Juan Raul Mellado Garcia y Alejandro Martinez Terrones

Practica 2 EDA II
Algoritmo Greedy

Introducción a Greedy

Greedy es un algoritmo que se basa en intentar darnos una solución de la manera más rápida posible, por esto no siempre es seguro que el algoritmo vaya a dar con una solución y en el caso de encontrarla tampoco nos asegura que sea la óptima.

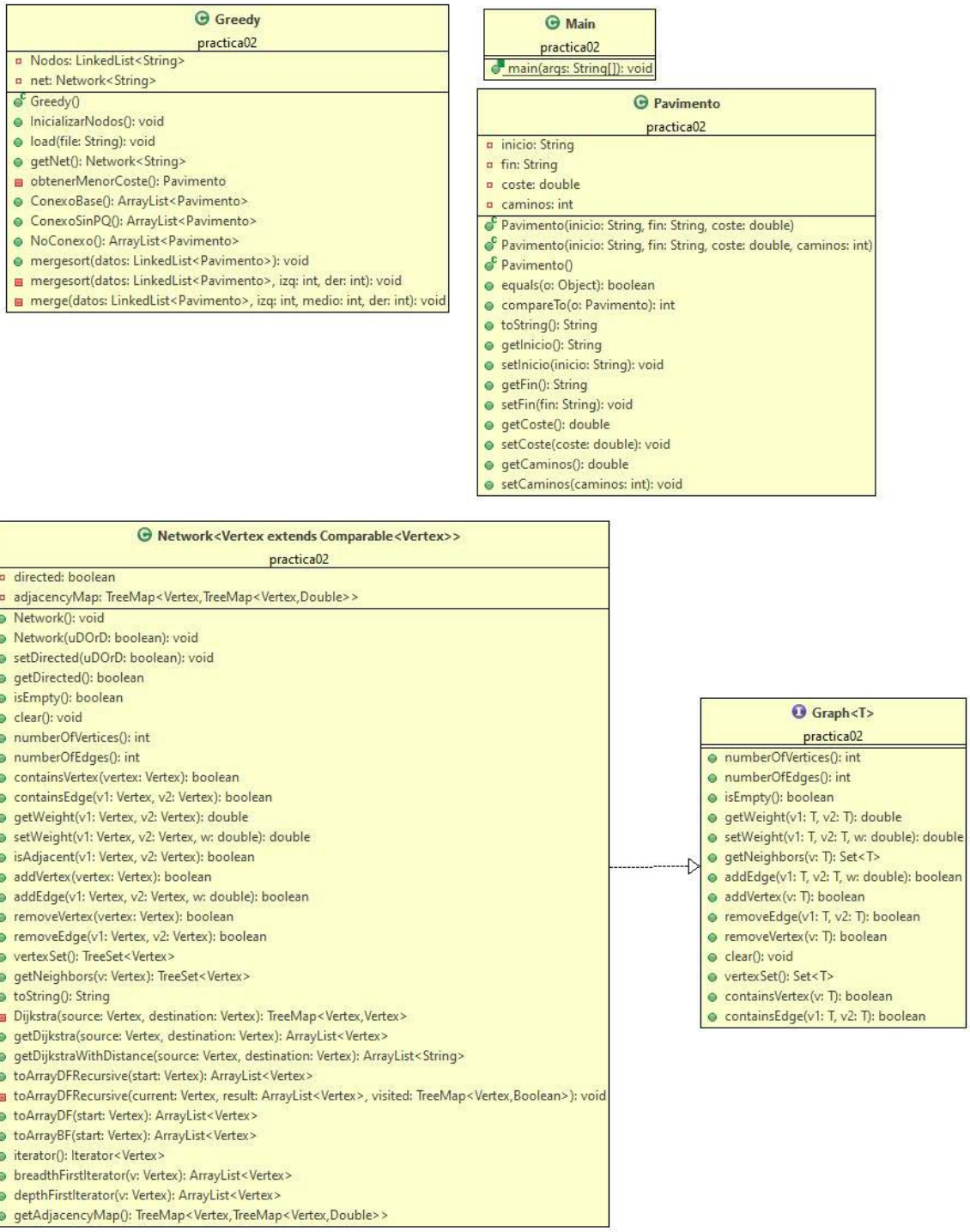
Se suele usar en problemas de optimización, y su funcionamiento se basa en etapas y en tomar una decisión con la información disponible en cada una de ellas y estas decisiones no se vuelven a replantear y en base a esto se van generando dos conjuntos , el conjunto

de seleccionados y el de descartados, después de cada se comprueba si lo obtenido hasta el momento es una solución parcial o ya es la solución completa.

Estudio de la implementación

General

Lo primero para poder entender la implementación de cada uno de los algoritmos es echar un vistazo a la estructuras de datos generales que tenemos para ello hemos elaborado un diagrama UML



Como podemos ver aquí contamos con diferentes clases, por ejemplo, el grafo como tal lo almacenamos con la clase Network , la cual hereda de Graph<T>, luego contamos con

nuestra clase Greedy donde implementamos los diferente métodos y algoritmos que vamos a necesitar para procesar los datos, pavimento es la clase que hemos usado para designar a la aristas a la hora de usarla dentro de los algoritmos y para mantener un buena encapsulación, tenemos la clase main desde la cual solo llamamos a métodos para realizar las pruebas.

Prim

```
public ArrayList<Pavimento> ConexoSinPQ() {

    ArrayList<Pavimento> result = new ArrayList<>();
    Pavimento pav = obtenerMenorCoste();
    ArrayList<Pavimento> cola = new ArrayList<Pavimento>();
    LinkedList<String> Visitados = new LinkedList<>();


    Visitados.add(pav.getInicio());

    for (String ciudad : this.net.getAdjacencyMap().get(pav.getFin()).keySet()) {
        if (Visitados.contains(ciudad))
            continue;
        Pavimento aux = new Pavimento(pav.getFin(), ciudad, this.net.getWeight(pav.getFin(), ciudad));
        cola.add(aux);
    }

    while (Nodos.size() != Visitados.size()) {
        if (!Visitados.contains(pav.getFin())) {
            Visitados.add(pav.getFin());
            result.add(pav);
            for (String ciudad : this.net.getAdjacencyMap().get(pav.getFin()).keySet()) {
                if (Visitados.contains(ciudad))
                    continue;
                Pavimento aux = new Pavimento(pav.getFin(), ciudad, this.net.getWeight(pav.getFin(), ciudad));
                cola.add(aux);
            }
        }
        mergesort(cola);
        pav = cola.get(cola.size() - 1);
        cola.remove(cola.size() - 1);

    }

    return result;
}
```

Aquí podemos ver la implementació de nuestro prim, el cual se basa en tener un ArrayList ordenado con las posibles aristas disponibles en ese momento para que seleccione entre ellas la de menos coste , cumpliendo la condición de que no se haya visitado el nodo destino, el funcionamiento es el siguiente:

Seleccionamos la arista de menor coste y agregamos todas las aristas de los nodos que conecta y que su destino no haya sido visitado.

Una vez tenemos esto, ordenamos la cola y empezamos el bucle, el cual tiene la condición de estar iterando hasta que el número de nodos visitados sea el mismo que el número de nodos totales.

Dentro del bucle se dedica a añadir el nodo final de la arista seleccionada visitados, introducir esa arista en la solución e introducir en la cola todas las arista de este que no acaben en un nodo ya visitado, una vez hecho, se ordenada el ArrayList mediante un MergeSort y se selecciona como arista actual de menor coste de este, empezando así una nueva interacción.

Una vez acabado el bucle se devuelve el resultado.

Prim con PriorityQueue

```
public ArrayList<Pavimento> ConexoBase() {

    ArrayList<Pavimento> result = new ArrayList<>();
    Pavimento pav = obtenerMenorCoste();
    PriorityQueue<Pavimento> cola = new PriorityQueue<>();
    LinkedList<String> Visitados = new LinkedList<>();

    Visitados.add(pav.getInicio());

    for (String ciudad : this.net.getAdjacencyMap().get(pav.getFin()).keySet()) {
        if (Visitados.contains(ciudad))
            continue;
        Pavimento aux = new Pavimento(pav.getFin(), ciudad, this.net.getWeight(pav.getFin(), ciudad));
        cola.add(aux);
    }

    while (Nodos.size() != Visitados.size()) {
        if (!Visitados.contains(pav.getFin())) {
            Visitados.add(pav.getFin());
            result.add(pav);
            for (String ciudad : this.net.getAdjacencyMap().get(pav.getFin()).keySet()) {
                if (Visitados.contains(ciudad))
                    continue;
                Pavimento aux = new Pavimento(pav.getFin(), ciudad, this.net.getWeight(pav.getFin(), ciudad));
                cola.add(aux);
            }
        }
        pav = cola.poll();

    }

    return result;
}
```

El funcionamiento es exactamente el mismo que el caso anterior solo que al estar usando una PriorityQueue no tenemos que ordenar la estructura ya que se mantiene ordenada de manera automática y por inserción .

Kruskal

```
public ArrayList<Pavimento> NoConexo() {

    ArrayList<Pavimento> result = new ArrayList<>();
    Pavimento pav = new Pavimento();
    PriorityQueue<Pavimento> cola = new PriorityQueue<Pavimento>();
    LinkedList<String> Visitados = new LinkedList<>();


    for(String city : this.Nodos) {
        for (String ciudad : this.net.getAdjacencyMap().get(city).keySet()) {
            Pavimento aux = new Pavimento(city, ciudad, this.net.getWeight(city, ciudad));
            cola.add(aux);
            }
    }


    while (Nodos.size() != Visitados.size()){
        pav = cola.poll();
        if (!Visitados.contains(pav.getFin()) || !Visitados.contains(pav.getInicio())) {
            result.add(pav);
            if (!Visitados.contains(pav.getFin())) Visitados.add(pav.getFin());
            if (!Visitados.contains(pav.getInicio())) Visitados.add(pav.getInicio());
        }
    }

    return result;
}
```

En este caso lo que hacemos es añadir todas las aristas a una LinkedList y con un MergeSort en función del menor peso.

Una vez tenemos esto nos metemos en un bucle el cual itera hasta que el número de nodos visitados sea el mismo, que de nodo. Dentro de este bucle lo que se hace es coger el primer elemento de la lista y añadirlos al resultado si alguno o ambos extremos no ha sido visitado y los añade a la lista de nodos visitados.

Estudio Teórico

Teóricamente tenemos que Prim es un algoritmo con O(n^2) y Kruskal O(a log n) siendo n el número de nodos y a el número de arista, lo cual hace que Prim sea un algoritmo polinómico y Kruskal un algoritmo enelogarímico en casos normales , por lo cual podríamos asumir que Prim es peor algoritmo.

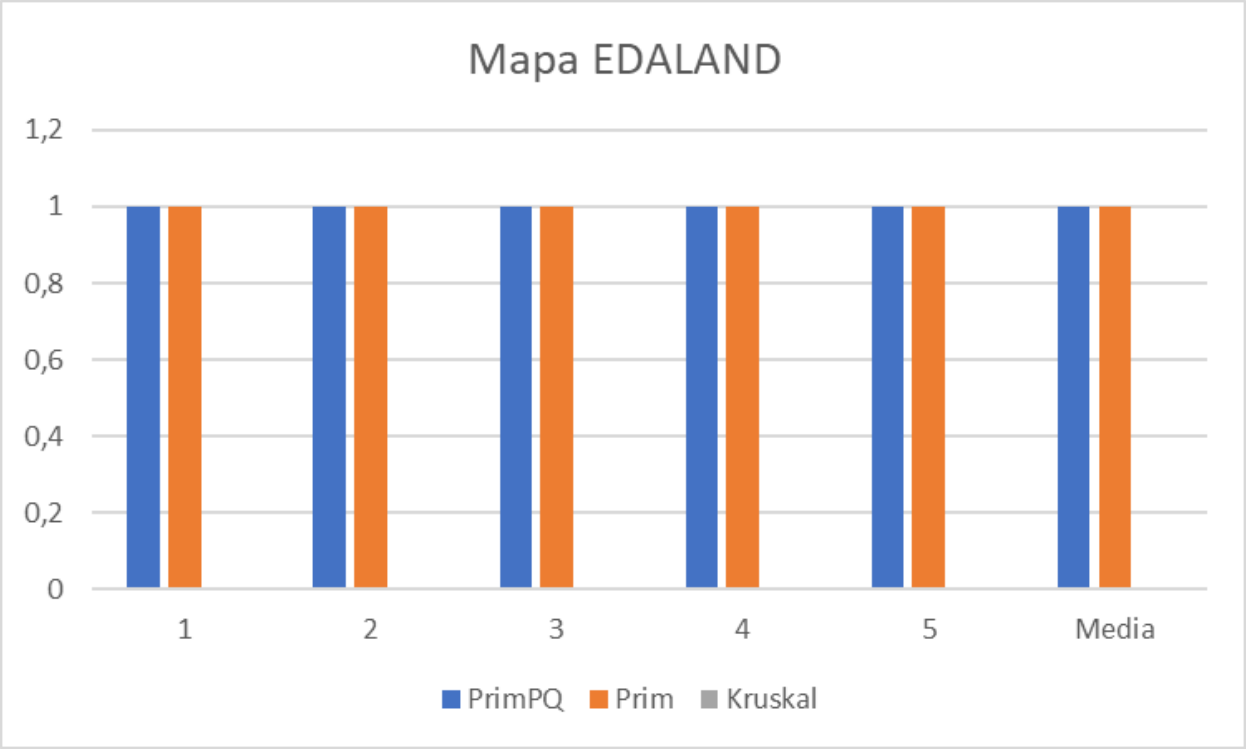
Pero dependiendo del tipo de mapa que tengamos, por ejemplo, esto podría cambiar un grafo denso(donde las aristas están cercanas al número máximo de estas) , teóricamente Prim debería funcionar mejor.

Y también nos podemos encontrar el caso contrario , si el grafo es disperso Prim pierde aún más eficiencia y Kruskal destaca más.

Además en nuestro caso para Prim tenemos dos variantes con PQ que hacer ordenamiento por inserción y con un ArrayList y MergeSort que ordena el ArrayList completamente antes de sacar algo de este , por lo cual aumenta mucho el coste algorítmico y teóricamente el tiempo debería empeorar al no usar las PriorityQueue.

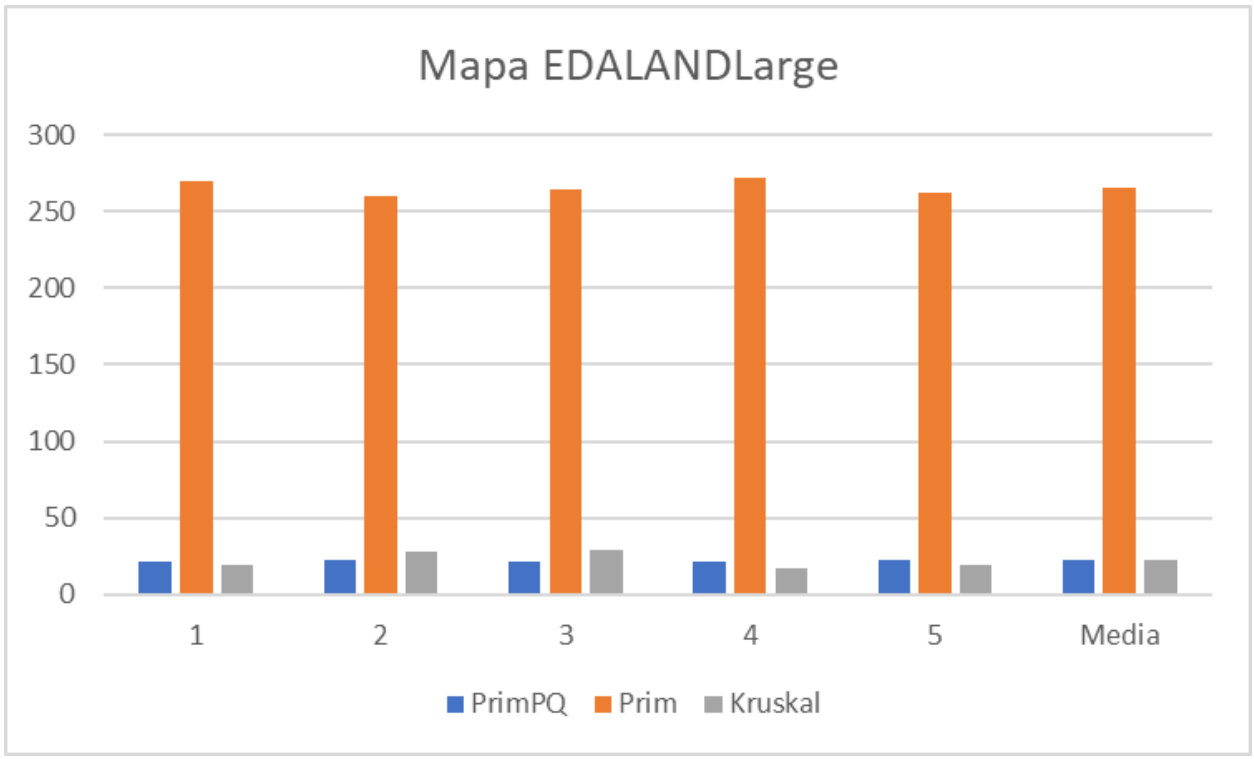
Estudio Experimental

Mapa EDALAND



Este mapa es demasiado pequeño para tenerlo en cuenta ya que incluso midiendo el tiempo en milisegundos el tiempo de Kruskal es 0 , por lo cual este conjunto de datos solo es válido para desarrollar los algoritmos y hacer el juego de pruebas en base a él.

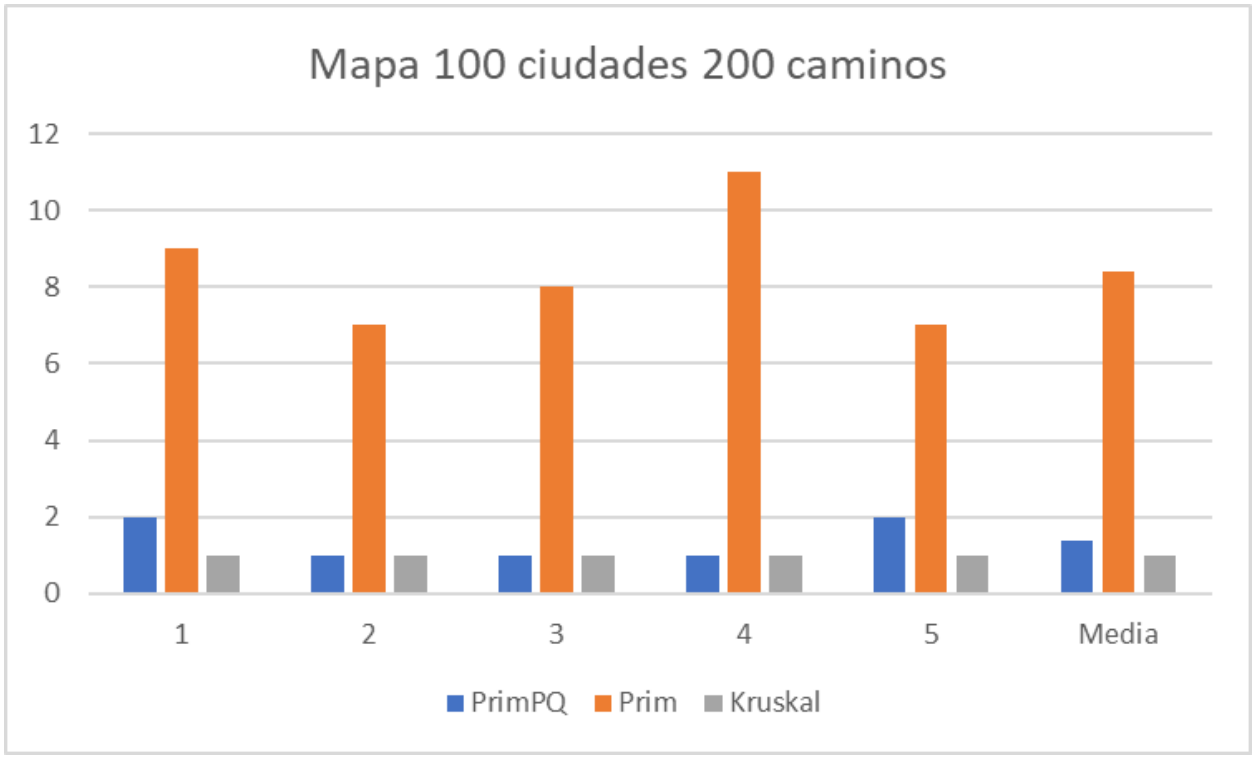
Mapa EDALAND Large



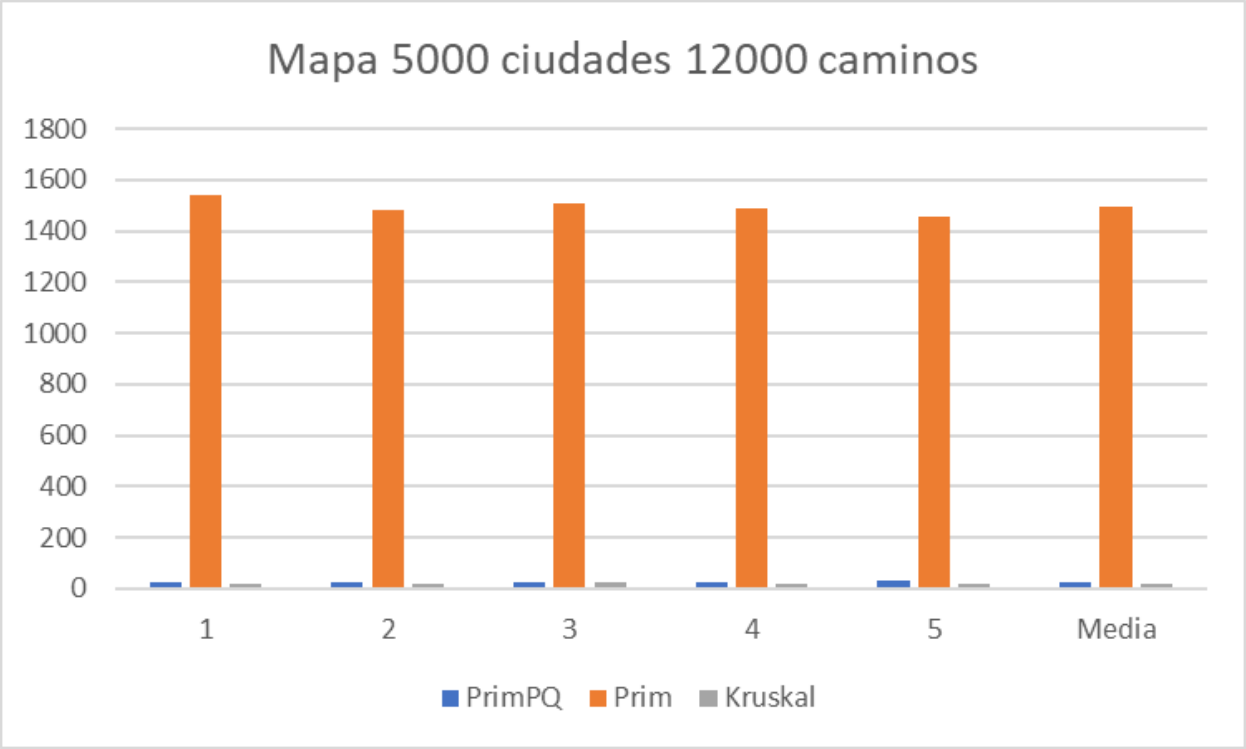
Como ya habíamos previsto en el análisis teórico el tiempo de Prim con ArrayList en nuestro caso es mucho peor ya que se ordena el array entero cada vez que se va a coger un elemento de este, mientras en que la PriorityQueue que se usa en el otro Prim y en Kruskal se está ordenando por inserción.

Mapa Mapa Generado

100 ciudades 200 caminos

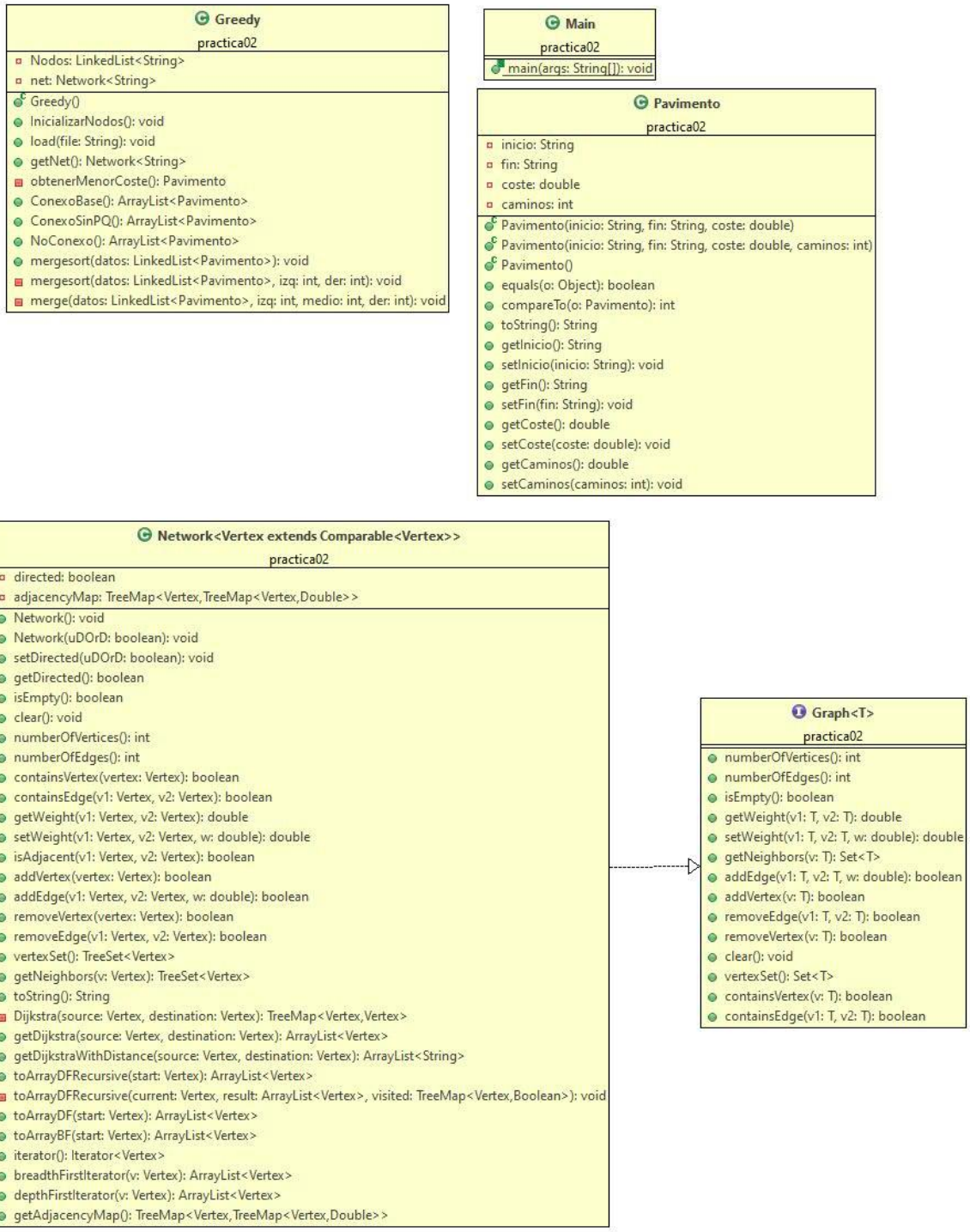


1000 ciudades 2500 caminos



Como podemos ver , al igual que en los casos anteriores Kruskal sigue siendo un poco mas optimo que Prim con Cola de Prioridad y el caso de Prim con el MergeSort sigue siendo el peor de los 3 de lejos, este tiempo se podría mejorar programando nosotros mismo un algoritmo de ordenación por inserción pero el tiempo sería bastante similar al de usar la PriorityQueue.

Anexo



Bibliografía

Para la realización de esta práctica solo se ha consultado los apuntes de la asignatura.