UNIVERSIDAD DE ALMERÍA

Proyecto 2 Instalación de fibra óptica

Estructura de datos y algoritmos II

equipo-ualeda2-2024-simon-antonio

Venzal Sánchez, Simón María
Guerrero Crespo-López, Antonio

Índice

Proyecto 2 Instalación de fibra óptica

1.- Objetivo.

1.1 Introducción teórica de algoritmos greedy.

Los algoritmos greedy o algoritmos voraces tienen un enfoque para la resolución de problemas en problemas de optimización, la estructura general para este tipo de algoritmos no queda totalmente definida ya que depende del enfoque y el problema a resolver.

Estructura general para problemas con greedy

Función algoritmo_voraz(problema):
  # Inicializar solución vacía
  solución ← conjunto vacío
 	
  # Mientras queden elementos en el problema
  mientras problema ≠ vacío:
  # Seleccionar el mejor elemento actual según la heurística
  mejor_elemento ← seleccionar_mejor_elemento(problema)
 	
  # Agregar el mejor elemento a la solución
  solución ← solución ∪ mejor_elemento
 	
  # Eliminar el mejor elemento del problema
  problema ← problema - mejor_elemento
 	
  # Devolver la solución final
  return solución
 	
Función seleccionar_mejor_elemento(problema):
  # Implementar la heurística para seleccionar el mejor elemento del problema
  # La heurística debe considerar el problema específico y la solución actual
 	
  # Devolver el mejor elemento seleccionado
  return mejor_elemento

Tabla 1: Estructura general de algoritmos voraces.

Un ejemplo clásico de un algoritmo greedy es el problema del cambio de monedas, en este problema, el objetivo es dar cambio utilizando el menor número de monedas posibles.

Esta práctica cuenta con diferentes tipos de implementaciones de algoritmos greedy tales como:

- Árboles de recubrimiento mínimo: Prim y Kruskal.
- El problema del viajante: Caminos simples y TSP.

A esto se debe añadir un generador de redes sintéticas para probar los diferentes algoritmos junto a otras redes del mapa nacional.

Imagen 1: Red de telecomunicaciones completa de EDAland, generado localmente con Graphviz.

1.2 División del trabajo.

La metodología de trabajo en esta segunda práctica es la misma que en la primera, en la que ambos hemos trabajado en todo el proyecto, aportando soluciones como corrección de errores y trabajo en la memoria.

2.- Antecedentes.

2.1 Motivación.

La principal motivación que fundamenta esta y todas las prácticas del curso es conocer herramientas con las que enfocar los problemas desde diferentes puntos de vista, no solo una estructura eficiente hace un buen codigo, tambien el enfoque adecuado nos aproxima a la solución óptima que estamos buscando, en este caso se aplican los conceptos de teoría de grafos para una red de puntos para el supuesto práctico de una instalación de fibra óptica.

2.2 Alternativas planteadas.

Los algoritmos greedy poseen una amplia documentación por lo que en el transcurso de esta práctica hemos realizado diferentes implementaciones.

Por ejemplo para prim sin cola de prioridad inicialmente usamos una LinkedList, no obstante pasamos a una estructura general con los HashMap.

Prim sin cola de prioridad inicial

```
Función prim(origen):
   solucion = Resultado(origen)
   activada = ListaEnlazada()
   verticesSol = Conjunto()
   verticesSol.agregar(origen)

   actual = origen
   Mientras tamaño(solucion) < numeroVertices() - 1:
      vecinos = obtenerVecinos(actual)
      Para cada destino en vecinos:
         Si destino está en verticesSol:
            Continuar
         peso = obtenerPeso(actual, destino)
         activada.agregar(Arista(actual, destino, peso))

      aristaMenor = Arista(nulo, nulo, ValorMaximo)
      Para cada arista en activada:
         Si arista.getPeso() < aristaMenor.getPeso():
            aristaMenor = arista

      solucion.agregarArista(aristaMenor)
      activada.remover(aristaMenor)
      verticesSol.agregar(aristaMenor.getDestino())
      actual = aristaMenor.getDestino()

      it = activada.iterador()
      Mientras it.tieneSiguiente():
         arista = it.siguiente()
         Si verticesSol.contiene(arista.getOrigen()) Y
verticesSol.contiene(arista.getDestino()):
            it.remover()

   Retornar solucion
```

Tabla 2:Pseudocódigo versión inicial de Prim sin cola de prioridad.

Prim con cola de prioridad inicial

```
Función primPQ(origen):
   solucion = Resultado(origen)
   activada = ColaPrioridad()
   verticesSol = Conjunto()
   verticesSol.agregar(origen)

   actual = origen
   Mientras tamaño(solucion) < numeroVertices() - 1:
      vecinos = obtenerVecinos(actual)
      Para cada destino en vecinos:
         Si destino está en verticesSol:
            Continuar
         peso = obtenerPeso(actual, destino)
```

```
            activada.agregar(Arista(actual, destino, peso))

        aristaMenor = activada.extraerMenor()
        solucion.agregarArista(aristaMenor)
        verticesSol.agregar(aristaMenor.getDestino())
        actual = aristaMenor.getDestino()

        it = activada.iterador()
        Mientras it.tieneSiguiente():
            arista = it.siguiente()
            Si verticesSol.contiene(arista.getOrigen()) Y
verticesSol.contiene(arista.getDestino()):
                it.remover()

    Retornar solucion
```

Tabla 3: Pseudocódigo versión inicial de Prim con cola de prioridad.

2.3 Solución final.

La solución final adoptada para los diferentes algoritmos voraces a implementar se ha realizado atendiendo a reducir el tiempo de ejecución y usar el mismo tipo de estructura a lo largo del proyecto facilitando la comprensión e implementación del mismo.

En este caso una LinkedList mantiene el orden de inserción de los elementos, lo que puede ser útil en algunos casos pero no en este, ya que la búsqueda de elementos en un HashMap es muy rápida, con un tiempo promedio de O(1), gracias a la estructura de tabla hash que permite acceder directamente al elemento deseado.

3.- Trabajo a desarrollar.

3.1 Estudio de la implementación.

3.1.1 Prim sin cola de prioridad.

Esta versión propuesta del algoritmo de Prim comienza inicializando: solución, una instancia de la clase Resultado. “claves” es un HashMap que mantiene la menor clave (peso mínimo) para cada vértice. Inicialmente, todos los vértices excepto el origen se inicializan con el valor máximo de la variable con Double.MAX_VALUE. “padres” es un HashMap que mantiene el vértice padre de cada vértice en el árbol de expansión. Esto ayuda a rastrear el árbol de expansión. Y por último visitados es un HashSet que recoge los vértices ya utilizados.

Asignamos el peso 0 al origen para que sea el primer seleccionado.

Se ejecuta un bucle hasta que todos los vértices han sido visitados, con las siguientes instrucciones:

- Se selecciona el vértice con la menor clave (peso) que no ha sido visitado, utilizando el método claveMenor.
- Si no hay vértices accesibles (todos los no visitados son infinitos), el bucle se detiene.
- Se marca el vértice seleccionado como visitado.
- Se revisan todos los vecinos del vértice seleccionado para actualizar sus pesos y padres si se encuentra una conexión con menor peso a través del vértice seleccionado.
- Si el vecino no ha sido visitado y el peso de la conexión actual es menor que el peso anterior, se actualiza su peso y se registra el vértice actual como su padre.

Finalmente y tras haber rellenado el árbol de expansión mínimo con los padres y las claves, se recorren padres con un Entry para añadir las aristas al resultado. Cada arista se añade a la solución con el peso correspondiente entre el vértice padre (destino) y su valor (desde).

Devuelve un Resultado que contiene el árbol de expansión mínimo con todos sus vértices y aristas, así como el peso total mínimo para conectar todos los vértices partiendo del origen.

Pseudocódigo prim sin cola de prioridad

```
Función primV2(origen):
    solucion = Nuevo Resultado(origen)
    claves = Nuevo HashMap<String, Double>()
    padres = Nuevo HashMap<String, String>()
    visitados = Nuevo HashSet<String>()

    Para cada vertice en obtenerVertices():
        claves[vertice] = ValorMaximoDouble
    Fin Para

    claves[origen] = 0.0

    Para i desde 0 hasta numeroVertices():
        desde = claveMenor(claves, visitados)
        Si desde == null, entonces
            Detener // Añadido para manejar si no hay más vértices accesibles
        Fin Si
        visitados.agregar(desde)

        Para cada vecino en obtenerVecinos(desde):
            Si vecino no está en visitados y obtenerPeso(desde, vecino) <
claves.obtenerODefecto(vecino, ValorMaximoDouble), entonces
                claves[vecino] = obtenerPeso(desde, vecino)
                padres[vecino] = desde
            Fin Si
        Fin Para
    Fin Para

    Para cada entrada en padres:
```

```
        destino = entrada.clave
        desde = entrada.valor
        Si desde != null y destino != null, entonces
            solucion.agregarArista(Nueva Arista(desde, destino, obtenerPeso(desde,
destino)))
        Fin Si
    Fin Para

    Devolver solucion
Fin Función
```

Tabla 4: Pseudocódigo prim sin cola de prioridad.

3.1.2 Prim con cola de prioridad.

Esta versión propuesta del algoritmo de Prim con cola de prioridad usa un TreeMap para manejar las claves.

Comienza inicializando una nueva instancia de Resultado como solución, Un TreeMap de claves que contiene como clave los pesos mínimos con los conjuntos de vértices que tienen ese peso. Un HashMap padres y un HashSet de vértices visitados.

Asignamos el peso 0 al origen para que sea el primer seleccionado y se añade a padres.

Se ejecuta un bucle mientras haya vértices no visitados, se selecciona el vértice con la menor clave disponible. Este vértice es extraído de la cola de prioridades gestionada por el TreeMap claves y añade a visitados.

Para cada vecino del vértice seleccionado, se comprueba si el vecino no ha sido visitado y si el nuevo peso propuesto es menor que el peso actual de ese vecino. Si se cumplen ambas condiciones, se actualiza la clave del vecino al nuevo peso y se registra el vértice actual como su padre. Si el peso viejo es diferente de null y ya no es relevante (es decir, hay un nuevo peso menor), se elimina el vecino del conjunto de claves anteriores en el TreeMap.

Después de haber procesado todos los vértices, se recorren los registros de padres para añadir las aristas correspondientes al resultado, conectando cada vértice con su padre registrado en el árbol de expansión. Cada arista se añade con el peso correspondiente entre el vértice padre y el vértice hijo.

Por último devuelve la solución.

Pseudocódigo prim con cola de prioridad

Función primPQV2(origen):
  solucion = Nuevo Resultado(origen)
  claves = Nuevo TreeMap<Double, HashSet<String>>()
  padres = Nuevo HashMap<String, String>()
  visitados = Nuevo HashSet<String>()

```
   claves.poner(0.0, Nuevo HashSet<>())
   claves.obtener(0.0).agregar(origen)
   padres.poner(origen, null)

   Para i desde 0 hasta numeroVertices():
      desde = menorClavePQ(claves, visitados)
      Si desde == null, entonces
         Detener // Si no hay más vértices para procesar, salir del bucle
      Fin Si
      visitados.agregar(desde)

      Para cada vecino en obtenerVecinos(desde):
         Si vecino no está en visitados, entonces
            peso = obtenerPeso(desde, vecino)
            padre = padres.obtener(vecino)
            pesoAntiguo = null
            Si padre != null, entonces
               pesoAntiguo = obtenerPeso(padre, vecino)
            Fin Si

            Si pesoAntiguo == null o peso < pesoAntiguo, entonces
               Si claves.contieneLlave(pesoAntiguo), entonces
                  claves.obtener(pesoAntiguo).eliminar(vecino)
                  Si claves.obtener(pesoAntiguo).estaVacio(), entonces
                     claves.eliminar(pesoAntiguo)
                  Fin Si
               Fin Si
               temp = claves.obtener(peso)
               Si temp == null, entonces
                  claves.poner(peso, Nuevo HashSet<>())
                  temp = claves.obtener(peso)
               Fin Si
               temp.agregar(vecino)
               padres.poner(vecino, desde)
            Fin Si
         Fin Si
      Fin Para
   Fin Para

   // Añadir aristas al resultado final
   Para cada arista en padres:
      Si arista.valor != null, entonces // Evitar añadir la arista para el origen que no tiene
padre
         solucion.agregarArista(Nueva Arista(arista.valor, arista.llave,
obtenerPeso(arista.valor, arista.llave)))
      Fin Si
   Fin Para

   Devolver solucion
Fin Función
```

Tabla 5: Pseudocódigo prim con cola de prioridad.

3.1.3 Kruskal.

Esta implementación del algoritmo de Kruskal comienza creando una nueva instancia de la clase Resultado para almacenar el árbol de expansión mínima. Utiliza una cola de prioridades para ordenar todas las aristas del grafo por su peso, permitiendo procesar siempre la arista de menor peso en primer lugar. Adicionalmente, emplea dos estructuras HashMap, vértices y grupos.

Cada vértice se inicializa en su propio conjunto en vértices, apuntando a sí mismo, indicando que es su propio líder inicialmente. En grupos, cada vértice comienza con un grupo de tamaño uno. Se iteran todas las aristas del grafo, añadiendoles a la PQ, pero sólo se consideran en una única dirección para evitar duplicados.
Mientras el tamaño del resultado sea menor que el número de vértices menos uno, se extrae la arista de menor peso de la cola. Se utilizan las raíces de los vértices conectados por la arista para determinar si están en el mismo conjunto. Si no lo están, se añade la arista al resultado, y se unen los conjuntos de los dos vértices, actualizando el representante y el tamaño del conjunto en vértices y grupos respectivamente, eligiendo siempre el representante del conjunto más grande como líder.
Este proceso continúa hasta que se han incluido suficientes aristas para conectar todos los vértices sin cerrar ciclos, asegurando así la formación de un árbol de expansión mínima.

Por último devolvemos el resultado.

Pseudocódigo Kruskal

```
Función kruskal(origen):
    resultado = Nuevo Resultado(origen)
    pq = Nuevo PriorityQueue<Arista>()
    vertices = Nuevo HashMap<String, String>()
    grupos = Nuevo HashMap<String, Integer>()

    Para cada entrada en grafo:
        desde = entrada.clave
        vertices.poner(desde, desde)
        grupos.poner(desde, 1)

        Para cada entrada2 en entrada.valor:
            hasta = entrada2.clave
            w = entrada2.valor
            Si desde.compareTo(hasta) > 0, entonces
                Continuar // Saltar esta iteración si la arista ya se ha procesado
            Fin Si
            pq.agregar(Nueva Arista(desde, hasta, w))
        Fin Para
    Fin Para

    Mientras tamaño(resultado) < tamaño(grafo) - 1:
        arista = pq.extraerMenor()
        raiz = buscarRaiz(vertices, arista.origen)
        raiz2 = buscarRaiz(vertices, arista.destino)
        Si raiz != raiz2, entonces
```

```
            resultado.agregarArista(arista)
            tam = grupos.obtener(raiz)
            tam2 = grupos.obtener(raiz2)
            Si tam < tam2, entonces
               vertices.poner(raiz, raiz2)
               grupos.poner(raiz2, tam + tam2)
            Sino,
               vertices.poner(raiz2, raiz)
               grupos.poner(raiz, tam + tam2)
            Fin Si
         Fin Si
      Fin Mientras

      Devolver resultado
Fin Función
```

Tabla 6: Pseudocódigo Kruskal.

3.1.4 Viajante de comercio con Caminos simples.

Esta implementación del problema del viajante de comercio (TSP) por fuerza bruta busca todos los caminos posibles entre el vértice de origen y todos los demás vértices del grafo.

El método simplePaths principal inicializa la búsqueda para cada destino posible desde el origen y almacena el resultado en una instancia de la clase Resultado.
El algoritmo se apoya en una función recursiva simplePaths que construye los caminos posibles utilizando backtracking. La función recibe una lista vértices que rastrea el camino actual, el objeto resultado que almacena el mejor camino encontrado hasta el momento, el vértice current que representa el vértice actual en el camino, el destino al que se quiere llegar, y el source que es el origen del camino completo.
En cada llamada, el vértice actual se añade a la lista de vértices. Si el vértice actual coincide con el destino y se han visitado todos los vértices (el camino tiene una longitud que es igual al número total de vértices), se calcula el peso total del camino cerrando el ciclo con una arista que regresa al origen. Si el peso total del nuevo camino es menor que el del mejor camino encontrado hasta el momento o si aún no se ha encontrado un camino válido (el total es cero), el nuevo camino se establece como el mejor.

Si el vértice actual no es el destino o no se han visitado todos los vértices, se llama recursivamente a simplePaths para cada vecino no visitado del vértice actual, evitando ciclos al no revisitar vértices ya incluidos en el camino actual.
Tras explorar todas las posibilidades desde el vértice actual, se elimina el último vértice de los vértices para retroceder en el proceso de backtracking y explorar otras ramas del árbol de búsqueda.

Finalmente, el método principal simplePaths devuelve el objeto Resultado que contiene el camino con el menor peso encontrado que visita todos los vértices y regresa al origen, ofreciendo una solución completa al TSP si existe un camino válido. Sin embargo, dada la naturaleza de fuerza bruta del algoritmo, su complejidad es exponencial y no es práctico para grafos grandes.

Pseudocódigo viajante de comercio con Caminos simples

```
Función simplePaths(origen):
   Si origen == null o no contiene clave origen en grafo, entonces
      Devolver null
   Fin Si

   resultado = Nuevo Resultado(origen)

   Para cada destino en obtenerClaves(grafo[origen]):
      simplePaths(Nueva LinkedList<String>(), resultado, origen, destino, origen)
   Fin Para

   Devolver resultado
Fin Función

Función simplePaths(camino, resultado, actual, destino, origen):
   camino.agregar(actual)

   Si actual == destino, entonces
      resultado.agregarCamino(camino)
   Sino,
      Para cada vecino en obtenerVecinos(actual):
         Si vecino != origen y !camino.contiene(vecino), entonces
            simplePaths(camino, resultado, vecino, destino, origen)
         Fin Si
      Fin Para
   Fin Si

   camino.eliminarUltimo()
Fin Función
```

Tabla 7: Pseudocódigo viajante de comercio con Caminos simples.

3.1.5 Problema del viajante (Greedy).

Esta implementación del problema del viajante de comercio (TSP) utiliza un enfoque greedy para encontrar una solución aproximada. El algoritmo comienza creando una instancia de la clase Resultado para almacenar la solución. El conjunto visitados se utiliza para mantener un registro de los vértices ya incluidos en la ruta.

El proceso inicia en el vértice de origen, que se añade a los visitados. En cada iteración del bucle, mientras que el número de vértices visitados sea menor que el número total de vértices en el grafo, el algoritmo busca el vértice más cercano al vértice actual que aún no ha sido visitado. Esta búsqueda se realiza mediante el método verticeMasCercano, que devuelve el vecino más cercano no visitado según el peso de las aristas.
Si se encuentra un vértice siguiente (siguiente), el algoritmo añade una arista desde el vértice actual hasta el siguiente en la solución, con el peso correspondiente obtenido por getWeight. El vértice siguiente se marca como visitado y se convierte en el nuevo vértice

actual para la siguiente iteración. Este proceso se repite hasta que se han visitado todos los vértices.
Después de visitar todos los vértices, el algoritmo intenta cerrar el circuito volviendo al vértice de origen desde el último vértice actual. Si el peso de esta arista de retorno es inválido (nulo o negativo), se lanza una excepción, indicando que no es posible regresar al origen, lo cual sería una señal de que no todos los vértices están conectados directamente con el origen o hay un error en los datos del grafo.
Finalmente, el algoritmo devuelve la solución que contiene el circuito generado, aunque no garantiza que sea el circuito más corto posible debido a la naturaleza greedy del algoritmo.

Pseudocódigo viajante de comercio con TSP

```
Función greedyTSP(origen):
    solucion = Nuevo Resultado(origen)
    visitados = Nuevo HashSet<String>()
    actual = origen
    visitados.agregar(actual)

    Mientras tamaño(visitados) < numeroVertices():
        siguiente = verticeMasCercano(visitados, actual)
        Si siguiente == null, entonces
            Detener
        Fin Si
        peso = obtenerPeso(actual, siguiente)
        solucion.agregarArista(actual, siguiente, peso)
        visitados.agregar(siguiente)
        actual = siguiente
    Fin Mientras

    peso = obtenerPeso(actual, origen)
    Si peso == null o peso < 0, entonces
        LanzarExcepcion("No es posible regresar al origen desde " + actual)
    Fin Si
    solucion.agregarArista(actual, origen, peso)

    Devolver solucion
Fin Función
```

Tabla 8: Pseudocódigo viajante de comercio con TSP.

3.1.6 Generador de redes aleatorias.

Pseudocódigo generador de redes aleatorias

```
Función generarRedSintetica(numNodos, numAristas, densidad):
    vertices = ListaVacia()
    Para i desde 0 hasta numNodos - 1:
        Agregar(vertices, ConvertirAString('A' + i))
    Fin Para

    todos = ListaVacia()
```

```
    Para i desde 0 hasta tamaño(vertices) - 2:
        Para j desde i + 1 hasta tamaño(vertices) - 1:
            Agregar(todos, vertices[i] + "-" + vertices[j])
        Fin Para
    Fin Para

    u = ListaVacia()
    resultado = ListaVacia()
    Aleatorizar(vertices)
    Agregar(u, Remover(vertices, 0))
    Mientras NoEstéVacia(vertices):
        desde = Remover(vertices, 0)
        hacia = u[EnteroAleatorioEntre(0, tamaño(u) - 1)]
        Agregar(resultado, desde + " " + hacia + " " + Redondear(Aleatorio(0, 100) + 1))
        Si desde < hacia, entonces
            Remover(todos, desde + "-" + hacia)
        Sino,
            Remover(todos, hacia + "-" + desde)
        Fin Si
        Agregar(u, desde)
    Fin Mientras

    minAristas = numNodos - 1
    Mientras tamaño(resultado) < minAristas y tamaño(resultado) < numAristas:
        x = RemoverAleatorio(todos)
        tokens = Separar(x, "-")
        Agregar(resultado, tokens[0] + " " + tokens[1] + " " + Redondear(Aleatorio(0, 100) +
1))
    Fin Mientras

    nAristasAAgregar = numAristas - tamaño(resultado)
    Para i desde 0 hasta nAristasAAgregar - 1 y NoEstéVacia(todos):
        x = Remover(todos, 0)
        tokens = Separar(x, "-")
        Agregar(resultado, tokens[0] + " " + tokens[1] + " " + Redondear(Aleatorio(0, 100) +
1))
    Fin Para

    rutaArchivo = DirectorioActual() + "/src/main/java/org/eda2/practica02/grafoSintetico.txt"

    Intentar:
        escritor = NuevoEscritor(rutaArchivo)
        Escribir(escritor, "0\n")
        Escribir(escritor, numNodos + "\n")
        Para cada vertice en vertices:
            Escribir(escritor, vertice + "\n")
        Fin Para
        Escribir(escritor, "\n")
        Escribir(escritor, numAristas + "\n")
        Para cada arista en resultado:
            Escribir(escritor, arista + "\n")
        Fin Para
        Cerrar(escritor)
        Mostrar("Grafo generado y guardado en " + rutaArchivo)
```

```
    Atrapar Error e:
        Mostrar(e)
    Fin Intentar

    Devolver resultado
Fin Función
```

Tabla 9: Pseudocódigo generador de redes aleatorias.

3.2 Estudio teórico.

Tanto Prim como Kruskal en el peor caso se obtiene un O(n^2)

- Si el grafo es muy denso (m → n * (n - 1) / 2), entonces el algoritmo de Kruskal requiere O((n^2)log n)), por lo que el algoritmo de Prim puede ser mejor.

- Si el grafo es disperso (m → n), entonces el algoritmo de Kruskal requiere O(n log n), por lo que el algoritmo de Prim puede ser menos eficiente.

En el problema del viajante obtenemos:

- Heurística greedy ⇒ Selección de nodos.
- El tiempo de ejecución es O(n^2), donde n es el número de nodos.
- Se trata de una heurística, no de un algoritmo aproximado, ya que no es posible acotar el error máximo.

3.2.1 Prim sin cola de prioridad.

El análisis de la complejidad temporal en este caso es el siguiente:

- La creación de las estructuras de datos HashMap y HashSet tiene un coste de O(N), donde N es el número de vértices en el grafo.

- El bucle que inicializa las claves tiene un costo de O(N).

- El bucle principal se ejecuta N veces, y dentro de este bucle, la función claveMenor se ejecuta con un coste de O(N) en el peor de los casos, ya que debe recorrer todos los vértices para encontrar la clave más pequeña. Por lo tanto, el costo total de este bucle es O(N^2).

- Dentro del bucle principal, también hay un bucle que recorre todos los vecinos de un vértice. En el peor de los casos, esto puede ser todos los vértices, por lo que este bucle tiene un coste de O(N) por cada iteración del bucle principal, lo que resulta en un coste total de O(N^2).

- Finalmente, el bucle que agrega las aristas a la solución tiene un coste de O(N), ya que se ejecuta una vez por cada vértice.

Por lo tanto, la complejidad temporal total del algoritmo es O(N^2) lo cual es típico para la implementación del algoritmo de Prim sin una cola de prioridad.

3.2.2 Prim con cola de prioridad.

Esta versión utiliza una cola de prioridad (implementada con un TreeMap) para mejorar la eficiencia del algoritmo.

- La creación de las estructuras de datos HashMap, HashSet y TreeMap tiene un coste de O(N).

- El bucle principal se ejecuta N veces. Dentro de este bucle, la función menorClavePQ se ejecuta dando O(log N) en el peor de los casos, ya que debe extraer el mínimo de un TreeMap. Por lo tanto, el coste total de este bucle es O(N log N).

- Dentro del bucle principal, también hay un bucle que recorre todos los vecinos de un vértice. En el peor de los casos, esto puede ser todos los vértices, por lo que este bucle tiene un coste de O(N) por cada iteración del bucle principal, lo que resulta en un coste total de O(N^2). Sin embargo, este pasa a O(N log N) debido a la actualización de la cola de prioridad siendo una mejora significativa respecto al anterior.

El bucle que agrega las aristas a la solución tiene un coste de O(N), ya que se ejecuta una vez por cada vértice, luego el coste total del algoritmo es O(N log N), que es más eficiente que la versión no optimizada del algoritmo de Prim que tiene un coste de O(N^2) gracias al uso de una cola de prioridad para seleccionar el vértice más cercano no visitado en cada iteración.

3.2.3 Kruskal.

Para la implementación que se ha realizado de Kruskal obtenemos:

- Las estructuras de datos HashMap, HashSet y PriorityQueue tiene un coste de O(N).

- El bucle que inicializa las claves y llena la cola de prioridad tiene un coste de O(M log M), donde M es el número de aristas en el grafo, ya que cada inserción en la cola de prioridad tiene un coste de O(log M).

- El bucle principal se ejecuta N veces. Dentro de este bucle, se extrae la arista de menor peso de la cola de prioridad, lo que tiene un coste de O(log N). Luego, se realiza la operación buscarRaiz, que tiene un coste de O(log N) en el peor de los casos, así que el coste total de este bucle es O((N+N) log N).
- Por último, el bucle que agrega las aristas a la solución tiene un coste de O(N), ya que se ejecuta una vez por cada vértice.

El coste resultante del algoritmo es O((N+N) log N), que es más eficiente que la versión no optimizada del algoritmo de Prim que tiene un coste de O(N^2). Esta mejora en la eficiencia se debe al uso de una cola de prioridad para seleccionar la arista de menor peso en cada iteración y a la estructura de conjuntos disjuntos para mantener el seguimiento de los componentes conectados.

3.2.4 Viajante de comercio con Caminos simples.

Esta función greedy da un resultado que tiene un gran impacto temporal en la ejecución del programa.

- La comprobación inicial de si el origen es null o si no está en el grafo tiene un coste de O(1).

- El bucle que recorre todos los vecinos del vértice de origen tiene un coste de O(N), donde N es el número de vértices en el grafo, ya que en el peor de los casos cada vértice podría ser un vecino del origen.

- Dentro de este bucle, se llama a la función simplePaths, la cual es una función recursiva que explora todos los caminos posibles en el grafo. En el peor de los casos, esto podría implicar visitar cada vértice una vez por cada posible camino en el grafo. Dado que el número de caminos en un grafo puede ser exponencial en el número de vértices (específicamente, 2^N), el coste de esta función recursiva es O(N * 2^N) en el peor de los casos.

Por lo tanto, el coste total de la función es O(N * 2^N). Esto es bastante costoso, y es típico de los algoritmos que deben explorar todos los caminos posibles en un grafo. Sin embargo, este coste es inevitable a la hora de encontrar todos los caminos simples.

3.2.5 Viajante de comercio con TSP.

En el algoritmo del Viajante de Comercio (TSP) utilizando un enfoque voraz obtenemos lo siguiente:

- La creación de las estructuras de datos Resultado y HashSet tiene un coste de O(N), como en los anteriores.

- El bucle principal se ejecuta N veces, ya que en cada iteración se visita un vértice nuevo. Dentro de este bucle, se llama a la función verticeMasCercano, que tiene un coste de O(N), ya que recorre todos los vértices para encontrar el más cercano, luego el coste total de este bucle es O(N^2).

- Para finalizar añade una arista desde el último vértice visitado al vértice de origen, lo que tiene un coste de O(1).

Por lo tanto, el coste total del algoritmo es O(N^2) típico para una implementación voraz del problema del Viajante de Comercio, ya que en cada paso se selecciona el vértice más cercano no visitado.

3.3 Estudio experimental.

El programa comienza solicitando por pantalla seleccionar una de las opciones disponibles, tras ello se ejecuta la función correspondiente y se selecciona un .txt sobre el que operar.

Para las grandes redes la imagen es tomada del inicio de la red y del final.

3.3.1 Prim sin cola de prioridad.

3.3.1.1 graphEDAland.txt.

```
   INSTALACIÓN DE FIBRA OPTICA
---------------------------------
1. Prim sin cola de prioridad.
2. Prim con cola de prioridad.
3. Kruskal.
4. Viajante de comercio con Caminos simples.
5. Generador de redes sinteticas.
6. Viajante de comercio con TSP.
7. Finalizar programa
---------------------------------

Seleccione una opcion: 1
Opción seleccionada: 1. Prim sin cola de prioridad.

Redes disponibles:
1. graphEDAland.txt.
2. graphEDAlandLarge.txt.

Seleccione una red: 1

Resultado:
Origen = Almeria
Set = [
Arista [Origen = Bilbao, Destino = Oviedo, Peso = 304.0],
Arista [Origen = Murcia, Destino = Albacete, Peso = 150.0],
Arista [Origen = Valladolid, Destino = Bilbao, Peso = 280.0],
Arista [Origen = Barcelona, Destino = Gerona, Peso = 100.0],
Arista [Origen = Badajoz, Destino = Caceres, Peso = 90.0],
Arista [Origen = Albacete, Destino = Valencia, Peso = 191.0],
Arista [Origen = Madrid, Destino = Valladolid, Peso = 193.0],
Arista [Origen = Zaragoza, Destino = Barcelona, Peso = 296.0],
Arista [Origen = Almeria, Destino = Granada, Peso = 173.0],
Arista [Origen = Bilbao, Destino = Zaragoza, Peso = 324.0],
Arista [Origen = Valladolid, Destino = Vigo, Peso = 356.0],
Arista [Origen = Albacete, Destino = Madrid, Peso = 251.0],
Arista [Origen = Huelva, Destino = Badajoz, Peso = 234.0],
Arista [Origen = Sevilla, Destino = Huelva, Peso = 92.0],
Arista [Origen = Gerona, Destino = Lerida, Peso = 222.0],
Arista [Origen = Almeria, Destino = Murcia, Peso = 224.0],
Arista [Origen = Granada, Destino = Jaen, Peso = 99.0],
Arista [Origen = Vigo, Destino = Corunya, Peso = 171.0],
Arista [Origen = Sevilla, Destino = Cadiz, Peso = 125.0],
Arista [Origen = Jaen, Destino = Sevilla, Peso = 242.0]]
Peso Total = 4117.0

Tiempo promedio de ejecución: 264190 nanosegundos
```

Imagen 2: Salida por pantalla Prim sin cola de prioridad graphEDAland.txt.

3.3.1.2 graphEDAlandLarge.txt.

```
Seleccione una opcion: 1
Opción seleccionada: 1. Prim sin cola de prioridad.

Redes disponibles:
1. graphEDAland.txt.
2. graphEDAlandLarge.txt.

Seleccione una red: 2

Resultado:
Origen = 1
Set = [
Arista [Origen = 118, Destino = 134, Peso = 23.0],
Arista [Origen = 32, Destino = 33, Peso = 16.0],
Arista [Origen = 847, Destino = 1004, Peso = 26.0],
Arista [Origen = 1041, Destino = 658, Peso = 18.0],
Arista [Origen = 194, Destino = 204, Peso = 13.0],
Arista [Origen = 391, Destino = 390, Peso = 15.0],
Arista [Origen = 68, Destino = 67, Peso = 19.0],
Arista [Origen = 181, Destino = 167, Peso = 15.0],
Arista [Origen = 308, Destino = 275, Peso = 25.0]]
Peso Total = 17315.0

Tiempo promedio de ejecución: 21299940 nanosegundos
```

Imagen 3: Salida por pantalla Prim sin cola de prioridad graphEDAlandLarge.txt.

3.3.2 Prim con cola de prioridad.

3.3.2.1 graphEDAland.txt.

```
Seleccione una opcion: 2
Opción seleccionada: 2. Prim con cola de prioridad.

Redes disponibles:
1. graphEDAland.txt.
2. graphEDAlandLarge.txt.
3. grafoSintetico.txt.

Seleccione una red: 1

Resultado:
Origen = Almeria
Set = [
Arista [Origen = Bilbao, Destino = Oviedo, Peso = 304.0],
Arista [Origen = Murcia, Destino = Albacete, Peso = 150.0],
Arista [Origen = Valladolid, Destino = Bilbao, Peso = 280.0],
Arista [Origen = Barcelona, Destino = Gerona, Peso = 100.0],
Arista [Origen = Badajoz, Destino = Caceres, Peso = 90.0],
Arista [Origen = Albacete, Destino = Valencia, Peso = 191.0],
Arista [Origen = Madrid, Destino = Valladolid, Peso = 193.0],
Arista [Origen = Zaragoza, Destino = Barcelona, Peso = 296.0],
Arista [Origen = Almeria, Destino = Granada, Peso = 173.0],
Arista [Origen = Bilbao, Destino = Zaragoza, Peso = 324.0],
Arista [Origen = Valladolid, Destino = Vigo, Peso = 356.0],
Arista [Origen = Albacete, Destino = Madrid, Peso = 251.0],
Arista [Origen = Huelva, Destino = Badajoz, Peso = 234.0],
Arista [Origen = Sevilla, Destino = Huelva, Peso = 92.0],
Arista [Origen = Gerona, Destino = Lerida, Peso = 222.0],
Arista [Origen = Almeria, Destino = Murcia, Peso = 224.0],
Arista [Origen = Granada, Destino = Jaen, Peso = 99.0],
Arista [Origen = Vigo, Destino = Corunya, Peso = 171.0],
Arista [Origen = Sevilla, Destino = Cadiz, Peso = 125.0],
Arista [Origen = Jaen, Destino = Sevilla, Peso = 242.0]]
Peso Total = 4117.0

Tiempo promedio de ejecución: 207090 nanosegundos
```

Imagen 4: Salida por pantalla Prim con cola de prioridad graphEDAland.txt.

3.3.2.2 graphEDAlandLarge.txt.

```
Seleccione una opcion: 2
Opción seleccionada: 2. Prim con cola de prioridad.

Redes disponibles:
1. graphEDAland.txt.
2. graphEDAlandLarge.txt.
3. grafoSintetico.txt.

Seleccione una red: 2

Resultado:
Origen = 1
Set = [
Arista [Origen = 118, Destino = 134, Peso = 23.0],
Arista [Origen = 32, Destino = 33, Peso = 16.0],
Arista [Origen = 308, Destino = 275, Peso = 25.0]]
Peso Total = 17315.0

Tiempo promedio de ejecución: 4722650 nanosegundos
```

Imagen 5: Salida por pantalla Prim con cola de prioridad graphEDAland.txt.

3.3.3 Kruskal.

3.3.3.1 graphEDAland.txt.

```
Seleccione una opcion: 3
Opción seleccionada: 3. Kruskal.

Redes disponibles:
1. graphEDAland.txt.
2. graphEDAlandLarge.txt.
3. grafoSintetico.txt.

Seleccione una red: 1

Resultado:
Origen = Almeria
Set = [
Arista [Origen = Barcelona, Destino = Zaragoza, Peso = 296.0],
Arista [Origen = Bilbao, Destino = Oviedo, Peso = 304.0],
Arista [Origen = Bilbao, Destino = Valladolid, Peso = 280.0],
Arista [Origen = Huelva, Destino = Sevilla, Peso = 92.0],
Arista [Origen = Badajoz, Destino = Caceres, Peso = 90.0],
Arista [Origen = Barcelona, Destino = Gerona, Peso = 100.0],
Arista [Origen = Albacete, Destino = Valencia, Peso = 191.0],
Arista [Origen = Madrid, Destino = Valladolid, Peso = 193.0],
Arista [Origen = Almeria, Destino = Granada, Peso = 173.0],
Arista [Origen = Bilbao, Destino = Zaragoza, Peso = 324.0],
Arista [Origen = Badajoz, Destino = Huelva, Peso = 234.0],
Arista [Origen = Valladolid, Destino = Vigo, Peso = 356.0],
Arista [Origen = Albacete, Destino = Madrid, Peso = 251.0],
Arista [Origen = Cadiz, Destino = Sevilla, Peso = 125.0],
Arista [Origen = Gerona, Destino = Lerida, Peso = 222.0],
Arista [Origen = Almeria, Destino = Murcia, Peso = 224.0],
Arista [Origen = Granada, Destino = Jaen, Peso = 99.0],
Arista [Origen = Albacete, Destino = Murcia, Peso = 150.0],
Arista [Origen = Corunya, Destino = Vigo, Peso = 171.0],
Arista [Origen = Jaen, Destino = Sevilla, Peso = 242.0]]
Peso Total = 4117.0

Tiempo promedio de ejecución: 194030 nanosegundos
```

Imagen 7: Salida por pantalla Kruskal graphEDAland.txt.

3.3.3.2 graphEDAlandLarge.txt.

```
Seleccione una opcion: 3
Opción seleccionada: 3. Kruskal.

Redes disponibles:
1. graphEDAland.txt.
2. graphEDAlandLarge.txt.
3. grafoSintetico.txt.

Seleccione una red: 2

Resultado:
Origen = 1
Set = [
Arista [Origen = 63, Destino = 65, Peso = 13.0],
Arista [Origen = 76, Destino = 77, Peso = 13.0],
Arista [Origen = 308, Destino = 309, Peso = 17.0],
Arista [Origen = 595, Destino = 642, Peso = 14.0],
Arista [Origen = 836, Destino = 838, Peso = 15.0],
Arista [Origen = 206, Destino = 208, Peso = 16.0]]
Peso Total = 17315.0

Tiempo promedio de ejecución: 3686750 nanosegundos
```

Imagen 8: Salida por pantalla Kruskal graphEDAlandLarge.txt.

3.3.4 Viajante de comercio con Caminos simples.

3.3.4.1 graphEDAlandNewroads.txt.

```
   INSTALACIÓN DE FIBRA OPTICA
---------------------------------
1. Prim sin cola de prioridad.
2. Prim con cola de prioridad.
3. Kruskal.
4. Viajante de comercio con Caminos simples.
5. Generador de redes sinteticas.
6. Viajante de comercio con TSP.
7. Finalizar programa
---------------------------------

Seleccione una opcion: 4
Opción seleccionada: 4. Viajante de comercio con Caminos simples.

Resultado:
Origen = Almeria
Set = [
Arista [Origen = Barcelona, Destino = Zaragoza, Peso = 296.0],
Arista [Origen = Cadiz, Destino = Huelva, Peso = 101.0],
Arista [Origen = Zaragoza, Destino = Valencia, Peso = 308.0],
Arista [Origen = Badajoz, Destino = Caceres, Peso = 90.0],
Arista [Origen = Corunya, Destino = Oviedo, Peso = 386.0],
Arista [Origen = Murcia, Destino = Almeria, Peso = 224.0],
Arista [Origen = Madrid, Destino = Valladolid, Peso = 193.0],
Arista [Origen = Gerona, Destino = Barcelona, Peso = 100.0],
Arista [Origen = Almeria, Destino = Granada, Peso = 173.0],
Arista [Origen = Oviedo, Destino = Bilbao, Peso = 304.0],
Arista [Origen = Valladolid, Destino = Vigo, Peso = 356.0],
Arista [Origen = Bilbao, Destino = Lerida, Peso = 449.0],
Arista [Origen = Valencia, Destino = Albacete, Peso = 191.0],
Arista [Origen = Huelva, Destino = Badajoz, Peso = 234.0],
Arista [Origen = Lerida, Destino = Gerona, Peso = 222.0],
Arista [Origen = Granada, Destino = Jaen, Peso = 99.0],
Arista [Origen = Caceres, Destino = Madrid, Peso = 297.0],
Arista [Origen = Vigo, Destino = Corunya, Peso = 171.0],
Arista [Origen = Jaen, Destino = Sevilla, Peso = 242.0],
Arista [Origen = Sevilla, Destino = Cadiz, Peso = 125.0],
Arista [Origen = Albacete, Destino = Murcia, Peso = 150.0]]
Peso Total = 4711.0

Tiempo promedio de ejecución: 17537710 nanosegundos
```

Imagen 9: Salida por pantalla Viajante de comercio con Caminos simples graphEDAlandNewroads.txt.

3.3.5 Generador de redes sintéticas.

Para aplicar este método se ha de indicar el número de vértices y de aristas, a continuación el resultado se guarda en un .txt que podrá usarse en Prim y Kruskal.

```
   INSTALACIÓN DE FIBRA OPTICA
---------------------------------
1. Prim sin cola de prioridad.
2. Prim con cola de prioridad.
3. Kruskal.
4. Viajante de comercio con Caminos simples.
5. Generador de redes sinteticas.
6. Viajante de comercio con TSP.
7. Finalizar programa
---------------------------------

Seleccione una opcion: 5
Opción seleccionada: 5. Generador de redes sintéticas.

Ingrese el número de nodos para la red sintética (mínimo 2): 20
Ingrese el número de aristas para la red sintética: 5
ERROR Introduzca una opcion entre 19 y 190
Ingrese el número de aristas para la red sintética: 20

Grafo generado y guardado en E:\Universidad\EDA2\equipo-ualeda2-2024-simon-antonio\src\main\java\org\eda2\practica02\grafoSintetico.txt
[C E 57, N E 84, H N 8, J C 39, Q H 71, I C 100, M I 11, G M 28, D E 89, R I 51, P H 51, F C 59, S E 59, O R 67, T P 16, K G 60, B D 97, L C 62, A D 80, A B 83]
```

Imagen 10: Generador de redes aleatorias.

3.3.5.1 Prim con cola de prioridad.

```
Seleccione una opcion: 2
Opción seleccionada: 2. Prim con cola de prioridad.

Redes disponibles:
1. graphEDAland.txt.
2. graphEDAlandLarge.txt.
3. grafoSintetico.txt.

Seleccione una red: 3

Resultado:
Origen = A
Set = [
Arista [Origen = E, Destino = S, Peso = 59.0],
Arista [Origen = C, Destino = J, Peso = 39.0],
Arista [Origen = I, Destino = R, Peso = 51.0],
Arista [Origen = M, Destino = G, Peso = 28.0],
Arista [Origen = I, Destino = M, Peso = 11.0],
Arista [Origen = G, Destino = K, Peso = 60.0],
Arista [Origen = H, Destino = P, Peso = 51.0],
Arista [Origen = C, Destino = F, Peso = 59.0],
Arista [Origen = R, Destino = O, Peso = 67.0],
Arista [Origen = E, Destino = C, Peso = 57.0],
Arista [Origen = P, Destino = T, Peso = 16.0],
Arista [Origen = A, Destino = B, Peso = 83.0],
Arista [Origen = A, Destino = D, Peso = 80.0],
Arista [Origen = D, Destino = E, Peso = 89.0],
Arista [Origen = N, Destino = H, Peso = 8.0],
Arista [Origen = C, Destino = L, Peso = 62.0],
Arista [Origen = H, Destino = Q, Peso = 71.0],
Arista [Origen = E, Destino = N, Peso = 84.0],
Arista [Origen = C, Destino = I, Peso = 100.0]]
Peso Total = 1075.0

Tiempo promedio de ejecución: 304540 nanosegundos
```

Imagen 11: Salida por pantalla Generador de redes aleatorias Prim con cola de prioridad.

3.3.5.2 Kruskal con cola de prioridad.

```
Seleccione una red: 3

Resultado:
Origen = A
Set = [
Arista [Origen = E, Destino = S, Peso = 59.0],
Arista [Origen = C, Destino = J, Peso = 39.0],
Arista [Origen = I, Destino = R, Peso = 51.0],
Arista [Origen = H, Destino = N, Peso = 8.0],
Arista [Origen = I, Destino = M, Peso = 11.0],
Arista [Origen = H, Destino = P, Peso = 51.0],
Arista [Origen = C, Destino = E, Peso = 57.0],
Arista [Origen = G, Destino = K, Peso = 60.0],
Arista [Origen = C, Destino = F, Peso = 59.0],
Arista [Origen = O, Destino = R, Peso = 67.0],
Arista [Origen = P, Destino = T, Peso = 16.0],
Arista [Origen = A, Destino = B, Peso = 83.0],
Arista [Origen = A, Destino = D, Peso = 80.0],
Arista [Origen = D, Destino = E, Peso = 89.0],
Arista [Origen = G, Destino = M, Peso = 28.0],
Arista [Origen = C, Destino = L, Peso = 62.0],
Arista [Origen = H, Destino = Q, Peso = 71.0],
Arista [Origen = E, Destino = N, Peso = 84.0],
Arista [Origen = C, Destino = I, Peso = 100.0]]
Peso Total = 1075.0

Tiempo promedio de ejecución: 125930 nanosegundos
```

Imagen 12: Salida por pantalla Generador de redes aleatorias Kruskal con cola de prioridad.

3.3.6 Viajante de comercio con TSP.

3.3.6.1 Viajante de comercio con TSP entrada 10.

```
Seleccione una opcion: 5
Opción seleccionada: 5. Generador de redes sintéticas.

Ingrese el número de nodos para la red sintética (mínimo 2): 10
Ingrese el número de aristas para la red sintética: 1
ERROR Introduzca una opcion entre 9 y 45
Ingrese el número de aristas para la red sintética: 45

Grafo generado y guardado en E:\Universidad\EDA2\equipo-ualeda2-20
[A H 89, B A 97, E H 100, F E 39, I B 2, G A 56, D H 53, J F 92, C

   INSTALACIÓN DE FIBRA OPTICA
---------------------------------
1. Prim sin cola de prioridad.
2. Prim con cola de prioridad.
3. Kruskal.
4. Viajante de comercio con Caminos simples.
5. Generador de redes sinteticas.
6. Viajante de comercio con TSP.
7. Finalizar programa
---------------------------------

Seleccione una opcion: 6
Opción seleccionada: 6. Viajante de comercio con Greedy TSP.

Resultado:
Origen = A
Set = [
Arista [Origen = G, Destino = D, Peso = 14.0],
Arista [Origen = C, Destino = G, Peso = 4.0],
Arista [Origen = A, Destino = I, Peso = 18.0],
Arista [Origen = E, Destino = F, Peso = 39.0],
Arista [Origen = B, Destino = J, Peso = 16.0],
Arista [Origen = I, Destino = B, Peso = 2.0],
Arista [Origen = J, Destino = C, Peso = 8.0],
Arista [Origen = D, Destino = E, Peso = 12.0],
Arista [Origen = F, Destino = H, Peso = 15.0],
Arista [Origen = H, Destino = A, Peso = 89.0]]
Peso Total = 217.0

Tiempo promedio de ejecución: 144010 nanosegundos
```

Imagen 13: Salida por pantalla Viajante de comercio con TSP entrada 10.

3.3.6.2 Viajante de comercio con TSP entrada 20.

```
Seleccione una opcion: 5
Opción seleccionada: 5. Generador de redes sintéticas.

Ingrese el número de nodos para la red sintética (mínimo 2): 20
Ingrese el número de aristas para la red sintética: 1
ERROR Introduzca una opcion entre 19 y 190
Ingrese el número de aristas para la red sintética: 190

Grafo generado y guardado en E:\Universidad\EDA2\equipo-ualeda2-2024-sim
[T Q 25, C Q 34, I T 94, R Q 86, K R 18, O T 10, H O 96, N C 50, J I 76,

   INSTALACIÓN DE FIBRA OPTICA
---------------------------------
1. Prim sin cola de prioridad.
2. Prim con cola de prioridad.
3. Kruskal.
4. Viajante de comercio con Caminos simples.
5. Generador de redes sinteticas.
6. Viajante de comercio con TSP.
7. Finalizar programa
---------------------------------

Seleccione una opcion: 6
Opción seleccionada: 6. Viajante de comercio con Greedy TSP.

Resultado:
Origen = A
Set = [
Arista [Origen = R, Destino = A, Peso = 58.0],
Arista [Origen = F, Destino = J, Peso = 2.0],
Arista [Origen = J, Destino = B, Peso = 20.0],
Arista [Origen = O, Destino = F, Peso = 18.0],
Arista [Origen = T, Destino = D, Peso = 9.0],
Arista [Origen = Q, Destino = I, Peso = 26.0],
Arista [Origen = L, Destino = T, Peso = 3.0],
Arista [Origen = B, Destino = R, Peso = 2.0],
Arista [Origen = A, Destino = P, Peso = 8.0],
Arista [Origen = M, Destino = L, Peso = 4.0],
Arista [Origen = D, Destino = C, Peso = 4.0],
Arista [Origen = N, Destino = K, Peso = 18.0],
Arista [Origen = I, Destino = S, Peso = 6.0],
Arista [Origen = P, Destino = M, Peso = 14.0],
Arista [Origen = C, Destino = G, Peso = 4.0],
Arista [Origen = K, Destino = O, Peso = 1.0],
Arista [Origen = H, Destino = Q, Peso = 4.0],
Arista [Origen = S, Destino = N, Peso = 32.0],
Arista [Origen = G, Destino = E, Peso = 8.0],
Arista [Origen = E, Destino = H, Peso = 7.0]]
Peso Total = 248.0

Tiempo promedio de ejecución: 265250 nanosegundos
```

Imagen 14: Salida por pantalla Viajante de comercio con TSP entrada 20.

3.3.6.3 Viajante de comercio con TSP entrada 50.

```
Seleccione una opcion: 5
Opción seleccionada: 5. Generador de redes sintéticas.

Ingrese el número de nodos para la red sintética (mínimo 2): 50
Ingrese el número de aristas para la red sintética: 1
ERROR Introduzca una opcion entre 49 y 1225
Ingrese el número de aristas para la red sintética: 1225

Grafo generado y guardado en E:\Universidad\EDA2\equipo-ualeda2
[J c 65, K J 36, U K 4, Q U 55, l U 23, h Q 93, H h 65, C K 34,

   INSTALACIÓN DE FIBRA OPTICA
---------------------------------
1. Prim sin cola de prioridad.
2. Prim con cola de prioridad.
3. Kruskal.
4. Viajante de comercio con Caminos simples.
5. Generador de redes sinteticas.
6. Viajante de comercio con TSP.
7. Finalizar programa
---------------------------------

Seleccione una opcion: 6
Opción seleccionada: 6. Viajante de comercio con Greedy TSP.

Resultado:
Origen = A
Set = [
Arista [Origen = R, Destino = J, Peso = 5.0],
Arista [Origen = r, Destino = W, Peso = 23.0],
Arista [Origen = \, Destino = L, Peso = 16.0]]
Peso Total = 451.0

Tiempo promedio de ejecución: 1145920 nanosegundos
```

Imagen 15: Salida por pantalla Viajante de comercio con TSP entrada 50.

4.- Anexo 1.

4.1 Diseño del código.

En este diagrama ilustrativo del procedimiento que sigue en línea general el programa se puede ver cómo interactuar con él para realizar las diferentes funciones.

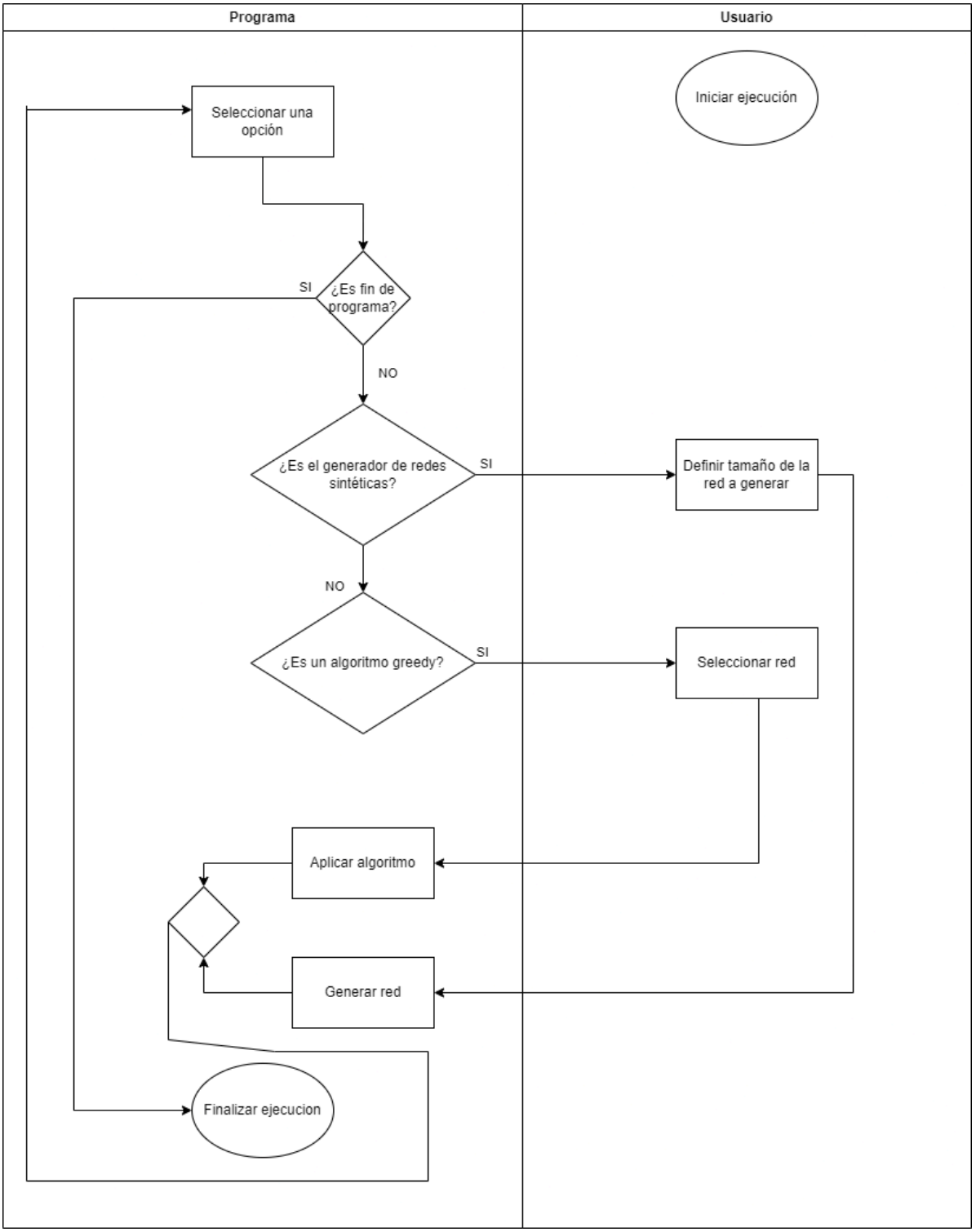

Imagen 16: Diagrama funcionamiento instalacionFibraOptica.

4.2 Diagramas de clase.

Buscando hacer un código modular y reutilizable en esta práctica se ha decidido crear masa clases con diferentes funciones en lugar de mantenerlo todo en una única clase, para además así facilitar la legibilidad del código.

El método Main es InstalacionFibraOptica, donde se inicia el programa y se llama a los demas metodos.

4.2.1 Clase InstalacionFibraOptica.

Atributos: Utiliza un Scanner estático para la entrada de usuario.

Métodos mencionables:
leerOpcion(): Lee una opción del usuario dentro de un rango especificado y maneja errores de entrada.
ejecutar(): Ejecuta diferentes funciones de algoritmos de red basadas en la opción elegida por el usuario.
main(): Controla el flujo del programa, mostrando un menú y procesando la selección del usuario.

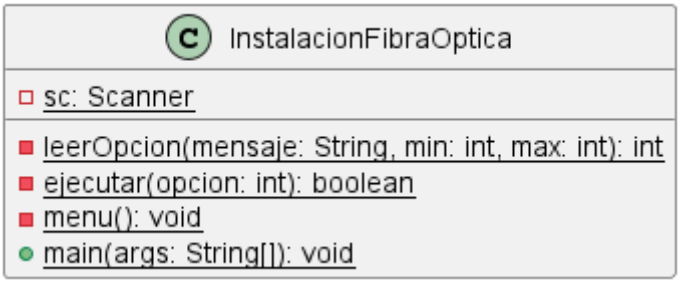

Imagen 17: Diagrama clase InstalacionFibraOptica.

4.2.2 Clase Arista.

Atributos: Almacena origen, destino y peso de una arista.

Métodos mencionables:
compareTo(): Compara dos aristas con el peso como atributo principal a comparar.

Interfaces implementadas:
Interfaz Comparable.

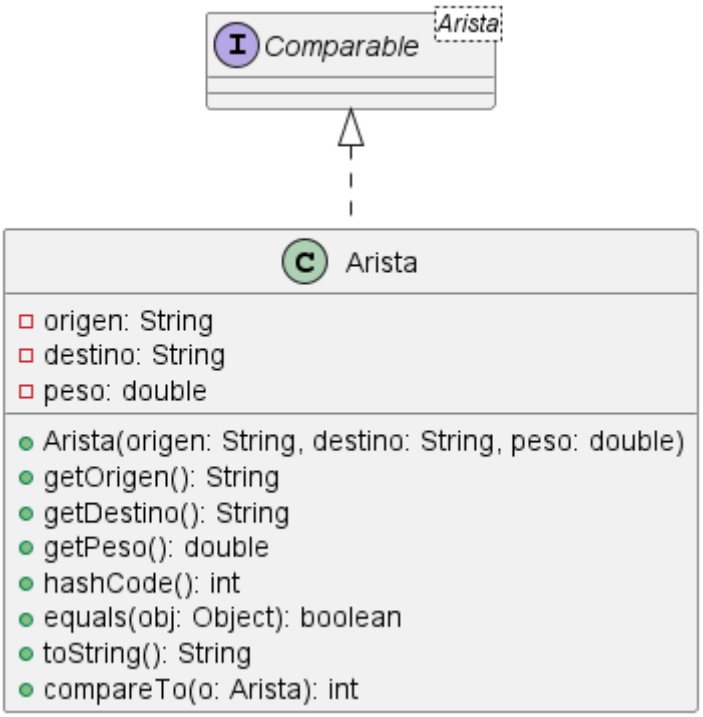

Imagen 18: Diagrama clase Arista.

4.2.3 Clase Generador.

Métodos mencionables:
generarRedSintetica(): Genera una red sintética basada en un número dado de nodos, aristas y densidad. Calcula y guarda la red en un archivo de texto.

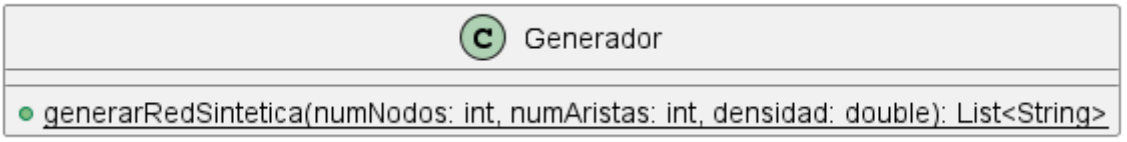

Imagen 19: Diagrama clase Generador.

4.2.4 Clase Network.

Atributos: Contiene un grafo representado como un mapa de vértices a otros mapas que vinculan vértices adyacentes y sus pesos. Además, maneja la dirección del grafo.

Métodos mencionables:
adVertise(): Añade un vértice al grafo si aún no existe.
addArista(): Añade una arista al grafo y, si el grafo no es dirigido, añade también la arista inversa.
getWeight(): Devuelve el peso de la arista entre dos vértices.
simplePaths(): Encuentra todos los caminos simples desde un origen hasta un destino usando backtracking.
primV2(): Implementa el algoritmo de Prim para encontrar el árbol de expansión mínimo.
kruskal(): Implementa el algoritmo de Kruskal para encontrar el árbol de expansión mínimo.

greedyTSP(): Resuelve el problema del viajante de comercio utilizando un enfoque voraz para seleccionar el siguiente vértice más cercano no visitado.

Interfaces implementadas:
Interfaz Iterable.

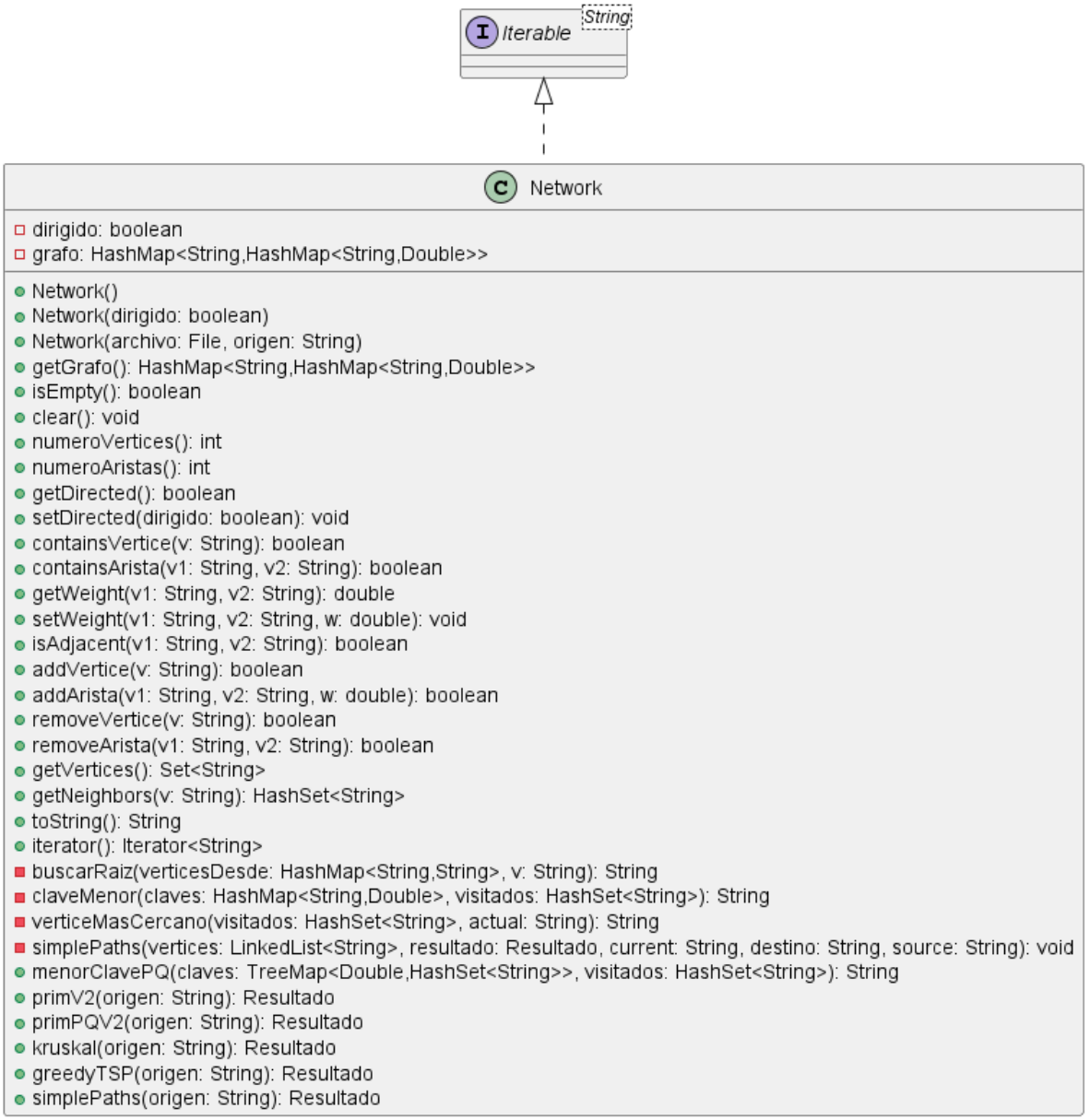

Imagen 20: Diagrama clase Network.

4.2.5 Clase Resultado.

Atributos: Almacena el origen del conjunto de resultados, un conjunto de aristas y el peso total de estas aristas.

Métodos mencionables:
addArista(): Añade una arista al conjunto y actualiza el peso total.
copy(): Copia otro objeto Resultado en este objeto, replicando el origen, las aristas y el peso total.

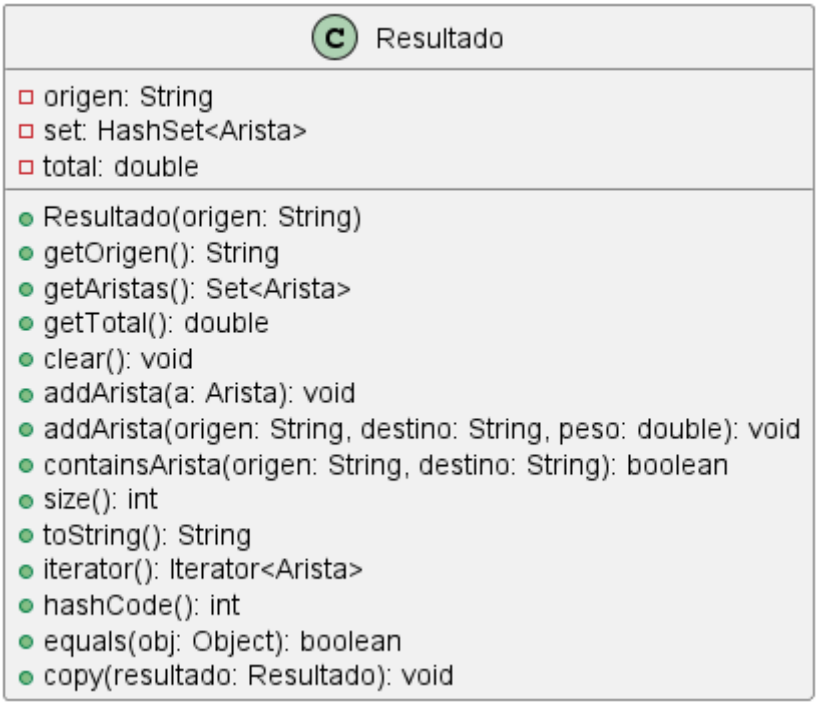

Imagen 21: Diagrama clase Resultado.

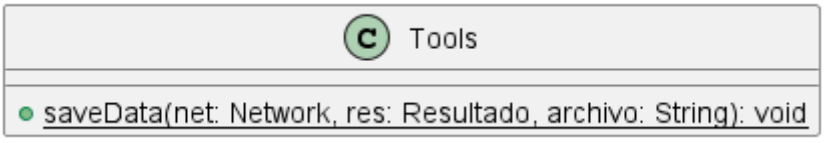

Imagen 22: Diagrama clase Tools.

4.3 Listado de archivos fuente.

Siguiendo la estructura de moralidad para su reutilización así como los documentos necesarios para probar los algoritmos una paquete siguiendo la nomenclatura indicada con las clases ya expuestas.

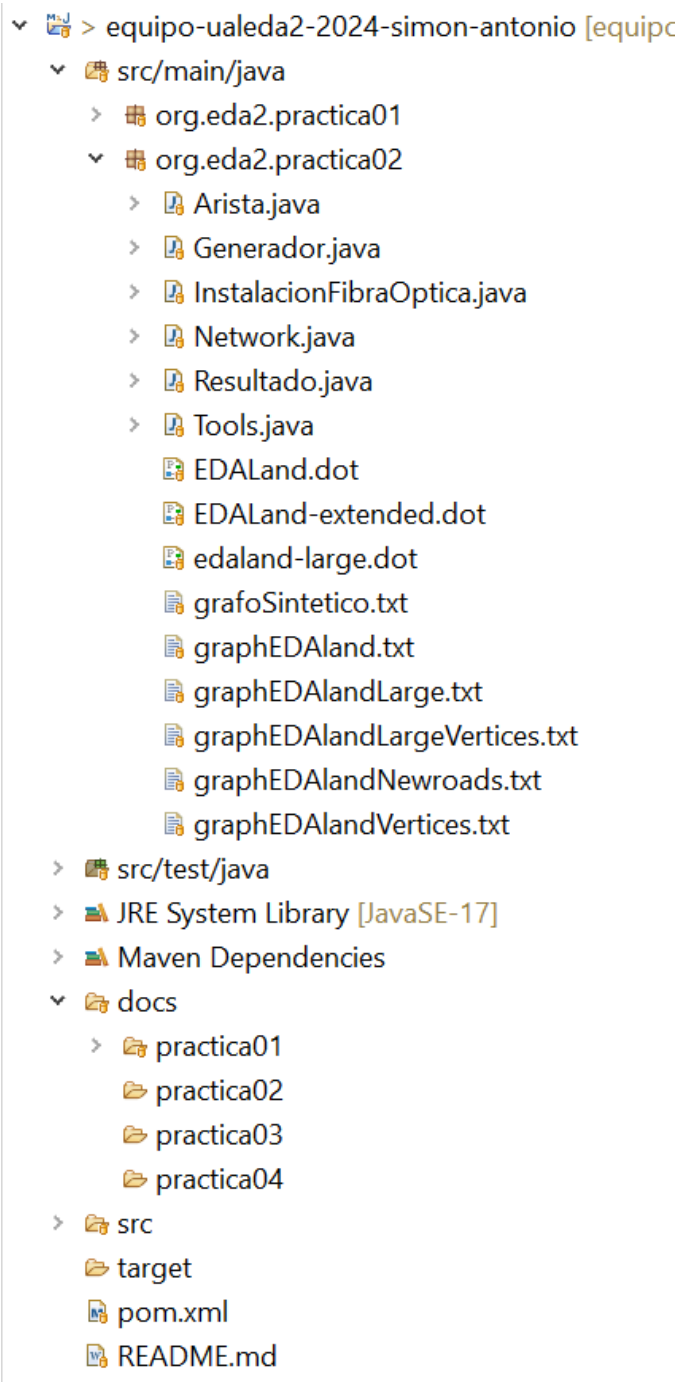

Imagen 23: Listado archivos fuente en eclipse.

5.- Anexo 2.

5.1 Cálculos realizados.

5.1.1 Prim sin cola de prioridad.

Resultados de la gráfica de resultados de los .txt dados en la práctica para la ejecución con Prim.

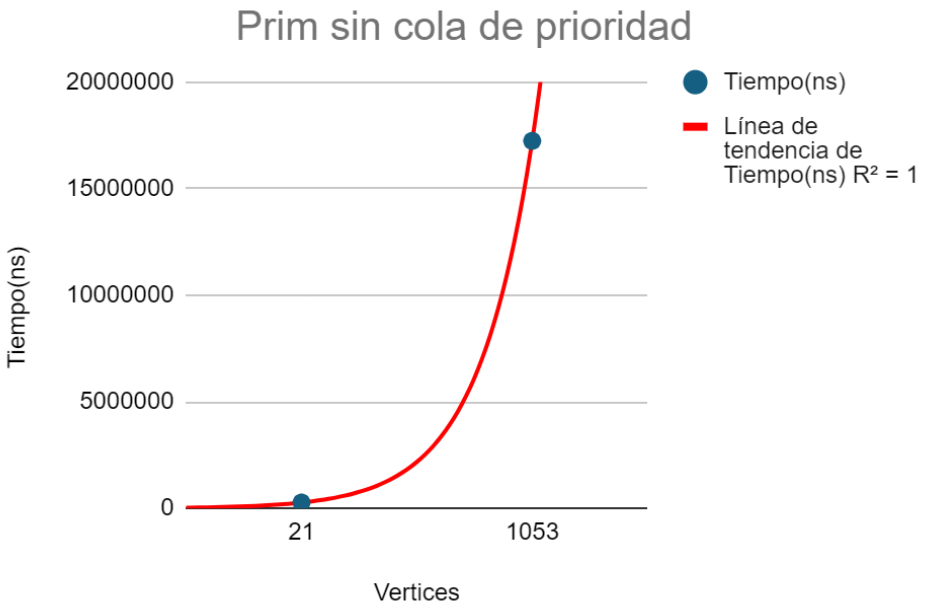

Imagen 24: Gráfico Prim sin cola de prioridad.

5.1.2 Prim con cola de prioridad.

Resultados de la gráfica de resultados de los .txt dados en la práctica para la ejecución de Prim con PriorityQueue.

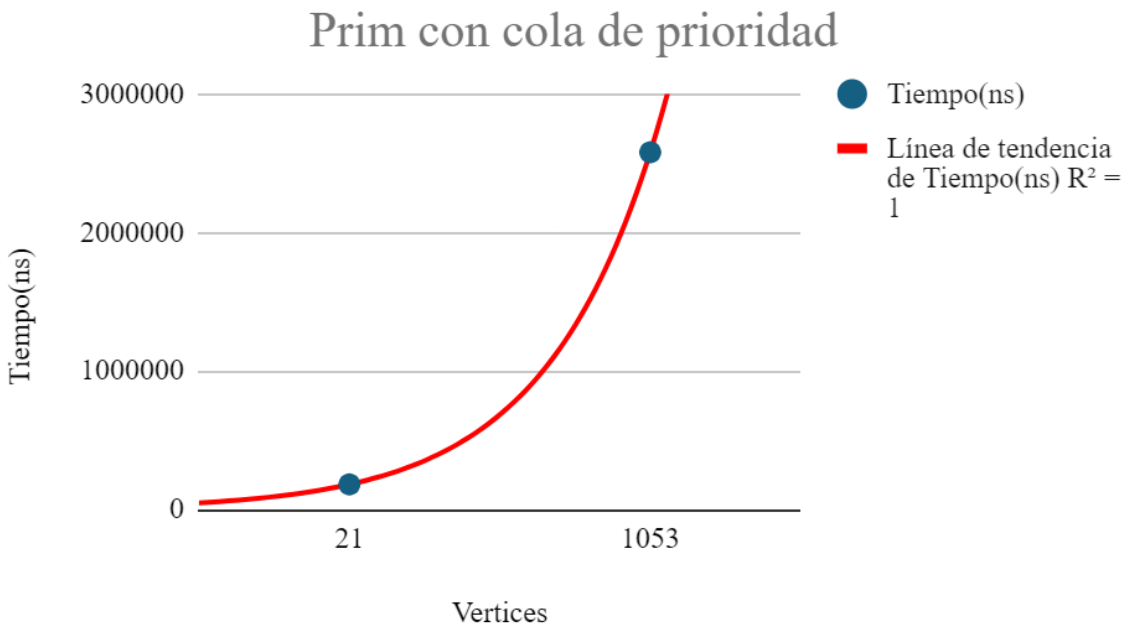

Imagen 25: Gráfico Prim con cola de prioridad.

5.1.3 Kruskal.

Resultados de la gráfica de resultados de los .txt dados en la práctica para la ejecución con Kruskal.

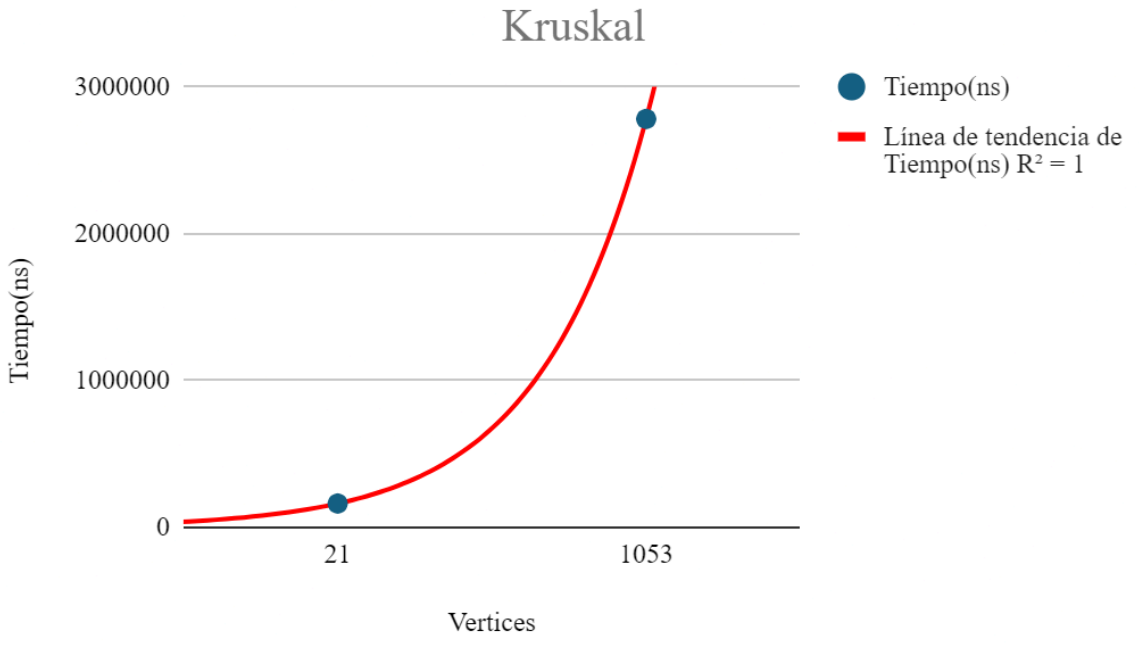

Imagen 26: Gráfico Kruskal.

5.1.4 Viajante de comercio con Caminos simples.

Resultados de la gráfica de resultados de los .txt dados en la práctica para la ejecución de el algoritmo TSP con caminos simples.

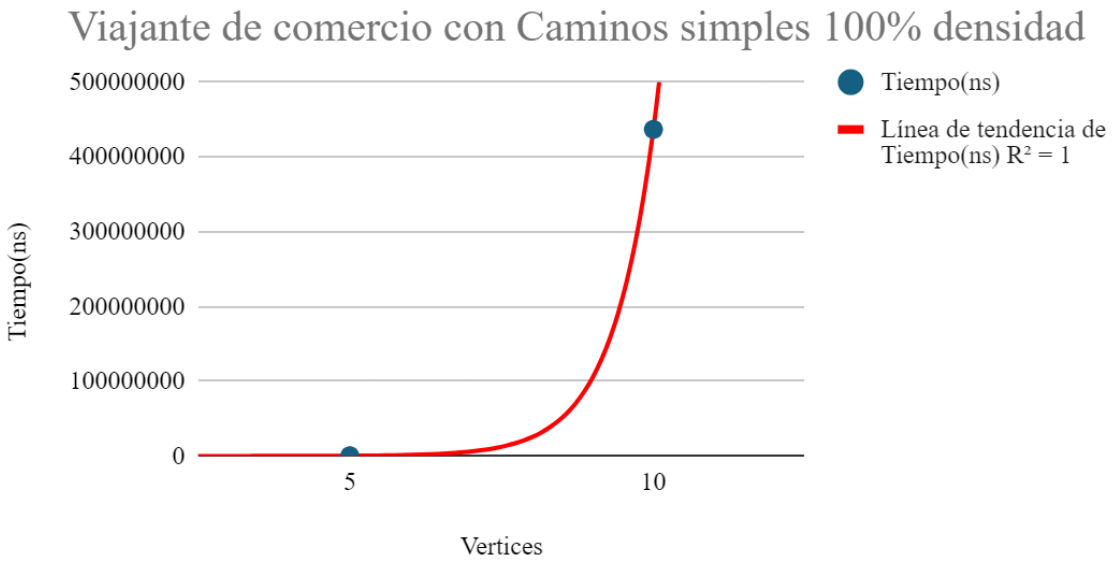

Imagen 27: Gráfico Viajante de comercio con Caminos simples.

5.1.5 Viajante de comercio con TSP

Resultados de la gráfica de resultados de los .txt dados en la práctica para la ejecución de el algoritmo TSPGreedy

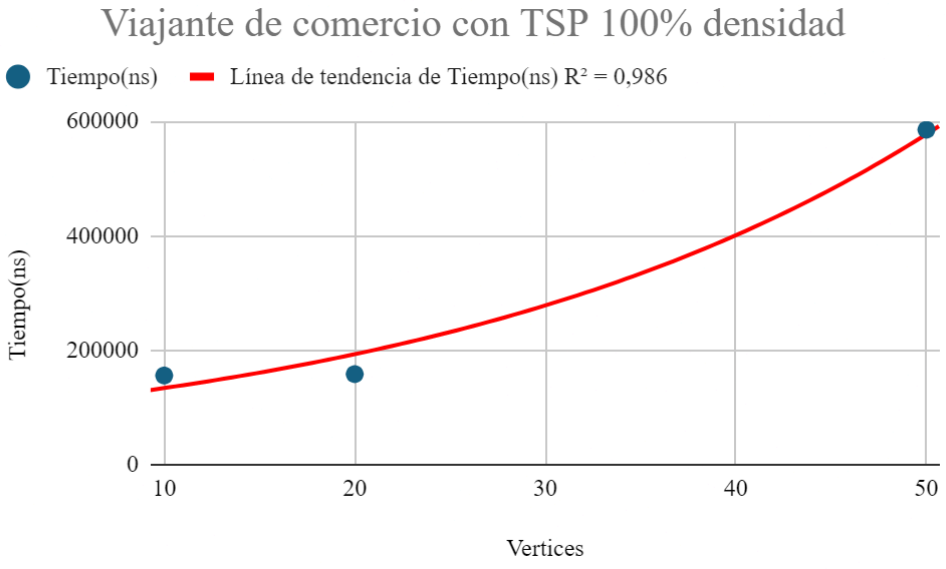

Imagen 28: Gráfico Viajante de comercio con Caminos simples.

5.1 Conclusiones finales.

Como podemos comprobar la correlación del estudio teórico con el experimental es muy positiva.

En el desarrollo de esta práctica se han puesto en valor diferentes enfoques con los que atacar un problema real como es la instalación de redes a través de grafos conexos.

Como puntos positivos respecto a la anterior se ha mejorado en la modularidad para mejorar la reusabilidad, la legibilidad y una compenetración en equipo más eficiente.

Se ha trabajado también en el quinto punto de la relación no obstante no se ha conseguido finalizar con total satisfacción, de igual manera se presenta hasta donde se ha alcanzado en este caso concreto.

Como puntos negativos estamos ante un proyecto que requiere de una planificación previa más extensa, cuestión que en un inicio no se planteó pero a la larga fue un factor que impactó negativamente en nuestro trabajo, poniendo en valor que la programación no se centra en la codificación exclusiva del problema, sino con planteamientos más abstractos de resolución general se pueden obtener mejores resultados.

6.- Bibliografía.

- UNE 157001:2014 Criterios generales para la elaboración formal. . . (s. f.). https://www.une.org/encuentra-tu-norma/busca-tu-norma/norma?c=N0052985
- Scribbr. (s. f.). Guía rápida de cómo citar en APA según su 7a edición. https://www.scribbr.es/category/normas-apa/
- Algoritmos voraces | Aprende programación competitiva. (s. f.). https://aprende.olimpiada-informatica.org/algoritmia-voraz
- Árboles de peso mínimo: algoritmos de Prim y Kruskal — Matemáticas Discretas para Ciencia de Datos. (s. f.). https://madi.nekomath.com/P5/ArbolPesoMin.html#:~:text=El%20algoritmo%20de%20Prim%20es,no%20est%C3%A9%20en%20el%20%C3%A1rbol.
- VGA. Un visualizador genérico de algoritmos. El Algoritmo de Dijkstra o de Caminos mínimos. (s. f.). http://atlas.uned.es/algoritmos/voraces/dijkstra.html
- Apuntes de clase Estructura de datos y algoritmos 2, Universidad de Almería.
- Apuntes de clase Lógica y algoritmica, Universidad de Almería.