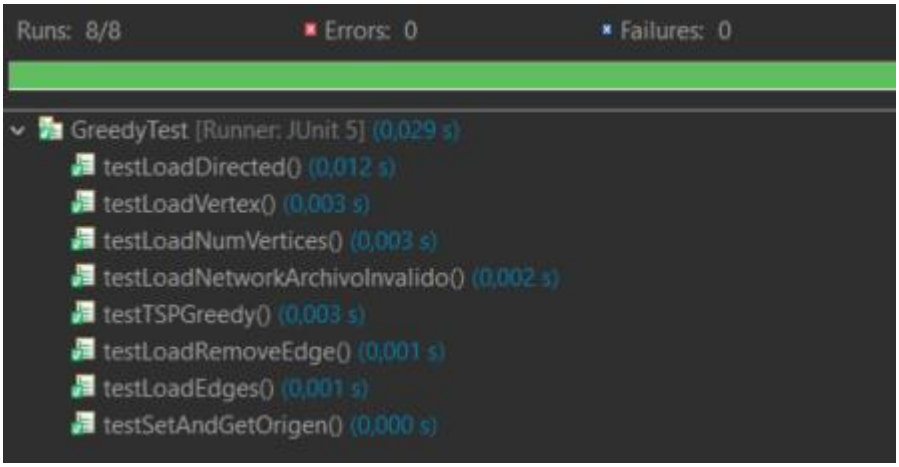

Ilustración 2. Correcta ejecución de los test.

3. Proyecto.

Este documento se ha elaborado con el objetivo principal de profundizar en los conocimientos adquiridos sobre el tema de algoritmos Voraces (Greedy), aplicando la teoría estudiada y presentada para poder aplicarla de manera práctica para solventar diversos problemas que se han presentado en esta práctica para poder realizar instalaciones de fibra óptica.

3.1. Ejercicio 1. Prim.

En este apartado, se va a detallar el primer ejercicio de la práctica, donde determinaremos el árbol de recubrimiento de coste mínimo partiendo desde una ciudad que el operario designe. En este caso nos vamos a centrar en dos distintos casos. El primero, utilizado el algoritmo de *Prim* sin cola de prioridad y otro utilizando el algoritmo de *Prim* con cola de prioridad.

El algoritmo de Prim es una estrategia voraz que busca hallar el árbol de recubrimiento de coste mínimo en un grafo dado. En su funcionamiento, mantiene dos particiones: una, S, con los vértices vinculados por las aristas del árbol en desarrollo, y otra, V-S, con los restantes vértices. Inicialmente, S posee un único vértice, el cual puede ser cualquiera. En cada paso, se incorpora un nuevo vértice al conjunto S hasta que este contenga todos los vértices del grafo.

El algoritmo de Prim (Algoritmo de Prim sin Priority Queue) selecciona en cada iteración la arista de menor coste que conecta un vértice en S con otro en V-S. Para esto, examina todas las aristas que conectan ambos conjuntos. La fase inicialización del algoritmo tiene una complejidad de $O(n)$. Por su parte, el bucle principal se ejecuta n-1 veces, dado que se añade un vértice en cada iteración. En su interior, se lleva a cabo la selección de la arista de menor coste y las operaciones de actualización de las estructuras de datos, resultando en una complejidad total de $O(n^2)$.

Una mejora considerable al algoritmo de Prim (Algoritmo de Prim con PriorityQueue) radica en emplear una cola de prioridad de aristas con peso en lugar de inspeccionar todas las aristas que conectan los conjuntos S y V-S. Con ello, se reduce la complejidad a $O(m \log n)$, siendo m el número de aristas que conforma el grafo. Esta optimización resulta especialmente valiosa en grafos densos, donde el número de aristas se acerca a n^2.

A continuación, subdividiremos en dos los apartados para abordar los distintos casos.

3.1.1. Prim sin PQ.

En este caso, nos centraremos en el algoritmo de Prim para encontrar el árbol de recubrimiento de coste mínimo sin utilizar cola de prioridad. La clase implementada es *Prim(Boolean visual)*. Para este caso, pasamos como argumento un booleano para poder diferenciar si queremos o no mostrar de manera visual los resultados. El origen lo tratamos como una variable de clase llamada *origen*.

i. Estudio teórico.

El algoritmo de Prim implementado sin cola de prioridad busca encontrar el árbol de recubrimiento de coste mínimo de un grafo conexo y no dirigido. Utiliza dos estructuras principales:

- Una lista de aristas para almacenar la solución.
- Una lista de aristas para mantener los candidatos.

Además, utiliza un conjunto para rastrear los vertices ya incluidos en la solución. En cada iteración, busca los candidatos conectados al vertice actual y elige el de menor peso. Luego, añade esa arista a la solución, actualiza los conjuntos de candidatos y vértices utilizados, y avanza al siguiente vertice. Para evitar ciclos, elimina de los candidatos cualquier arista que conecte vértices ya incluidos en la solución.

Para analizar el orden de complejidad se deben de observar los dos bucles anidados:

- Uno que itera sobre los candidatos para encontrar el de menor peso, lo cuál requiere $O(n)$ operaciones en cada iteración principal, siendo n el número de vertices del grafo.
- Y otro que elimina los candidatos para evitar ciclos, que también es $O(n)$ en el peor de los casos.

Dado que el algoritmo recorre todos los vertices, la complejidad total del algoritmo es $O(n^2)$, y como se puede observar, concuerda con el algoritmo de Prim general.

Con respecto al orden espacial se usan estructuras de datos para las aristas candidatas, la solución parcial y los vértices utilizados. Explicado más detalladamente:

- El ArrayList solución almacena las aristas seleccionatas en el árbol de recubrimiento de coste mínimo. En el peor caso, este ArrayList puede contener hasta n-1 aristas.
- El ArrayList candidatos almacena las aristas candidatas que se están considerando en cada iteración. Puede contener como máximo m aristas.
- El TreeSet usadas almacena los vértices que ya han sido incluidos en la solución parcial. En el peor caso, este TreeSet puede contener hasta n elementos.

El orden espacial del algoritmo de Prim sin cola de prioridad es de $O(n + m)$. Esto significa que el algoritmo usa memoria adicional proporcional al número de vértices más el número de aristas en el grafo. Para grafos grandes, esto puede requerir una cantidad significativa de memoria.

En resumen el algoritmo de Prim sigue un enfoque ingenuo para encontrar el arbol de recubrimiento de coste mínimo sin usar una cola de prioridad, lo que resulta en una complejidad cuadrática en el peor caso. Aunque es sencillo de entender y de implementar, puede no ser eficiente para grafos grandes o grafos dispersos, debido a su alto orden de complejidad. En escenarios donde el rendimiento es crítico, considerar el uso de una estructura de datos más eficiente, como una cola de prioridad, podría mejorar significativamente el tiempo de ejecución del algoritmo.

3.1.2. Prim con PQ.

En este caso, nos centraremos en el algoritmo de Prim para encontrar el árbol de recubrimiento de coste mínimo utilizando cola de prioridad. La clase implementada es *PrimPriorityQueue(Boolean visual)*. Para este caso, pasamos como argumento un booleano para poder diferenciar si queremos o no mostrar de manera visual los resultados. El origen lo tratamos como una variable de clase llamada *origen*.

i. Estudio teórico.

El algoritmo de Prim implementado con cola de prioridad utiliza una estructura PriorityQueue para mantener los candidatos ordenados por peso de arista. En cada iteración, agrega todas las aristas conectadas al vértice actual al PriorityQueue. Luego, extrae la arista con el menor peso de la cola de prioridad, asegurándose de que su destino no haya sido utilizado previamente en la solución. Este proceso garantiza que siempre se seleccione la arista más liviana que no cause ciclos en el árbol de recubrimiento de coste mínimo. La complejidad del algoritmo es $O(m \log n)$, siendo:

- m representa el número de aristas en el grafo, en cada iteración del algoritmo, se agregan todas las aristas al PriorityQueue en total, ya que cada arista se agrega una vez durante todo el proceso.
- n es el número de vértices en el grafo. La operación de extracción (poll) de la cola de prioridad tiene una complejidad de $O(\log n)$. Como máximo, este proceso de extracción se realiza una vez por cada una de las m aristas del grafo.

Con respecto al orden espacial, las estructuras de datos que se usa en el algoritmo son:

- La PriorityQueue candidatos que almacena las aristas candidatas ordenadas por su peso. En el peor caso, puede contener hasta m aristas.
- El TreeSet usadas sigue siendo necesario para rastrear los vértices ya utilizados en la solución parcial. En el peor caso, este TreeSet puede contener hasta n elementos.

Como se ha podido determinar de la misma manera que en el algoritmo de Prim sin cola de prioridad, el orden espacial del algoritmo de Prim con cola de prioridad es $O(n + m)$.

En resumen, el algoritmo de Prim con cola de prioridad ofrece una mejora significativa en la eficiencia gracias a la estructura de datos PriorityQueue, lo que resulta en una complejidad temporal más favorable para encontrar el árbol de recubrimiento de coste mínimo.

3.1.3. Pruebas sobre redes de telecomunicaciones de EDAland.

Con respecto a las pruebas sobre las redes de telecomunicaciones de EDAland se detalla en la siguiente tabla con los datos obtenidos según las pruebas realizadas que subdividiremos en dos apartados, una para el grafo reducido y otra para el grafo completo.

- Grafo reducido.

Origen = “Almería”	**Tiempo (milisegundos)**
PRIM	0,1111
PRIM con PQ	0,005

Tabla 3. Datos obtenidos grafo reducido.

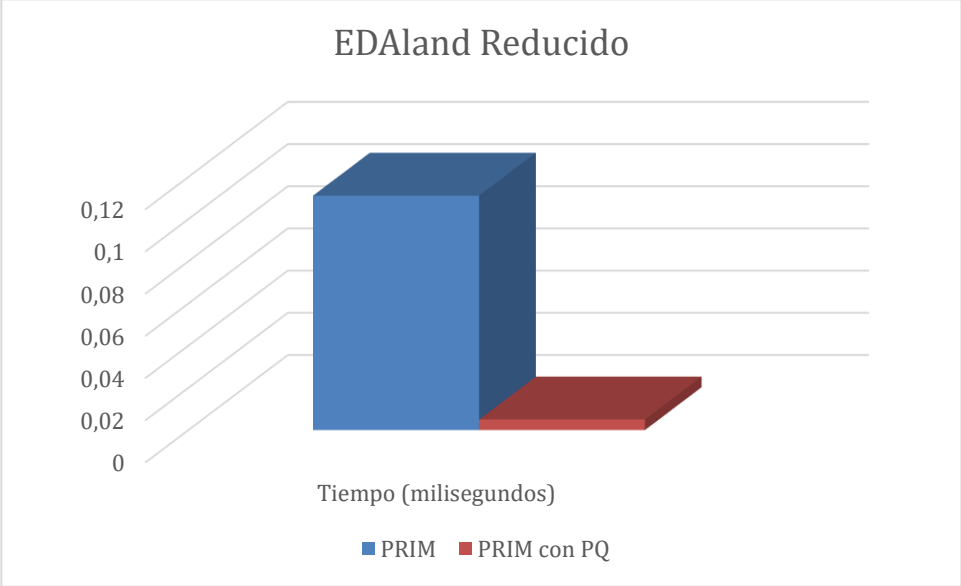

Ilustración 3. Resultados grafo reducido PRIM.

- Grafo completo.

Origen = "1"	**Tiempo (milisegundos)**
PRIM	11,33
PRIM con PQ	1,3333

Tabla 4. Datos obtenidos grafo reducido.

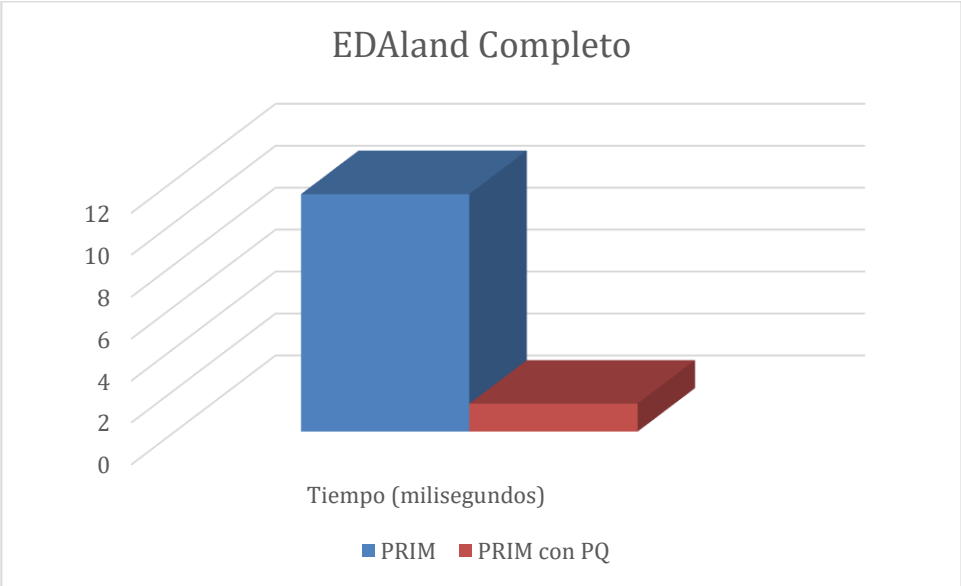

Ilustración 4. Resultados grafo completo PRIM.

Como conclusiones podemos ver que en las dos aplicaciones la variante con cola de prioridad tiene una elevada rapidez con respecto a la variante original.

3.2. Ejercicio 2. Kruskal.

En este caso, nos centraremos en el algoritmo de Kruskal para encontrar el árbol de recubrimiento de coste mínimo utilizando cola de prioridad. La clase implementada es *Kruskal(Boolean visual)*. Para este caso, pasamos como argumento un booleano para poder diferenciar si queremos o no mostrar de manera visual los resultados.

El algoritmo de Kruskal es una estrategia voraz que busca encontrar el Árbol de Recubrimiento de Coste Mínimo en un grafo dado. Inicia con un grafo T que contiene todos los vértices de G pero sin ninguna arista. Durante el proceso, se mantiene un bosque de árboles donde cada árbol inicialmente consiste en un único vértice. Las aristas se examinan en orden creciente según su peso, lo cual puede lograrse eficientemente utilizando una cola de prioridad.

Para seleccionar las aristas y construir el árbol de recubrimiento, se sigue un proceso específico: si una arista conecta dos vértices de árboles distintos, se agrega al conjunto de aristas de T y une ambas componentes en una sola componente conexa. Por otro lado, si la arista conecta vértices dentro del mismo árbol, se descarta para evitar la formación de ciclos en el árbol de recubrimiento. El algoritmo continúa fusionando componentes hasta que todos los vértices de T estén en una única componente conexa, momento en el que se ha encontrado el árbol de recubrimiento de coste mínimo.

En términos de complejidad, ordenar las aristas requiere $O(m \log m)$, equivalente a $O(m \log n)$, donde n es el número de vértices y m es el número de aristas. La inicialización de los n conjuntos disjuntos tiene un orden de $O(n)$. El bucle principal del algoritmo se ejecuta n-1 veces, realizando operaciones de obtener y fusionar conjuntos que, en el peor de los casos, implican 2m operaciones de obtener y n-1 operaciones de fusión, con una complejidad de $O(\log n)$ en conjuntos disjuntos. Esta implementación utiliza una cola de prioridad para ordenar las aristas según su peso.

i. Estudio teórico.

El algoritmo de Kruskal con cola de prioridad busca encontrar el árbol de recubrimiento de coste mínimo de un grafo ponderado y no dirigido. Inicia ordenando todas las aristas del grafo por pesos crecientes utilizando una PriorityQueue. Luego, inicializa conjuntos disjuntos para cada vértice, donde cada conjunto contiene inicialmente solo un vértice. En cada iteración, se selecciona la arista de menor peso de la cola de prioridad y verifica si los vértices que conecta pertenecen a conjuntos disjuntos diferentes. Si es así, se agrega esta arista a la solución y se fusionan los conjuntos disjuntos correspondientes.

Para no tener una acumulación de estructuras de datos del tipo TreeMap<String, TreeSet<String>> en vez de crearse muchas estructuras de datos del estilo TreeSet<String>, se elimina la estructura de datos que sobra al hacer la fusión y se hace referencia en el mismo TreeMap. Esto hace que se termine usando solo una estructura de datos y se pueda realizar fácilmente la búsqueda.

Este proceso continúa hasta que se hayan agregado suficientes aristas para formar el árbol de recubrimiento de coste mínimo, es decir, hasta que el tamaño de la solución sea igual a n-1, siendo n el número de vértices en el grafo.

El orden de complejidad temporal del algoritmo de Kruskal con cola de prioridad es $O(m \log n)$, donde m es el número de aristas y n es el número de vértices en el grafo. La mayor parte del tiempo se gasta en ordenar las aristas inicialmente en $O(m \log n)$.

El orden de complejidad espacial se puede ver de la siguiente manera:

- La PriorityQueue candidatos almacena las aristas candidatas ordenadas por peso. En el peor caso, puede contener hasta m aristas.
- El TreeMap conjuntos almacena conjuntos disjuntos para cada vertice. En el peor caso, puede contener hasta n elementos.

Por lo tanto, el orden espacial del algoritmo de Kruskal es O(n+m). Esto refleja el hecho de que el algoritmo utiliza memoria adicional proporcional al número de aristas más el número de vértices en el grafo.

En resumen, el algoritmo de Kruskal con cola de prioridad ofrece una manera eficiente de encontrar el árbol de recubrimiento de coste mínimo utilizando la estructura de datos PriorityQueue para seleccionar rápidamente las aristas de menor peso. Sin embargo, su orden de complejidad se ve significativamente afectado si el grafo es denso. Por lo que el algoritmo aprovecha muy bien las ventajas que ofrece en grafos dispersos.

ii. Estudio experimental.

Para el estudio experimental se probará también sobre las redes de telecomunicaciones reducida y completa de EDAland, en la siguiente tabla recogemos los datos obtenidos de las pruebas.

EDALand Reducido		
	Tiempo (milisegundos)	Memoria (MB)
KRUSKAL	0,3333	0,014484406

Tabla 5. Datos obtenidos para el grafo reducido de EDAland con Kruskal.

EDALand Completo		
	Tiempo (milisegundos)	Memoria (MB)
KRUSKAL	1,666666667	48,6403389

Tabla 6. Datos obtenidos para el grafo completo de EDAlan con kruskal.

3.3. Ejercicio 3. TSP Backtracking.

El algoritmo TSP Backtracking enfrenta una complejidad exponencial, lo que significa que su tiempo de ejecución aumenta rápidamente a medida que se incrementa el número de ciudades a visitar. Este enfoque se basa en la enumeración de todas las posibles permutaciones de ciudades, comenzando desde una ciudad inicial y explorando recursivamente todas las opciones de ciudades siguientes. Sin embargo, para evitar una exploración exhaustiva, el algoritmo aplica pruebas de factibilidad y corte de ramas. Estas pruebas determinan si la solución parcial actual puede extenderse para formar una solución válida completa. Si no es posible, se poda esa rama del árbol de búsqueda, optimizando así el tiempo de cómputo.

A pesar de garantizar la solución óptima, el enfoque de Backtracking puede volverse ineficiente en instancias grandes del problema. Por lo tanto, suele combinarse con heurísticas para mejorar el rendimiento. Entre estas heurísticas se encuentran la poda alfa-beta y el uso de información de dominio para guiar la búsqueda. Estas técnicas ayudan a reducir el espacio de búsqueda y acelerar el proceso de encontrar la solución óptima en el problema del viajante.

i. Estudio teórico.

El algoritmo de TSP (Traveling Salesman Problem) con Backtracking busca encontrar el camino más corto que debe seguir una unidad de mantenimiento, partiendo desde un origen y visitando todas las ciudades exactamente una vez antes de regresar al origen. Utiliza un enfoque de Backtracking para explorar todas las posibles permutaciones de las ciudades y determinar la ruta óptima. En cada caso, el algoritmo agrega una ciudad a la ruta actual, marca esa ciudad por visitada y continúa explorando las ciudades vecinas no visitadas. Cuando se alcanza el destino y se han visitado todas las ciudades, se evalúa el peso total del camino. Si es menor que el mínimo encontrado hasta ahora, se actualiza el mínimo y se guarda la solución. Esto se repite hasta que se han explorado todas las posibles permutaciones de ciudades.

El orden de complejidad temporal del algoritmo de TSP con backtracking depende de varios factores, incluido el número de ciudades y la estructura del grafo. En el peor caso, donde todas las ciudades son vecinas entre sí, el número total de permutaciones a encontrar es $O(n!)$ donde n es el número de ciudades. Sin embargo, mediante el uso de podas y optimizaciones, como verificar si el peso actual supera el mínimo encontrado hasta ahora, se pueden evitar algunas exploraciones innecesarias, reduciendo efectivamente el tiempo de ejecución del algoritmo.

Con respecto al orden espacial:

- El ArrayList resultado almacena la ruta actual que está siendo explorada en el backtracking. En el peor caso, puede contener hasta n elementos.
- El TreeMap visitados se utiliza para rastrear qué vértices han sido visitados durante el proceso de backtracking. En el peor caso, puede contener hasta n elementos.

Por lo tanto, el orden espacial del algoritmo TSP Backtracking es $O(n)$. Esto refleja el hecho de que el algoritmo utiliza memoria adicional proporcional al número de vértices en el grafo para almacenar la ruta actual y para rastrear qué vértices han sido visitados durante la exploración.

En resumen, el algoritmo de TSP Backtracking es una solución exhaustiva para encontrar la ruta más corta que pasa por todas las ciudades exactamente una vez. Aunque puede ser eficaz para conjuntos pequeños de ciudades, su rendimiento puede degradarse rápidamente con un aumento en el número de ciudades debido al alto orden de complejidad $O(n!)$.

ii. Estudio experimental.

Para el estudio experimental de este ejercicio realizaremos las pruebas para comprobar el correcto funcionamiento en la nueva red de telecomunicaciones reducida. En la siguiente tabla podemos observar los datos obtenidos tras lanzar el algoritmo en la nueva red de telecomunicaciones.

Origen	**Almería**	
Tiempo	51,33333333	Milisegundos
Memoria	0,020172119	MB

Tabla 7. Datos obtenidos nueva red de telecomunicaciones por TSP Backtracking.

3.4. Ejercicio 4. Generador de redes sintéticas.

En este apartado se va a desarrollar el ejercicio 4, este ejercicio se basa en poder generar grandes redes aleatorias (grafos no orientados, valorados positivamente y conexos) de diferentes tamaños. Para la realización de este ejercicio se ha creado una clase *GenerateGraph* que gestiona la creación y visualización del grafo (*véase Ejercicio 5 (Opcional)*). Para el formato de salida en .txt se ha seguido el esquema facilitado en la práctica. En esta clase se generan los grafos comprobando que el número de aristas es válido para las configuraciones de grafo que se nos han impuesto, la comprobación se realiza con el método *esNumeroAristaValido*, el cuál comprueba que el parámetro introducido de aristas este en dentro del rango para que la configuración sea correcta.

El funcionamiento básico del generador es el siguiente:

- En primer lugar, en el constructor se preparan los parámetros *vertice*, *aristas* y *densidad*. La densidad se calcula con el método *calcularDensidad()* el cual se basa en los *vértices y aristas* introducidos.
- Para la generación del grafo se implementa el método *GenerateGraphRandom* el cual genera un ArrayList con el número de vértices. Seguidamente agregamos todas las conexiones posibles de vértices en la variable *edges*. Luego, barajamos con un *suffle()* toda la lista.
- Añadimos las aristas al grafo, de tal manera que se asegure la conectividad del grafo. Después de agregar la conexión en la variable *conexiones*, eliminamos la arista de la lista de aristas. De estas maneras tenemos un grafo conexo. A continuación, obtenemos el número de aristas que faltan para completar la densidad del grafo, de la misma manera para que no se repitan aristas, barajamos la lista de aristas.

- Finalmente, en caso de que sea necesario completarse, se agrega como se ha realizado anteriormente las aristas faltantes para completar el grafo. Cuando el grafo ya está creado, se llama a los distintos métodos para guardar el grafo en formato .txt para su respectivo uso y .dot para su visualización.

Para los estudios teóricos nos basamos en los ejercicios anteriores Prim (*véase Ejercicio1*) y Kruskal (*véase Ejercicio2*). En el siguiente apartado veremos los resultados experimentales de la aplicación de los distintos algoritmos mencionados en los nuevos grafos creados.

3.4.1. Resultados Experimentales.

3.4.1.1.Prim.

i. Prim sin cola de prioridad.

Nodos	Aristas	Tiempo (milisegundos)	Memoria (MB)	Origen
500	24950	422,4444444	0,76994324	1
1000	99900	4072,777778	3,06680298	1
1500	224850	17358	6,88713074	1
2000	399800	45958,22222	12,2339783	1
2500	624750	103916,7778	19,107193	1
3000	899700	202912,3333	27,5065918	1

Tabla 8. Datos obtenidos en Prim sin PQ

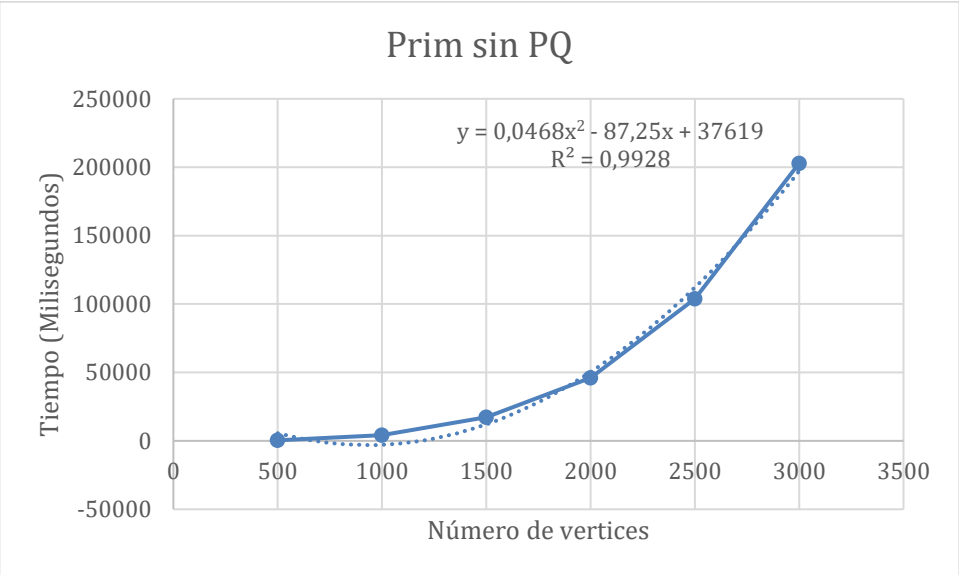

Ilustración 5. Orden temporal Algoritmo Prim sin PQ

Como podemos ver en la *ilustración 5* el orden de complejidad es $O(n^2)$. Corroborando nuestro análisis teórico (*véase Prim sin PQ.*).

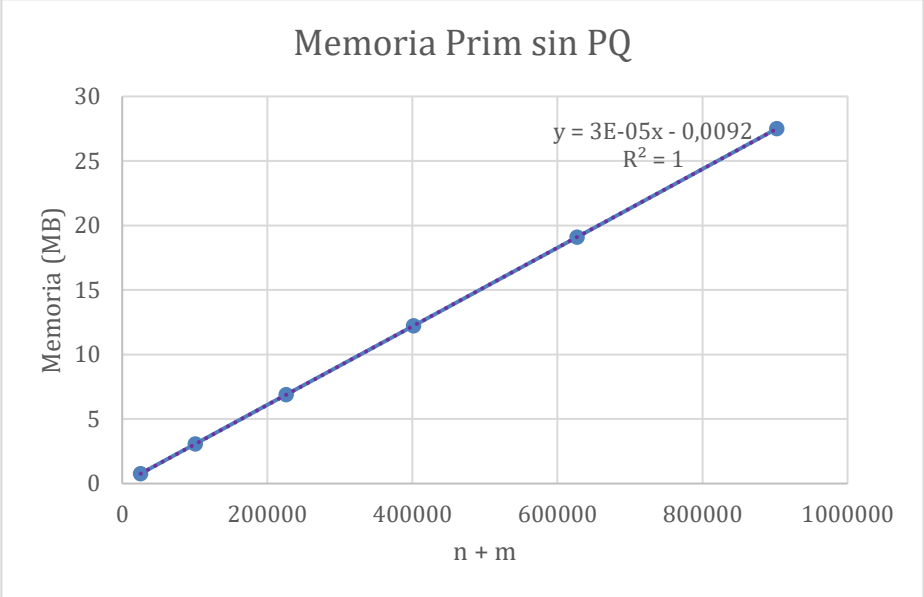

Ilustración 6. Memoria espacial Algoritmo Prim sin PQ

ii. Prim con cola de prioridad.

Nodos	Aristas	Tiempo (Milisegundos)	Memoria (MB)	Origen
500	24950	13,77777778	0,76994324	1
1000	99900	52	3,06680298	1
1500	224850	123,7777778	6,88713074	1
2000	399800	230,2222222	12,2339783	1
2500	624750	368,3333333	19,107193	1
3000	899700	564,2222222	27,5065918	1
3500	1224650	792	37,431778	1
4000	1599600	1145,777778	48,8834229	1
4500	2024550	1448,333333	61,8612518	1
5000	2499500	1927,666667	76,3654175	1

Tabla 9. Datos obtenidos en Prim con PQ

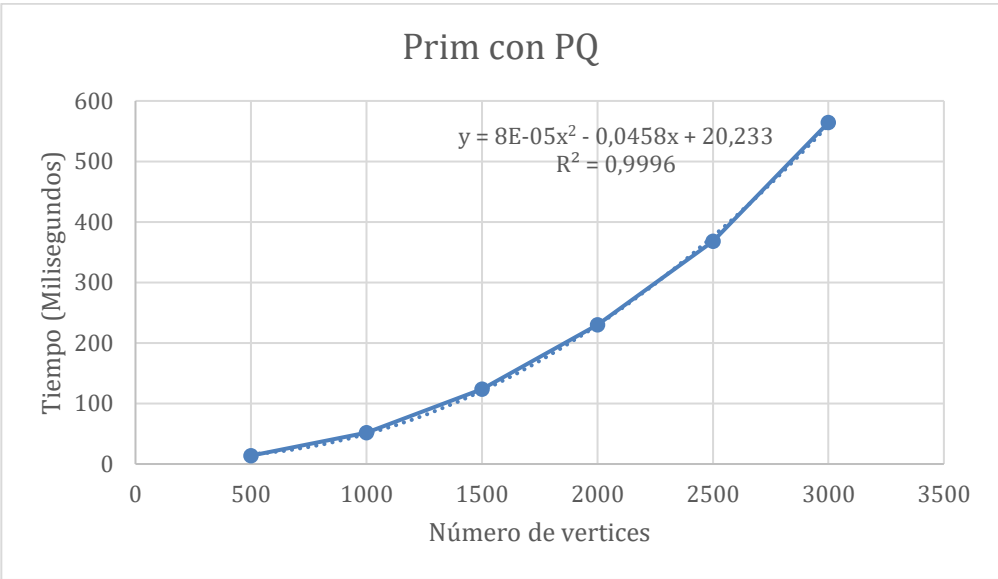

Ilustración 7. Orden temporal Algoritmo Prim con PQ

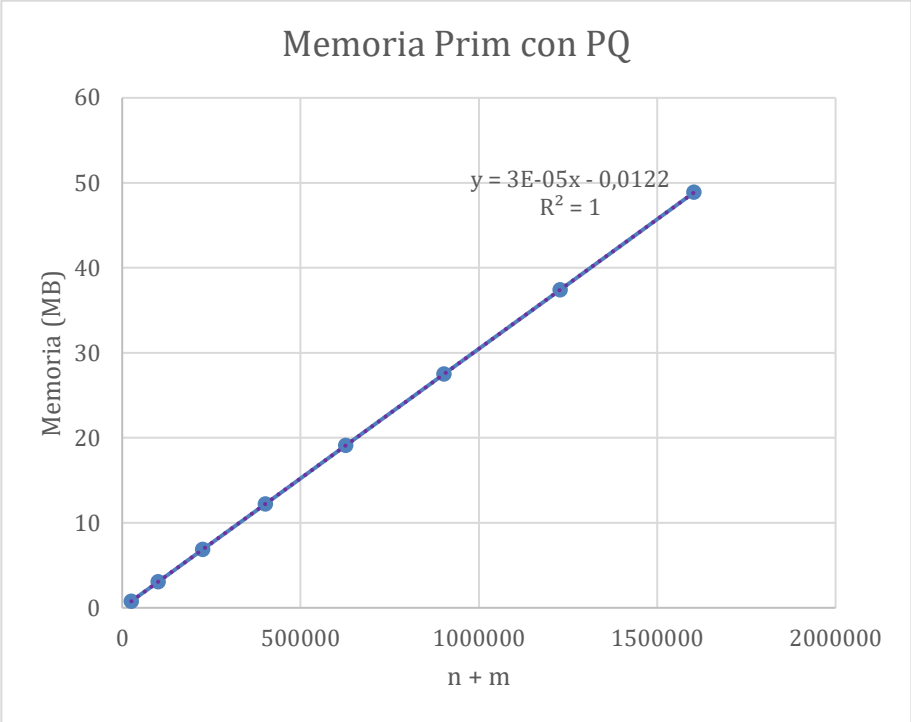

Ilustración 8. Memoria espacial Algoritmo Prim con PQ

3.4.1.2. Kruskal.

n	aristas	tiempo	memoria
500	24950	6,666666667	0,769958496
1000	99900	20,44444444	3,066818237
1500	224850	43,22222222	6,887145996
2000	399800	63,44444444	12,23399353
2500	624750	99,11111111	19,10720825
3000	899700	171,6666667	27,50660706
3500	1224650	222,4444444	37,4317932
4000	1599600	305,6666667	48,8834229
4500	2024550	434,4444444	61,8612518
5000	2499500	552,1111111	76,3654175

Tabla 10. Datos obtenidos Kruskal con PQ

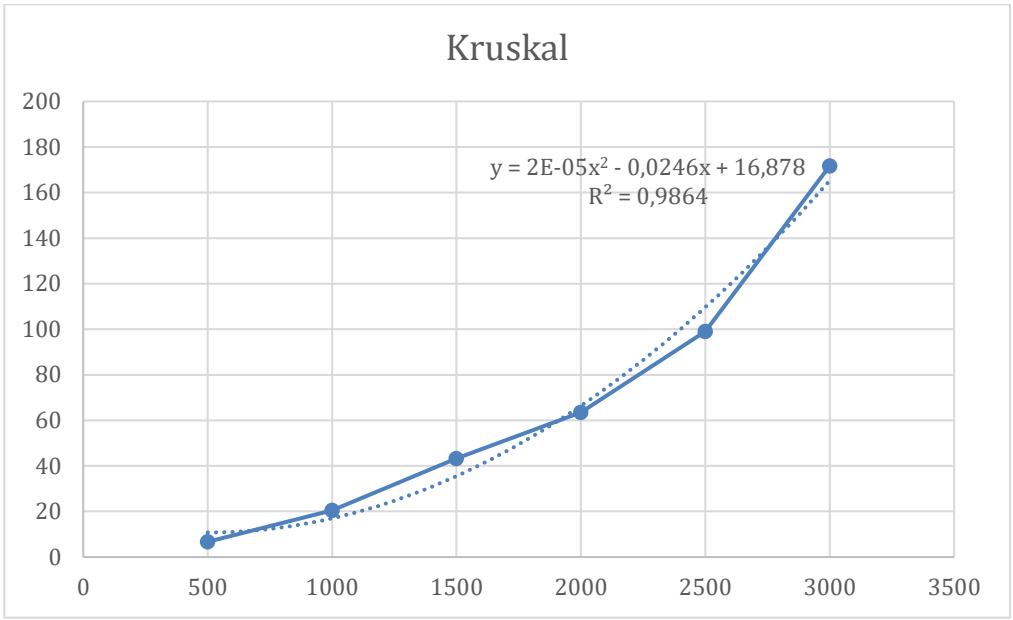

Ilustración 9. Orden temporal Algoritmo Kruskal con PQ

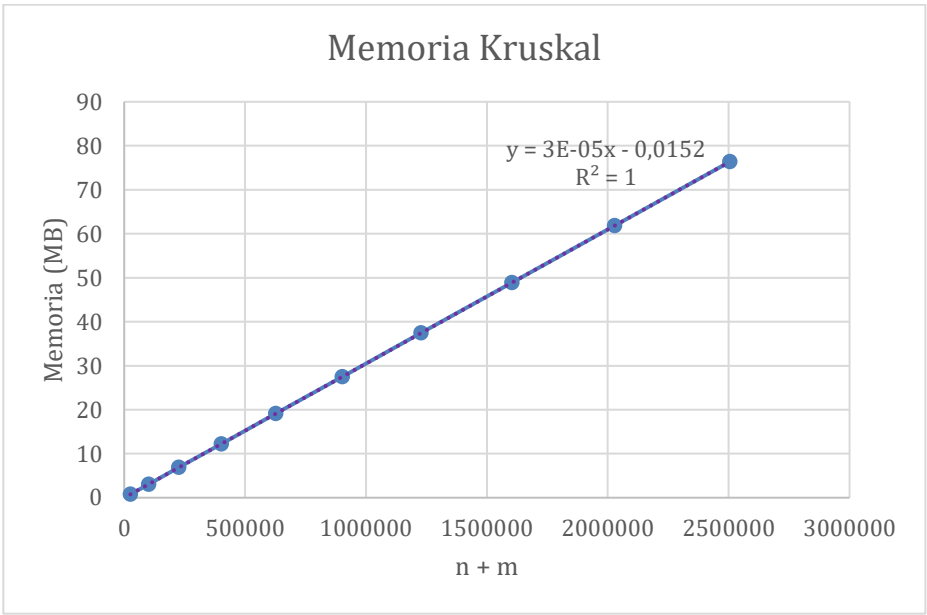

Ilustración 10. Memoria espacial Algoritmo Kruskal con PQ

Los resultados experimentales muestran una aparente discrepancia entre la complejidad teórica esperada y el tiempo de ejecución observado para los algoritmos de Prim y Kruskal con cola de prioridad. Aunque la complejidad teórica es $O(m \log n)$, los datos experimentales sugieren una tendencia cuadrática. Sin embargo, al comparar el tiempo de ejecución con el cálculo de $m * \log n$, se revela una relación lineal, indicando una dependencia consistente entre el tiempo y la complejidad teórica. Esto sugiere que, a pesar de posibles implementaciones subóptimas o condiciones de prueba no representativas, los algoritmos mantienen una relación predecible con respecto al tamaño de entrada y la estructura de los datos. Es crucial realizar un análisis detallado de los resultados experimentales y la implementación para comprender completamente esta discrepancia y considerar posibles mejoras para alinear los resultados experimentales con la complejidad teórica esperada. Para poder realizar el análisis con más detalle quizás se debería tener grafos con mayor cantidad de vértices y aristas, pero debido al bajo volumen computacional que se tiene, puede resultar a ser dificultoso.

En resumen, el orden de complejidad teórico y el análisis experimental, gracias a poder ver que el cálculo de $m * \log n$ es linealmente dependiente con el tiempo de ejecución, podemos deducir que se ha obtenido un resultado satisfactible en el estudio.

A continuación, se va a mostrar cuál es la dependencia de la que se ha hablado anteriormente.

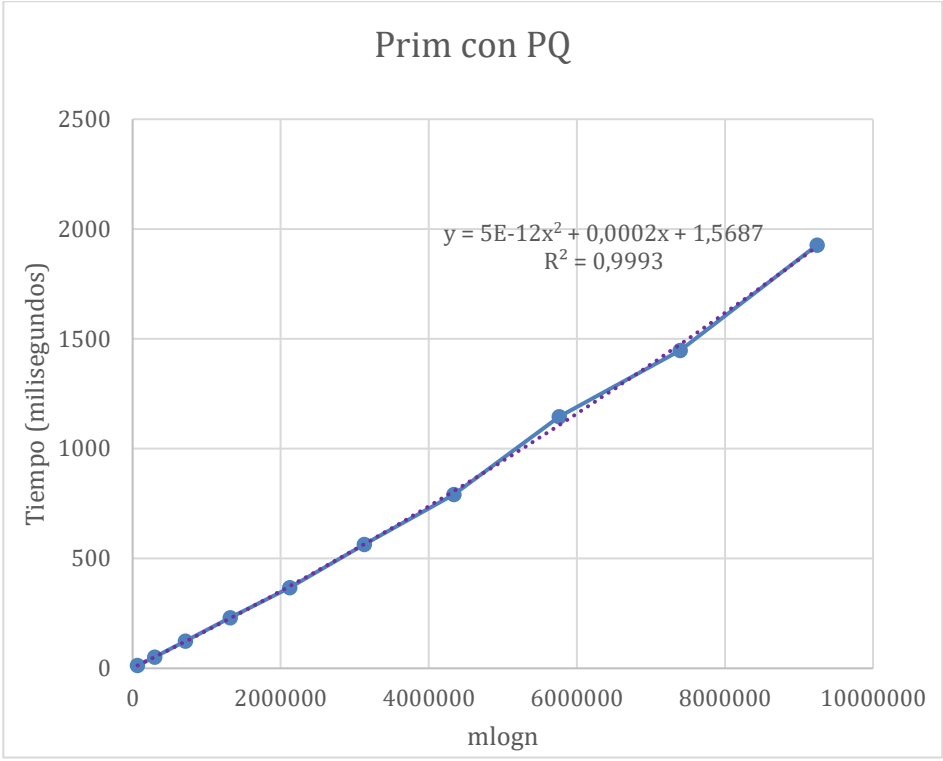

Ilustración 11: Relación mlogn/tiempo Prim PQ

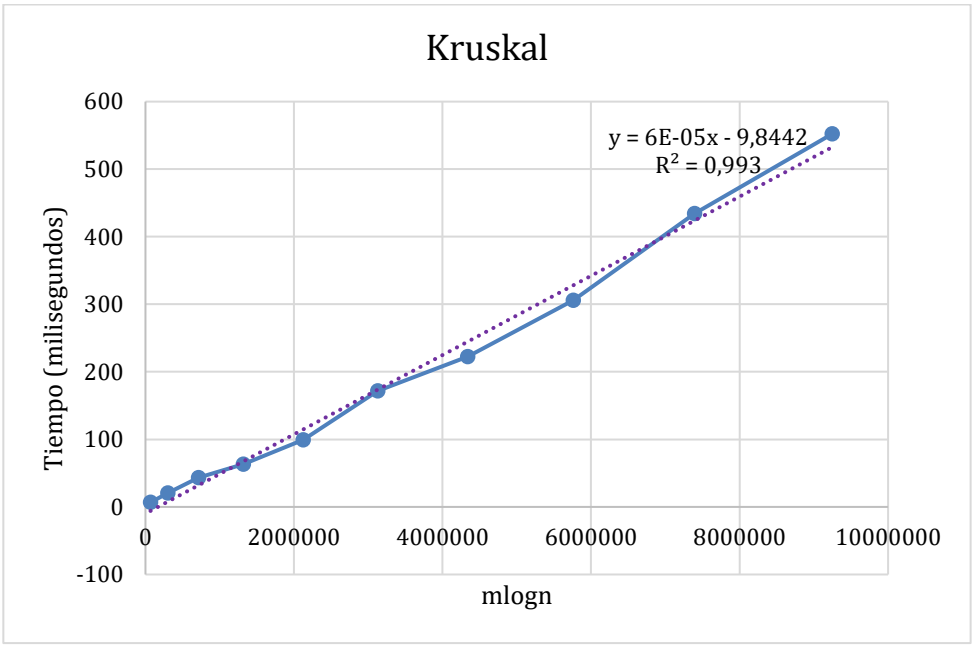

Ilustración 12: Relación mlogn/tiempo Kruskal

3.5. Ejercicio 5 (Opcional).

En este apartado se va a desarrollar el ejercicio 5 (no es obligatorio para nuestro grupo debido a que somos dos miembros). En este ejercicio se realizará la representación gráfica de las soluciones de los distintos ejercicios anteriores utilizando la herramienta *Graphviz. Graphviz* e s una herramienta de software de visualización gráfica de código abierto. Se utiliza principalmente para dibujar gráficos dirigidos, lo que la hace útil para representar estructuras de datos, diagramas de flujo, redes, árboles de decisión, y mucho más. Utiliza un lenguaje de descripción gráfica llamado DOT para definir los gráficos, permitiendo a los usuarios describir nodos, aristas y sus atributos textualmente. Graphviz luego procesa estos archivos DOT para generar visualizaciones en varios formatos gráficos, como PNG, PDF, SVG, entre otros. A continuación, se ilustra un ejemplo del funcionamiento de esta herramienta.

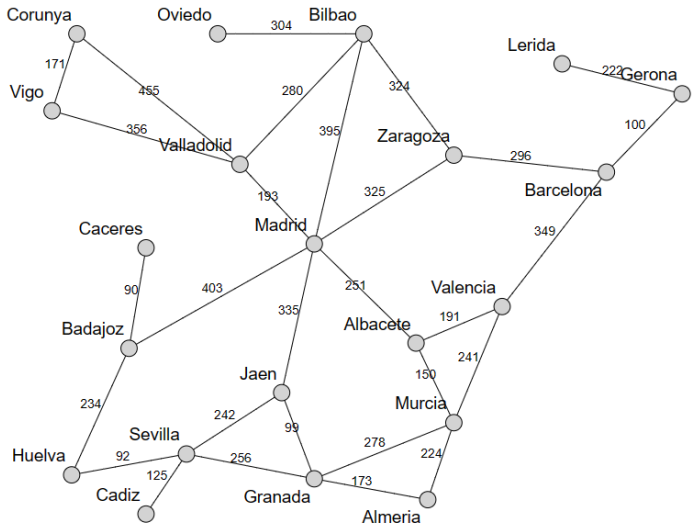

Ilustración 13. Graphviz EDAland reducido.

Para la realización de los archivos, se han realizado dos implementaciones, aunque no se haya pedido implementar el generador de redes sintéticas, una de ellas para la creación de un generador de archivos para visualizar, pero sin marcar el camino mínimo, en este caso al guardar el archivo se genera una cabecera modificada para cada uno de los distintos grafos con las características de cada uno de ellos y diferentes configuraciones que son necesarias para que la aplicación se ejecute correctamente. A continuación, listamos tantos vértices como las diferentes conexiones con la nomenclatura necesaria.

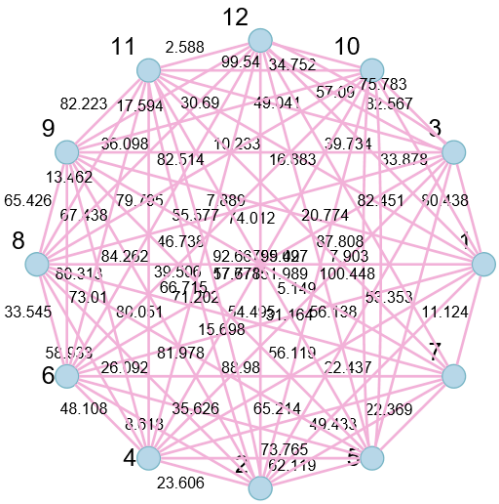

Ilustración 14. Ejemplo visualización grafo.

Se puede realizar diferentes modificaciones para que se vean los grafos con distintas configuraciones, en nuestro caso sería simplemente modificando la variable cabecera.

La siguiente implementación, se dedica a la visualización del camino, que es la que se ha pedido. En este caso una implementación con la misma estructura, la única variación es que comprobamos si las aristas pertenecen al conjunto solución y cuál es el origen. A partir de ahí si es el origen y las aristas pertenecen al conjunto solución se guardan en el archivo con una configuración diferentes (modificación de colores y grosores) para poder ver el camino correctamente. En la siguiente ilustración podemos ver un ejemplo.

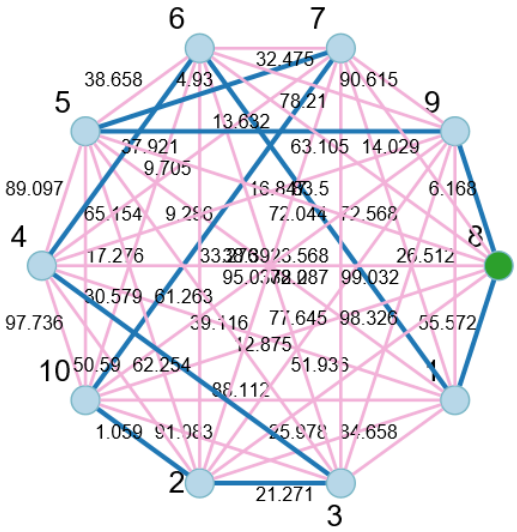

Ilustración 15. Ejemplo visualización camino grafo.

Tanto como para el caso anterior como para el que se va a mostrar a continuación, que es la resolución de Prim, se puede observar cómo se ha marcado el camino mínimo resultante del algoritmo. Tanto para los dos que se han mostrado en la memoria de la práctica, como para todos los demás, funciona correctamente y se puede ver claramente cuál es el camino mínimo que se obtiene.

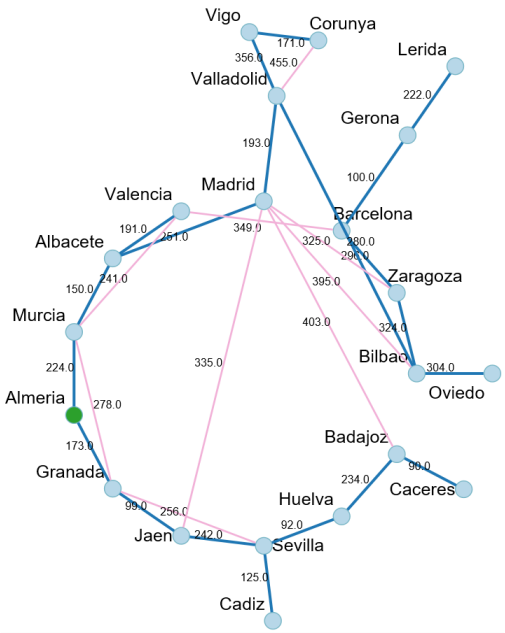

Ilustración 16: Ejemplo Prim visualización EDAland Reducido

i. Estudio experimental.

Para la realización del estudio experimental de este ejercicio el cuál comprobará el correcto funcionamiento del algoritmo implementado para resolver el problema del viajante de comercio (TSP) con la técnica algorítmica Greedy, destacando el error cometido al utilizar la heurística con respecto a la solución exacta (que utiliza caminos *simples*). En nuestro caso, al utilizar un algoritmo de Backtracking el cual necesita un elevado coste de tiempo como se ha explicado (*véase* Ejercicio 3. TSP Backtracking.), utilizaremos unos valores de n no muy grandes (4,5,6,7,8,9,10,11,12). En las siguientes tablas detallamos los distintos datos obtenidos en la ejecución mediante Greedy y mediante Backtracking.

- TSP Backtracking.

TSPBacktracking			
Origen = 1			
n(vértices)	**aristas**	**tiempo (milisegundos)**	**pesos**
4	6	0,111111111	177,758
5	10	0,111111111	177,758
6	15	0,222222222	177,758
7	21	0,666666667	163,959
8	28	4,222222222	106,875
9	36	26,33333333	106,875
10	45	204,6666667	106,875
11	55	2357,333333	106,875
12	66	26354	120,156

Tabla 11. Datos obtenidos para TSP Backtracking.

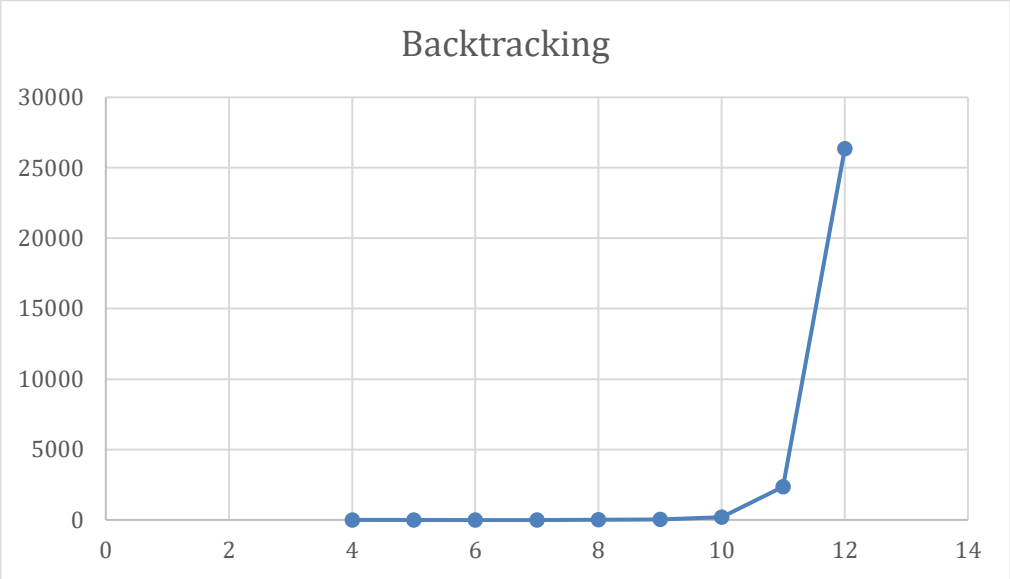

Ilustración 17. Gráfica para TSP Backtracking.

Como podemos ver en la ilustración el consumo de tiempo del algoritmo es muy elevado en comparación con la implementación por Greedy.

- TSP Greedy.

n	aristas	tiempo (milisegundos)	pesos	Error cometido (%)
4	6	0	177,758	1,60E-14
5	10	0	236,711	33,1647521
6	15	0	284,059	59,8009654
7	21	0	183,329	11,8139291
8	28	0	139,014	30,0715789
9	36	0	234,778	119,675322
10	45	0,222222222	178,77	67,2701754
11	55	0,222222222	149,735	40,102924
12	66	0	257,1	113,971837

Tabla 12. Datos obtenidos para TSP Greedy.

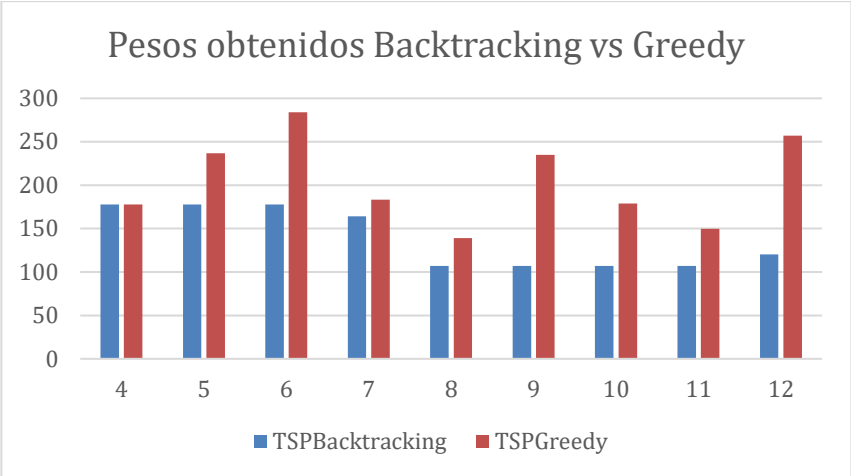

Ilustración 18. Gráfica de diferencia de pesos.

Los datos muestran que para problemas pequeños (con un número menor de ciudades), el error cometido por el algoritmo Greedy es muy bajo o incluso inexistente. Sin embargo, a medida que el número de ciudades aumenta, el error tiende a crecer, lo que sugiere que el algoritmo Greedy se vuelve menos preciso y es más probable que ofrezca soluciones subóptimas para problemas más grandes. Esto es típico de los algoritmos heurísticos como el Greedy.

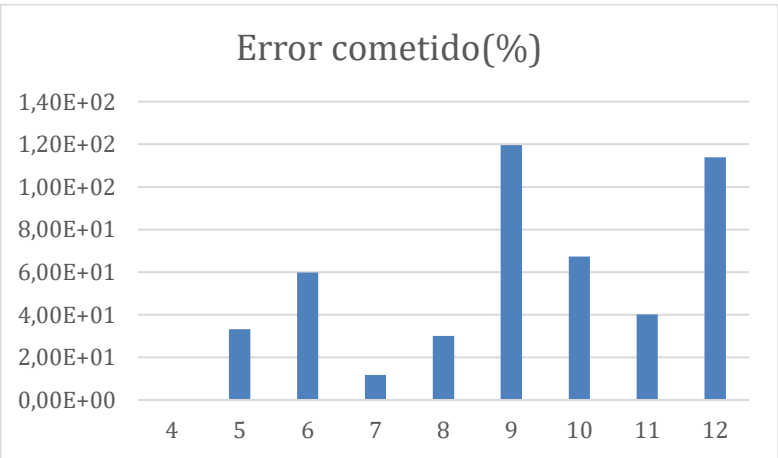

Ilustración 19. Gráfica de errores cometidos.

A. Anexo.

A.1. Diseño del código.

Diagramas de clases.

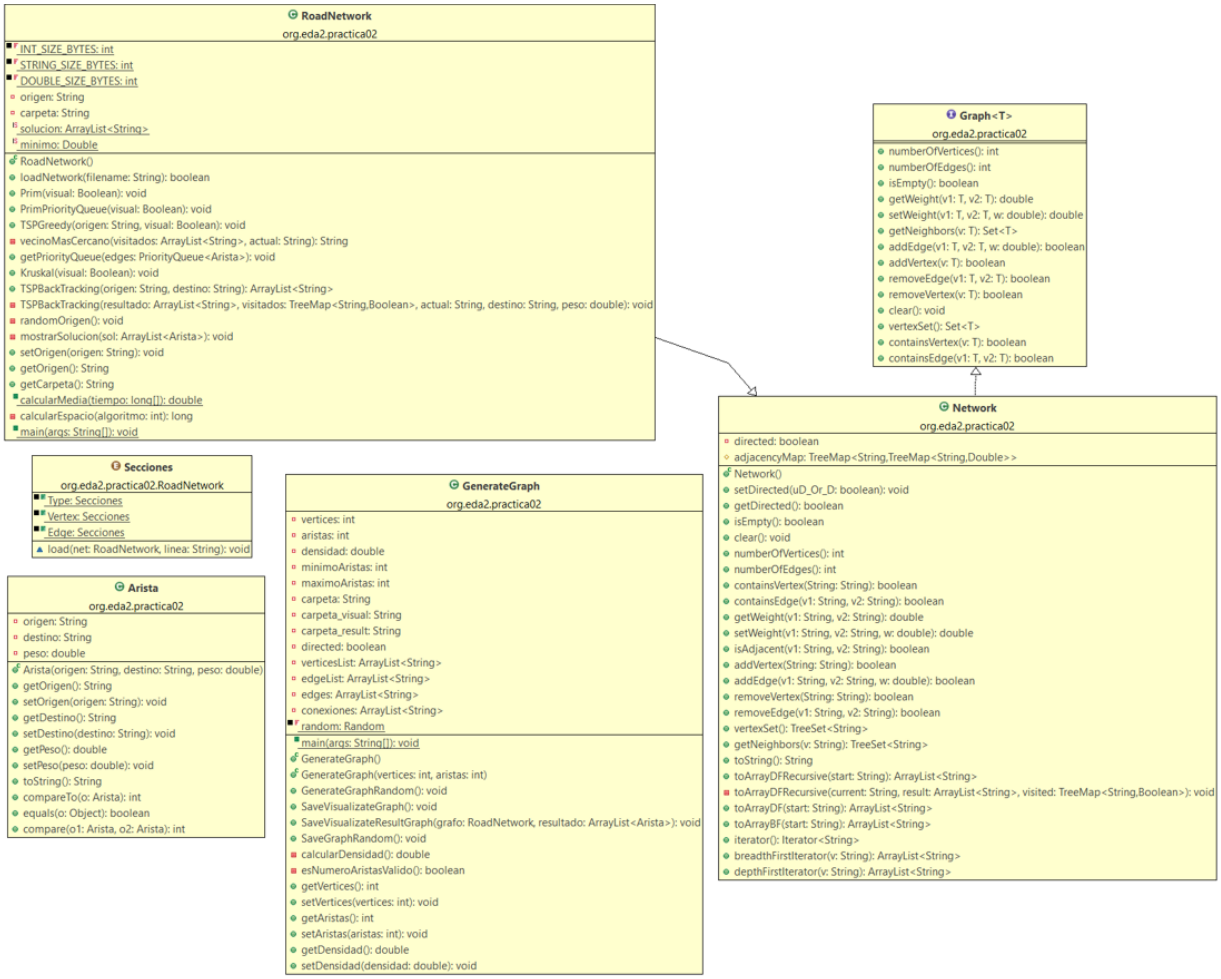

Ilustración 20. Diagrama de Clases.

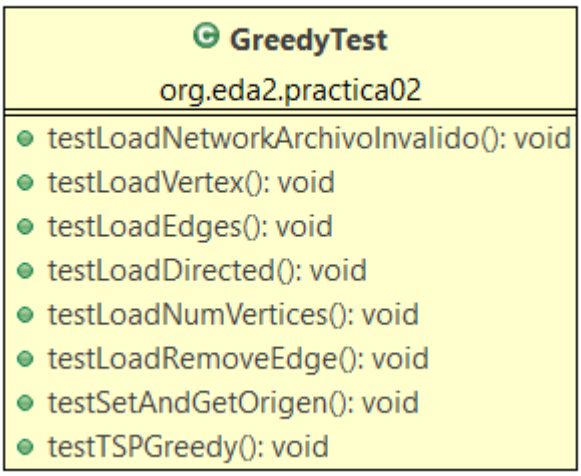

Ilustración 21. Diagrama de clases asociados a los test.

A.2. Esquema archivos fuente.

El esquema de archivos fuente es el siguiente:

Arista.java

Ruta: main\java\org\eda2\practica02\Arista.java

Descripción: Este archivo contiene la implementación para poder utilizar las implementaciones y tener un uso mas eficiente con las respectivas aristas de los grafos.

Network.java

Ruta: main\java\org\eda2\practica02\Network.java

Descripción: Este archivo contiene la implementación para poder tratar con los distintos grafos.

RoadNetwork.java

Ruta: main\java\org\eda2\practica02\RoadNetwork.java

Descripción: Este archivo contiene las implementaciones de los distintos algoritmos y el menú para poder dar uso y comprobar el correcto funcionamiento de las ejecuciones para los distintos ejercicios.

GenerateGraph.java

Ruta: test\java\org\eda2\practica02\GenerateGraph.java

Descripción: Este archivo contiene las implementaciones para la creación de los distintos archivos para poder generar los grafos y sus correspondientes visualizaciones.

Dataset

Ruta: main\java\org\eda2\practica02\dataset\

Descripción: Esta carpeta contiene los archivos generados con los grafos, tanto la estructura del grafo como las distintas representaciones en carpetas anidadas.

Memoria.docx

Ruta: docs\practica02\Memoria.docx

Descripción: Este archivo contiene la memoria de la práctica. Incluye una descripción detallada del problema abordado, el enfoque utilizado para resolverlo, los resultados obtenidos y cualquier otra información relevante relacionada con la práctica.

PR2_Estudio_Experimental.xlsx

Ruta: docs\practica02\PR2_Estudio_Experimental.xlsx

Descripción: Este archivo Excel alberga los cálculos y datos recopilados durante el estudio experimental realizado como parte de la práctica. Puede incluir tablas, gráficos u otros elementos visuales que representen los resultados de las pruebas realizadas.

PR2_Tareas_a_Repartir.xlsx

Ruta: docs\practica02\PR2_Tareas_a_Repartir.xlsx

Descripción: Este archivo Excel contiene la distribución de tareas asignadas a cada uno de los compañeros que trabajan en la práctica. Incluye una lista de tareas específicas y las personas responsables de completar cada tarea.

B. Bibliografía.

Temario de la asignatura.