Tabla de contenido

Ejercicio1

DatosEjercicio1.java

```
package _datos;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.stream.Collectors;

import us.lsi.common.Files2;
import us.lsi.common.String2;

public class DatosEjercicio1 {

	public static List<Tipo> tipos;
	public static record Tipo(String Nombre_Tipo, Integer kgdisponibles, Integer id) {
		public Tipo of(String Nombre_Tipo, Integer kgdisponibles, Integer id) {
			return new Tipo(Nombre_Tipo, kgdisponibles, id);
		}

		public static Tipo ofFormat(String linea) {
			String[] formato = linea.split(":");
			String Nombre_Tipo = formato[0];
			Integer kgdisponibles =
Integer.parseInt(formato[1].replace("kgdisponibles=", "").replace(";", "").trim());
			Integer id = Integer.parseInt(formato[0].replace("C", ""));
			return new Tipo(Nombre_Tipo,kgdisponibles, id);
		}

	}

	public static List<Variedad> variedades;
	public static record Variedad(String nombre, Integer id, Double beneficio, Map<String,
Double> porcentaje) {
		public Variedad of(String nombre, Integer id, Double beneficio, Map<String,
Double> porcentaje) {
			return new Variedad(nombre, id, beneficio, porcentaje);
		}

		public static Variedad ofFormat(String linea) {
			String[] formato = linea.split("->");
			String nombre = formato[0].trim();
			String segundaParte = formato[1].trim();
			String[] segundaPartePars = segundaParte.split(";");
			Double beneficio =
Double.parseDouble(segundaPartePars[0].split("=")[1].trim());
			Map<String, Double> porcentaje =
Arrays.stream(segundaPartePars[1].split("=")[1].split(","))
					.map(pair -> pair.split(":"))
					.collect(Collectors.toMap(
							keyValue -> keyValue[0].replace("(", "").replace(")", ""),
							keyValue -> Double.parseDouble(keyValue[1].replace("(",
"").replace(")", ""))
					));

			Integer id = null;
```

```
                return new Variedad(nombre, id, beneficio, porcentaje);
            }
        }


        public static void iniDatos(String fichero) {
            List<String> lineas = Files2.linesFromFile(fichero);
            tipos = lineas.stream().filter(l -> l.startsWith("C")).map(x ->
Tipo.ofFormat(x)).toList();
            variedades = lineas.stream().filter(l -> l.startsWith("P")).map(x ->
Variedad.ofFormat(x)).toList();
            toConsole();
        }

        //.....................................................
        //head section del .lsi

        public static Integer getTipos() {
            return tipos.size();
        }

        public static Integer getVariedades() {
            return variedades.size();
        }

        public static Double getBeneficio(Integer i) {
            return variedades.get(i).beneficio;
        }
        public static Double getPorcentajeVariedad(Integer i, Integer j) {
            Set<String> setTipos = variedades.get(i).porcentaje.keySet();
            String aux = tipos.stream().filter(x -> tipos.indexOf(x) ==
j).map(Tipo::Nombre_Tipo).findFirst().orElse(null);
            return setTipos.contains(aux)?variedades.get(i).porcentaje().get(aux):0.;
        }
        public static Integer getKgDisponibles(Integer i) {
            return tipos.get(i).kgdisponibles;
        }
        public static Integer getKgDisponiblesMax() {
            return tipos.stream().map(Tipo::kgdisponibles).reduce((a, b) -> a > b?a:b).get();
        }

        public static Double getBounds(Integer i, Integer j) {
            Tipo tipo = tipos.get(i);
            Double porcentaje = getPorcentajeVariedad(i, j);
            return tipo.kgdisponibles()/porcentaje;
        }

        //.....................................................
        public static List<Tipo> getListaTipos(){
            return tipos;
        }
        public static List<Variedad> getListaVariedades(){
            return variedades;
        }

        public static void toConsole() {
            //String2.toConsole("Conjunto de Entrada Tipos: %s\nConjunto de Entrada
Variedades: %d", tipos, variedades);
            System.out.println(tipos);
            System.out.println(variedades);
        }
```

```
	// Test de la lectura del fichero
	public static void main(String[] args) {
		iniDatos("ficheros/Ejercicio1DatosEntrada1.txt");
	}
}
```

SolucionEjercicio1.java

```
package _soluciones;

import java.util.List;

import _datos.DatosEjercicio1;
import _datos.DatosEjercicio1.Variedad;
import us.lsi.common.List2;

public class SolucionEjercicio1 {

	public static SolucionEjercicio1 of_Range(List<Integer> ls) {
		return new SolucionEjercicio1(ls);
	}

	private Double beneficio; //beneficio que tenemos
	private List<Variedad> variedades; //variedades que tenemos
	private List<Integer> solucion; //cantidad que cogemos de cada variedad

	private SolucionEjercicio1() {
		beneficio = 0.;
		solucion = List2.empty();
		variedades = List2.empty();
	}
	private SolucionEjercicio1(List<Integer> ls) {
		beneficio  = 0.;
		solucion = List2.of();
		variedades = List2.empty();
		for(int i=0; i<ls.size(); i++) {
			if(ls.get(i)>0) {
				Integer e = ls.get(i); //x[i]
				Double v = DatosEjercicio1.getBeneficio(i); //beneficio
				beneficio += v * e; //beneificio += beneficio[i] * x[i]
				variedades.add(DatosEjercicio1.variedades.get(i));
				solucion.add(e);
			}
		}
	}

	public static SolucionEjercicio1 empty() {
		return new SolucionEjercicio1();
	}

	public static SolucionEjercicio1 create(List<Integer> ls) {
		return new SolucionEjercicio1(ls);
	}

	@Override
	public String toString() {
//		int error = Math.abs(DatosEjercicio1.getSuma() - suma);
//		String e = error<1? "": String.format("Error = %d", error);
		return String.format("Solucion = %s; Tamaño solucion = %d; beneficio total = %f;",
solucion, solucion.size(), beneficio);
	}
}
```

Ejercicio1Edge

```
package ejercicios.ejercicio1;

import _datos.DatosEjercicio1;
import us.lsi.graphs.virtual.SimpleEdgeAction;

public record Ejercicio1Edge(Ejercicio1Vertex source,
                             Ejercicio1Vertex target,
                             Integer action, //cuantas veces se coge el
numero en cuestion
                             Double weight)
implements SimpleEdgeAction<Ejercicio1Vertex,Integer> {

	public static Ejercicio1Edge of(Ejercicio1Vertex s, Ejercicio1Vertex t, Integer a) {
		// TODO La arista debe tener peso

		return new Ejercicio1Edge(s, t, a, a * DatosEjercicio1.getBeneficio(s.index()));
	}

	@Override
	public String toString() {
		return String.format("%d; %.1f", action, weight);
	}

}
```

Ejercicio1Heuristic

```
package ejercicios.ejercicio1;

import java.util.List;
import java.util.Map;
import java.util.function.Predicate;
import java.util.stream.IntStream;

import _datos.DatosEjercicio1;
import _datos.DatosEjercicio1.Tipo;
import _datos.DatosEjercicio1.Variedad;
import us.lsi.common.List2;

public class Ejercicio1Heuristic {

	// Se explica en practicas.
	public static Double heuristic( Ejercicio1Vertex v1,
	                                Predicate<Ejercicio1Vertex> goal,
	                                Ejercicio1Vertex v2)
	{
		//multiplicar el beneficio maximo por la cantidad que puedo coger de esa variedad
		Double res = 0.;
		List<Variedad> variedades = DatosEjercicio1.variedades;
		for(Variedad v : variedades) {
			Double resAux = v.beneficio()*cantidadDisponible(v1, v);
			if(resAux>res) res = resAux;
		}
		return res;

	}


	public static Integer cantidadDisponible(Ejercicio1Vertex v1, Variedad v) {
```

```
            List<Integer> disponibles = List2.empty();   //aqui almaceno los Kg que tengo
disponibles para cada tipo
            for (Map.Entry<String, Double> entry : v.porcentaje().entrySet()) {
                  String tipo = entry.getKey();
                  Tipo tipoNombre = DatosEjercicio1.tipos.stream().filter(x ->
x.Nombre_Tipo().equals(tipo)).findFirst().get();
                  Integer i = DatosEjercicio1.tipos.indexOf(tipoNombre); //indice del tipo en
el que estoy

                  Double porcentajeCojo = entry.getValue();
                  Integer disponible = 0;
                  while(v1.remaining().get(i)>=(porcentajeCojo*(disponible + 1))) {
                        disponible++;
                  }

                  disponibles.add(disponible);    //añado la cantidad disponible para este
tipo

            }
            Integer disponible = disponibles.stream().min(Integer::compare).get(); //me quedo
con la menor de ellas... esta será la mayor cantidad

                                          //que puedo coger de esta variedad
            return disponible;
      }
}
```

Ejercicio1Vertex

```
package ejercicios.ejercicio1;

import java.util.Comparator;
import java.util.List;
import java.util.Map;
import java.util.function.Predicate;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

import _datos.DatosEjercicio1;
import _datos.DatosEjercicio1.Tipo;
import _datos.DatosEjercicio1.Variedad;
import us.lsi.common.List2;
import us.lsi.graphs.virtual.VirtualVertex;

public record Ejercicio1Vertex(Integer index, List<Integer> remaining)
implements VirtualVertex<Ejercicio1Vertex, Ejercicio1Edge, Integer> {

	public static Ejercicio1Vertex of(Integer i, List<Integer> rest) {
		return new Ejercicio1Vertex(i, rest);
	}

	// TODO Consulte las clases GraphsPI5 y TestPI5

	@Override
	public List<Integer> actions() { //cuanto cojo en cada caso
		// TODO Alternativas de un vertice
		List<Integer> res = List2.empty();

		//si he llegado al final
		if(this.index==DatosEjercicio1.getVariedades()) {
			return res;
		}else if(this.remaining.stream().allMatch(x -> x==0)) {//si me quedo sin café
			res.add(0);
			return res;
		}else { //en caso contrario, selecciono el número de kilos disponibles
			Variedad v = DatosEjercicio1.variedades.get(this.index); //variedad actual
			Integer disponible = cantidadDisponible(v);
			res =  IntStream.rangeClosed(0, disponible) //creo un stream de enteros
desde 0 a disponible (Incluidos)
					.boxed() //devuelvo un stream a partir de esto
					.toList(); //lo paso a lista
		}
		return res;

	}

	@Override
	public Ejercicio1Vertex neighbor(Integer a) {
		// TODO Vertice siguiente al actual segun la alternativa a
		List<Integer> remaining2 = List2.copy(this.remaining);
		remaining2 = actualizaRemaining(a);
		return new Ejercicio1Vertex(index + 1, remaining2); //actualizarlo
	}

	@Override
	public Ejercicio1Edge edge(Integer a) {
		return Ejercicio1Edge.of(this, neighbor(a), a); //unir vertice con su vecino
	}
```

```
	// Se explica en practicas.
	public Ejercicio1Edge greedyEdge() {
		if (existeMayorMejor()) {
			return edge(0);
		} else {
			Variedad variedadActual = DatosEjercicio1.variedades.get(index);
			int kg = cantidadDisponible(variedadActual);
			return edge(kg);
		}
	}

	private Boolean existeMayorMejor() {
		Variedad variedadBenMax = IntStream.range(this.index + 1,
DatosEjercicio1.getVariedades())
				.boxed()
				.map(i -> DatosEjercicio1.variedades.get(i))
				.sorted(Comparator.comparingDouble(Variedad::beneficio).reversed())
				//.filter(x -> cumpleCondicion(x, this.remaining))
				.findFirst().orElse(null);
		if(variedadBenMax!=null) return resuelveProblema(variedadBenMax) &&
variedadBenMax.beneficio() > DatosEjercicio1.getBeneficio(index);
		else return false;
	}


	@Override
	public String toString() {
		return "Ejercicio1Vertex [index=" + index + ", remaining=" + remaining + "]";
	}

	public static Ejercicio1Vertex initial() {
		// TODO Auto-generated method stub
		return new Ejercicio1Vertex(0,
DatosEjercicio1.tipos.stream().map(Tipo::kgdisponibles).collect(Collectors.toList()));
	}

	public static Predicate<Ejercicio1Vertex> goal() {
		// TODO Auto-generated method stub
		Predicate<Ejercicio1Vertex> pred =  p -> p.index() ==
DatosEjercicio1.getVariedades(); //recorro las variedades
		return pred;
	}

	public static Predicate<Ejercicio1Vertex> goalHasSolution() {
		// TODO Auto-generated method stub
		Predicate<Ejercicio1Vertex> pred =  p -> p.remaining().stream().allMatch(x -> x>=
0); //tiene solucion si no me he pasado cogiendo kilos
		return pred;
	}

	public static Boolean cumpleCondicion(Variedad variedadActual, List<Integer> remai) {
		Map<String, Double> porcentaje = variedadActual.porcentaje();
		Boolean res = true;

		for (Map.Entry<String, Double> entry : porcentaje.entrySet()) {
			String tipo = entry.getKey();
			Tipo tipoNombre = DatosEjercicio1.tipos.stream().filter(x ->
x.Nombre_Tipo().equals(tipo)).findFirst().get();
			Integer i = DatosEjercicio1.tipos.indexOf(tipoNombre); //indice del tipo en
el que estoy
```

```
                Double porcentajeCojo = entry.getValue();
                if(remai.get(i)<porcentajeCojo) return false;
            }

            return res;
        }

        public Integer cantidadDisponible(Variedad v) {
            List<Integer> disponibles = List2.empty();   //aqui almaceno los Kg que tengo
disponibles para cada tipo
            for (Map.Entry<String, Double> entry : v.porcentaje().entrySet()) {
                String tipo = entry.getKey();
                Tipo tipoNombre = DatosEjercicio1.tipos.stream().filter(x ->
x.Nombre_Tipo().equals(tipo)).findFirst().get();
                Integer i = DatosEjercicio1.tipos.indexOf(tipoNombre); //indice del tipo en
el que estoy

                Double porcentajeCojo = entry.getValue();
                Integer disponible = 0;
                while(this.remaining.get(i)>=(porcentajeCojo*(disponible + 1))) {
                    disponible++;
                }
                disponibles.add(disponible);    //añado la cantidad disponible para este
tipo

            }

            Integer disponible = disponibles.stream().min(Integer::compare).get(); //me quedo
con la menor de ellas... esta será la mayor cantidad

            return disponible;
        }

        public Boolean resuelveProblema(Variedad v) {
            Integer cantidadDisp = cantidadDisponible(v);
            List<Integer> remain = actualizaRemaining( cantidadDisp);
            return remain.stream().allMatch(x -> x == 0);
        }

        public List<Integer> actualizaRemaining(Integer a){
            List<Integer> remaining2 = List2.copy(this.remaining);
            Variedad variedadActual = DatosEjercicio1.variedades.get(this.index);
            Map<String, Double> porcentaje = variedadActual.porcentaje();

            for (Map.Entry<String, Double> entry : porcentaje.entrySet()) {
                String tipo = entry.getKey();
                Double porcentajeCojo = entry.getValue();
                Tipo tipoNombre = DatosEjercicio1.tipos.stream().filter(x ->
x.Nombre_Tipo().equals(tipo)).findFirst().get();
                Integer i = DatosEjercicio1.tipos.indexOf(tipoNombre); //indice del tipo en
el que estoy
                Integer cantidadActual = remaining2.get(i);
                Integer cojo = (int) Math.floor(a*porcentajeCojo);//actualizo remaining
                remaining2.set(i, cantidadActual-cojo);
            }
            return remaining2;
        }


}
```

Manual

Ejercicio1PDR

```
package ejercicios.ejercicio1.manual;

import java.util.Comparator;
import java.util.List;
import java.util.Map;

import _datos.DatosEjercicio1;
import _soluciones.SolucionEjercicio1;
import us.lsi.common.List2;
import us.lsi.common.Map2;

public class Ejercicio1PDR {

	public static record Spm(Integer a, Integer weight) implements Comparable<Spm> {
		public static Spm of(Integer a, Integer weight) {
			return new Spm(a, weight);
		}
		@Override
		public int compareTo(Spm sp) {
			return this.weight.compareTo(sp.weight);
		}
	}

	public static Map<Ejercicio1Problem, Spm> memory;
	public static Integer mejorValor;

	public static SolucionEjercicio1 search() {
		memory =  Map2.empty();
		mejorValor = Integer.MIN_VALUE; // Estamos maximizando

		pdr_search(Ejercicio1Problem.initial(), 0, memory);
		return getSolucion();
	}

	private static Spm pdr_search(Ejercicio1Problem prob, Integer acumulado,
Map<Ejercicio1Problem, Spm> memoria) {

		Spm res = null;
		Boolean esTerminal = prob.index().equals(DatosEjercicio1.getVariedades());
		Boolean esSolucion = prob.remaining().stream().allMatch(x -> x>= 0);

		if (memory.containsKey(prob)) {
			res = memory.get(prob);

		} else if (esTerminal && esSolucion) {
			res = Spm.of(null, 0);
			memory.put(prob, res);
			if (acumulado > mejorValor) { // Estamos maximizando
				mejorValor = acumulado;
			}
		} else {
			List<Spm> soluciones = List2.empty();
			for (Integer action : prob.actions()) {
				Double cota = acotar(acumulado, prob, action);
				if (cota < mejorValor) {
					continue;
```

```
                        }
                        Ejercicio1Problem vecino = prob.neighbor(action);
                        Spm s = pdr_search(vecino, acumulado + action, memory);
                        if (s != null) {
                              Spm amp = Spm.of(action, s.weight() + action);
                              soluciones.add(amp);
                        }
                  }
                  // Estamos maximizando
                  res = soluciones.stream().max(Comparator.naturalOrder()).orElse(null);
                  if (res != null)
                        memory.put(prob, res);
            }

            return res;
      }

      private static Double acotar(Integer acum, Ejercicio1Problem p, Integer a) {
            return acum + a + p.neighbor(a).heuristic();
      }

      public static SolucionEjercicio1 getSolucion() {
            List<Integer> acciones = List2.empty();
            Ejercicio1Problem prob = Ejercicio1Problem.initial();
            Spm spm = memory.get(prob);
            while (spm != null && spm.a != null) {
                  Ejercicio1Problem old = prob;
                  acciones.add(spm.a);
                  prob = old.neighbor(spm.a);
                  spm = memory.get(prob);
            }
            return SolucionEjercicio1.of_Range(acciones);
      }

}
```

Ejercicio1Problem

```
package ejercicios.ejercicio1.manual;

import java.util.Comparator;
import java.util.List;
import java.util.Map;
import java.util.Objects;
import java.util.function.Predicate;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

import _datos.DatosEjercicio1;
import _datos.DatosEjercicio1.Tipo;
import _datos.DatosEjercicio1.Variedad;
import us.lsi.common.List2;

public record Ejercicio1Problem(
            Integer index,
            List<Integer> remaining) {

      public List<Integer> actions() {
            List<Integer> res = List2.empty();
            // Comprobar si estamos en el final.
```

```
			if(index == DatosEjercicio1.getVariedades()) {
				return List2.empty();
			}

//			Integer numero = DatosEjercicio1.getElemento(index);
//			Integer maximo_veces = (remaining / numero);

			if(this.remaining.stream().allMatch(x -> x==0)) {//si me quedo sin café
				return List2.of(0);
			}else { //en caso contrario, selecciono el número de kilos disponibles

				Variedad v = DatosEjercicio1.variedades.get(this.index); //variedad actual
				Integer disponible = cantidadDisponible(v);
				res =  IntStream.rangeClosed(0, disponible) //creo un stream de enteros desde 0 a disponible (Incluidos)
						.boxed() //devuelvo un stream a partir de esto
						.toList(); //lo paso a lista
//				System.out.println("acciones disponibles para la variedad " + v.nombre() + ": " +res);
			}
			return res;
		}


		public Ejercicio1Problem neighbor(Integer a) {
			List<Integer> remaining2 = List2.copy(remaining);
			remaining2 = actualizaRemaining(a);
			return new Ejercicio1Problem(index +1 ,remaining2);
		}


		public Boolean existeMayorMejor() {
			Variedad variedadBenMax = IntStream.range(this.index + 1, DatosEjercicio1.getVariedades())
					.boxed()
					.map(i -> DatosEjercicio1.variedades.get(i))
					.sorted(Comparator.comparingDouble(Variedad::beneficio).reversed())
					//.filter(x -> cumpleCondicion(x, this.remaining))
					.findFirst().orElse(null);
			if(variedadBenMax!=null) return resuelveProblema(variedadBenMax) && variedadBenMax.beneficio() > DatosEjercicio1.getBeneficio(index);
			else return false;
		}

		@Override
		public String toString() {
			return "Ejercicio1Problem [index=" + index + ", remaining=" + remaining + "]";
		}


		@Override
		public int hashCode() {
			return Objects.hash(index, remaining);
		}


		@Override
		public boolean equals(Object obj) {
			if (this == obj)
				return true;
			if (obj == null)
```

```
                return false;
            if (getClass() != obj.getClass())
                return false;
            Ejercicio1Problem other = (Ejercicio1Problem) obj;
            return Objects.equals(index, other.index) && Objects.equals(remaining,
other.remaining);
        }


        public static Ejercicio1Problem initial() {
            return new Ejercicio1Problem(0,
DatosEjercicio1.tipos.stream().map(Tipo::kgdisponibles).collect(Collectors.toList()));
        }

        public static Predicate<Ejercicio1Problem> goal() {
            Predicate<Ejercicio1Problem> pred = p -> p.index() ==
DatosEjercicio1.getVariedades();
            return pred;
        }

        public static Predicate<Ejercicio1Problem> goalHasSolution() {
            Predicate<Ejercicio1Problem> pred = p -> p.remaining().stream().allMatch(x -> x>=
0);
            return pred;
        }

        // Estimación optimista del futuro
        public Double heuristic() {
            Double res = 0.;
            List<Variedad> variedades = DatosEjercicio1.variedades;
            for(Variedad v : variedades) {
                Double resAux = v.beneficio()*cantidadDisponible(v);
                if(resAux>res) res = resAux;
            }
            return res;

        }


        public List<Integer> actualizaRemaining(Integer a){
            List<Integer> remaining2 = List2.copy(remaining);
            Variedad variedadActual = DatosEjercicio1.variedades.get(index);
            Map<String, Double> porcentaje = variedadActual.porcentaje();

            for (Map.Entry<String, Double> entry : porcentaje.entrySet()) {
                String tipo = entry.getKey();
                Double porcentajeCojo = entry.getValue();
                Tipo tipoNombre = DatosEjercicio1.tipos.stream().filter(x ->
x.Nombre_Tipo().equals(tipo)).findFirst().get();
                Integer i = DatosEjercicio1.tipos.indexOf(tipoNombre); //indice del tipo en
el que estoy
                Integer cantidadActual = remaining2.get(i);
                Integer cojo = (int) Math.floor(a*porcentajeCojo);//actualizo remaining
                remaining2.set(i, cantidadActual-cojo);
            }
            return remaining2;
        }

        public Boolean resuelveProblema(Variedad v) {
            Integer cantidadDisp = cantidadDisponible(v);
            List<Integer> remain = actualizaRemaining( cantidadDisp);
```

```
			return remain.stream().allMatch(x -> x == 0);
		}

		public Integer cantidadDisponible(Variedad v) {
			List<Integer> disponibles = List2.empty();   //aqui almaceno los Kg que tengo
disponibles para cada tipo
			for (Map.Entry<String, Double> entry : v.porcentaje().entrySet()) {
				String tipo = entry.getKey();
				Tipo tipoNombre = DatosEjercicio1.tipos.stream().filter(x ->
x.Nombre_Tipo().equals(tipo)).findFirst().get();
				Integer i = DatosEjercicio1.tipos.indexOf(tipoNombre); //indice del tipo en
el que estoy

				Double porcentajeCojo = entry.getValue();
				Integer disponible = 0;
				while(remaining.get(i)>=(porcentajeCojo*(disponible + 1))) {
					disponible++;
				}
				disponibles.add(disponible);    //añado la cantidad disponible para este
tipo

			}

			Integer disponible = disponibles.stream().min(Integer::compare).get(); //me quedo
con la menor de ellas... esta será la mayor cantidad

//			System.out.println("cantidad disponible de la variedad " + v.nombre() + ": "+
disponible);
			return disponible;
		}

}
```

Ejercicio2

DatosEjercicio2.java

SolucionEjercicio2.java

Ejercicio2Edge

```
package ejercicios.ejercicio2;

import _datos.DatosEjercicio2;
import us.lsi.graphs.virtual.SimpleEdgeAction;

public record Ejercicio2Edge(Ejercicio2Vertex source, Ejercicio2Vertex target, Integer action,
		Double weight) implements SimpleEdgeAction<Ejercicio2Vertex,Integer> {

	public static Ejercicio2Edge of(Ejercicio2Vertex s, Ejercicio2Vertex t, Integer a) {
		// TODO La arista debe tener peso
		return new Ejercicio2Edge(s, t, a, a *
DatosEjercicio2.getPrecioInscripcion(s.index()));
	}

	@Override
	public String toString() {
		return String.format("%d; %.1f", action, weight);
	}
}
```

Ejercicio2Heuristic

```
package ejercicios.ejercicio2;

import java.util.List;
import java.util.Set;
import java.util.function.Predicate;
import java.util.stream.IntStream;

import _datos.DatosEjercicio2;
import _datos.DatosEjercicio2.Curso;
import us.lsi.common.List2;
import us.lsi.common.Set2;

public class Ejercicio2Heuristic {

	// Se explica en practicas.
	public static Double heuristic(Ejercicio2Vertex v1, Predicate<Ejercicio2Vertex> goal,
Ejercicio2Vertex v2) {
//		//if(cubro alguna tematica)
//			//me quedo con la de menos coste
//		//else return Double.MAXVALUE
		return v1.remainingTem().isEmpty()? 0.:
			IntStream.range(v1.index(), DatosEjercicio2.getCursos())
					.filter(i -> cubre(i, v1)!=0)
					.mapToDouble(i -> DatosEjercicio2.getPrecioInscripcion(i))
					.min()
					.orElse(Double.MAX_VALUE);
	}

	public static Integer cubre(Integer i, Ejercicio2Vertex v1) {
		Curso cursoActual = DatosEjercicio2.cursos.get(i);
		Set<Integer> remainingTem2 =
Set2.difference(v1.remainingTem(),cursoActual.tematicas()); //tematicas actualizadas
		if(remainingTem2.equals(v1.remainingTem())) return 0; //si no cubro ninguna
devuelvo 0
		else if(remainingTem2.isEmpty()) return 1; //si cubro todas devuelvo 1
		else return 2; //si cubro pero no son todas, devuelvo 2
	}

}
```

Ejercicio2Vertex

```
package ejercicios.ejercicio2;


import java.util.List;
import java.util.Set;
import java.util.function.Predicate;

import _datos.DatosEjercicio2;
import _datos.DatosEjercicio2.Curso;
import us.lsi.common.List2;
import us.lsi.common.Set2;
import us.lsi.graphs.virtual.VirtualVertex;

public record Ejercicio2Vertex(Integer index, Set<Integer> remainingTem, Set<Integer>
selectedCent)
    implements VirtualVertex<Ejercicio2Vertex, Ejercicio2Edge, Integer>{

      public static Ejercicio2Vertex of(Integer i, Set<Integer> rest, Set<Integer> cent) {
            return new Ejercicio2Vertex(i, rest,cent);
      }

      // TODO Consulte las clases GraphsPI5 y TestPI5

      @Override
      public List<Integer> actions() {
            // TODO Alternativas de un vertice
            if(goal().test(this)) { //si llego al final devuelvo una lista vacia
                  return List2.empty();
            }else if(this.remainingTem().isEmpty()) { //si ya no quedan temáticas por recubrir
no hago nada
                  return List2.of(0);
            }else {
                  Curso cursoActual = DatosEjercicio2.cursos.get(this.index);
                  if(this.index == DatosEjercicio2.getCursos()-1) { //si estoy en el último
                        return noSuperaCentros(cursoActual) && cubre()==1? List2.of(1): //si
termino el problema devuelvo 1

               List2.of(0); //si voy a dar una solucion incorrecta o no lo termino no hago
nada
                  }else if(cubre()==0) { //si no cubro ninguna tematica, no hago nada
                        return List2.of(0);
                  }else { //si aporto algo, pero no lo termino
                        if(noSuperaCentros(cursoActual)){ //si la solucion cumple las
restricciones
                              return List2.of(0,1);
                        }else {      //si la solucion NO cumple las restricciones
                              return List2.of(0);
                        }
                  }
            }
      }

      @Override
      public Ejercicio2Vertex neighbor(Integer a) {
            // TODO Vertice siguiente al actual segun la alternativa a
            Set<Integer> remainingTem2 = Set2.copy(this.remainingTem);
            Set<Integer> selectedCent2 = Set2.copy(this.selectedCent);
            Set<Integer> tematicasActuales =
DatosEjercicio2.cursos.get(this.index).tematicas();
```

```
			if(a!=0) {
				remainingTem2 = Set2.difference(this.remainingTem,tematicasActuales);
//remaining.removeAll(tematicas curso actual)
				selectedCent2.add(DatosEjercicio2.cursos.get(this.index).centro());
//añadir centro actual
			}
			return of(index + 1, remainingTem2, selectedCent2);
		}

		@Override
		public Ejercicio2Edge edge(Integer a) {
			return Ejercicio2Edge.of(this, neighbor(a), a);
		}

		// Se explica en practicas.
		public Ejercicio2Edge greedyEdge() {
			Curso cursoActual = DatosEjercicio2.cursos.get(this.index);
			if(noSuperaCentros(cursoActual)) return cubre()==0? edge(0): edge(1);
			else return edge(0);
		}

		@Override
		public String toString() {
			return "Ejercicio2Vertex [index=" + index + ", remainingTem=" + remainingTem + ",
selectedCent=" + selectedCent
					+ "]";
		}

		public static Ejercicio2Vertex initial() {
			// TODO Auto-generated method stub
			return new Ejercicio2Vertex(0, Set2.of(DatosEjercicio2.getListaTematicas()),
Set2.empty());
		}

		public static Predicate<Ejercicio2Vertex> goal() {
			// TODO Auto-generated method stub
			Predicate<Ejercicio2Vertex> pred =  p -> p.index() == DatosEjercicio2.getCursos();
			return pred;
		}

		public static Predicate<Ejercicio2Vertex> goalHasSolution() {
			// TODO Auto-generated method stub
			Predicate<Ejercicio2Vertex> pred =  p -> p.remainingTem().isEmpty() &&
p.selectedCent().size()<=DatosEjercicio2.maxCentros; //tiene solucion si no quedan temáticas
por seleccionar y no se supera el máximo de centros
			return pred;
		}

		public Boolean noSuperaCentros(Curso cursoActual) {
			Boolean res1 = this.selectedCent.contains(cursoActual.centro());
			Boolean res2 = this.selectedCent.size() < DatosEjercicio2.maxCentros;
			return res1 || res2;
		}



		public Integer cubre() {
			Curso cursoActual = DatosEjercicio2.cursos.get(this.index);
			Set<Integer> remainingTem2 =
Set2.difference(this.remainingTem,cursoActual.tematicas()); //tematicas actualizadas
```

```
            if(remainingTem2.equals(this.remainingTem)) return 0; //si no cubro ninguna
devuelvo 0
            else if(remainingTem2.isEmpty()) return 1; //si cubro todas devuelvo 1
            else return 2; //si cubro pero no son todas, devuelvo 2
      }
}
```

Manual

Ejercicio2PDR

```
package ejercicios.ejercicio2.manual;

import java.util.Comparator;
import java.util.List;
import java.util.Map;

import _datos.DatosEjercicio2;
import _soluciones.SolucionEjercicio2;
import us.lsi.common.List2;
import us.lsi.common.Map2;

public class Ejercicio2PDR {

      public static record Spm(Integer a, Integer weight) implements Comparable<Spm> {
            public static Spm of(Integer a, Integer weight) {
                  return new Spm(a, weight);
            }
            @Override
            public int compareTo(Spm sp) {
                  return this.weight.compareTo(sp.weight);
            }
      }

      public static Map<Ejercicio2Problem, Spm> memory;
      public static Integer mejorValor;

      public static SolucionEjercicio2 search() {
            memory =  Map2.empty();
            mejorValor = Integer.MAX_VALUE; // Estamos minimizando

            pdr_search(Ejercicio2Problem.initial(), 0, memory);
            return getSolucion();
      }

      private static Spm pdr_search(Ejercicio2Problem prob, Integer acumulado,
Map<Ejercicio2Problem, Spm> memoria) {

            Spm res = null;
            Boolean esTerminal = prob.index().equals(DatosEjercicio2.getCursos());
            Boolean esSolucion = prob.remainingTem().isEmpty() &&
prob.selectedCent().size()<=DatosEjercicio2.maxCentros;

            if (memory.containsKey(prob)) {
                  res = memory.get(prob);

            } else if (esTerminal && esSolucion) {
                  res = Spm.of(null, 0);
                  memory.put(prob, res);
                  if (acumulado < mejorValor) { // Estamos minimizando
                        mejorValor = acumulado;
```

```
                }
            } else {
                List<Spm> soluciones = List2.empty();
                for (Integer action : prob.actions()) {
                    Double cota = acotar(acumulado, prob, action);
                    if (cota > mejorValor) {
                        continue;
                    }
                    Ejercicio2Problem vecino = prob.neighbor(action);
                    Spm s = pdr_search(vecino, acumulado + action, memory);
                    if (s != null) {
                        Spm amp = Spm.of(action, s.weight() + action);
                        soluciones.add(amp);
                    }
                }
                // Estamos minimizando
                res = soluciones.stream().min(Comparator.naturalOrder()).orElse(null);
                if (res != null)
                    memory.put(prob, res);
            }

            return res;
        }

        private static Double acotar(Integer acum, Ejercicio2Problem p, Integer a) {
            return acum + a + p.neighbor(a).heuristic();
        }

        public static SolucionEjercicio2 getSolucion() {
            List<Integer> acciones = List2.empty();
            Ejercicio2Problem prob = Ejercicio2Problem.initial();
            Spm spm = memory.get(prob);
            while (spm != null && spm.a != null) {
                Ejercicio2Problem old = prob;
                acciones.add(spm.a);
                prob = old.neighbor(spm.a);
                spm = memory.get(prob);
            }
            return SolucionEjercicio2.of_Range(acciones);
        }

}
```

Ejercicio2Problem

```
package ejercicios.ejercicio2.manual;

import java.util.Comparator;
import java.util.List;
import java.util.Map;
import java.util.Objects;
import java.util.Set;
import java.util.function.Predicate;
import java.util.stream.Collectors;
import java.util.stream.IntStream;


import _datos.DatosEjercicio2;
import _datos.DatosEjercicio2.Curso;
import ejercicios.ejercicio2.Ejercicio2Heuristic;
```

```
import ejercicios.ejercicio2.Ejercicio2Vertex;
import us.lsi.common.List2;
import us.lsi.common.Set2;

public record Ejercicio2Problem(
		Integer index,
		Set<Integer> remainingTem, Set<Integer> selectedCent) {

	public List<Integer> actions() {
		if(goal().test(this)) { //si llego al final devuelvo una lista vacia
			return List2.empty();
		}else if(remainingTem.isEmpty()) { //si ya no quedan temáticas por recubrir no
hago nada
			return List2.of(0);
		}else {
			Curso cursoActual = DatosEjercicio2.cursos.get(index);
			if(index == DatosEjercicio2.getCursos()-1) { //si estoy en el último
				return noSuperaCentros(cursoActual) && cubre()==1? List2.of(1): //si
termino el problema devuelvo 1

			   List2.of(0); //si voy a dar una solucion incorrecta o no lo termino no hago
nada
			}else if(cubre()==0) { //si no cubro ninguna tematica, no hago nada
				return List2.of(0);
			}else { //si aporto algo, pero no lo termino
				if(noSuperaCentros(cursoActual)){ //si la solucion cumple las
restricciones
					return List2.of(0,1);
				}else {      //si la solucion NO cumple las restricciones
					return List2.of(0);
				}
			}
		}
	}


	public Ejercicio2Problem neighbor(Integer a) {
		Set<Integer> remainingTem2 = Set2.copy(this.remainingTem);
		Set<Integer> selectedCent2 = Set2.copy(this.selectedCent);
		Set<Integer> tematicasActuales =
DatosEjercicio2.cursos.get(this.index).tematicas();
		if(a!=0) {
			remainingTem2 = Set2.difference(this.remainingTem,tematicasActuales);
//remaining.removeAll(tematicas curso actual)
			selectedCent2.add(DatosEjercicio2.cursos.get(this.index).centro());
//añadir centro actual
		}
		return new Ejercicio2Problem(index + 1, remainingTem2, selectedCent2);
	}


	@Override
	public String toString() {
		return "Ejercicio2Problem [index=" + index + ", remainingTem=" + remainingTem + ",
selectedCent=" + selectedCent
				+ "]";
	}


	@Override
	public int hashCode() {
```

```
		return Objects.hash(index, remainingTem, selectedCent);
	}


	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Ejercicio2Problem other = (Ejercicio2Problem) obj;
		return Objects.equals(index, other.index) && Objects.equals(remainingTem,
other.remainingTem)
				&& Objects.equals(selectedCent, other.selectedCent);
	}


	public static Ejercicio2Problem initial() {
		return new Ejercicio2Problem(0, Set2.of(DatosEjercicio2.getListaTematicas()),
Set2.empty());
	}

	public static Predicate<Ejercicio2Problem> goal() {
		Predicate<Ejercicio2Problem> pred = p -> p.index() == DatosEjercicio2.getCursos();
		return pred;
	}

	public static Predicate<Ejercicio2Problem> goalHasSolution() {
		Predicate<Ejercicio2Problem> pred = p -> p.remainingTem().isEmpty() &&
p.selectedCent().size()<=DatosEjercicio2.maxCentros;
		return pred;
	}

	// Estimación optimista del futuro
	public Double heuristic() {
		return remainingTem.isEmpty()? 0.:
			IntStream.range(index, DatosEjercicio2.getCursos())
					.filter(i -> cubre(i)!=0)
					.mapToDouble(i -> DatosEjercicio2.getPrecioInscripcion(i))
					.min()
					.orElse(Double.MAX_VALUE);
	}


	public Boolean noSuperaCentros(Curso cursoActual) {
		Boolean res1 = selectedCent.contains(cursoActual.centro());
		Boolean res2 = selectedCent.size()+1 <= DatosEjercicio2.maxCentros;
		return res1 || res2;
	}

	public Integer cubre() {
		Curso cursoActual = DatosEjercicio2.cursos.get(index);
		Set<Integer> remainingTem2 =
Set2.difference(remainingTem,cursoActual.tematicas()); //tematicas actualizadas
		if(remainingTem2.equals(remainingTem)) return 0; //si no cubro ninguna devuelvo 0
		else if(remainingTem2.isEmpty()) return 1; //si cubro todas devuelvo 1
		else return 2; //si cubro pero no son todas, devuelvo 2
	}
```

```
	public Integer cubre(Integer i) {
		Curso cursoActual = DatosEjercicio2.cursos.get(i);
		Set<Integer> remainingTem2 =
Set2.difference(remainingTem,cursoActual.tematicas()); //tematicas actualizadas
		if(remainingTem2.equals(remainingTem)) return 0; //si no cubro ninguna devuelvo 0
		else if(remainingTem2.isEmpty()) return 1; //si cubro todas devuelvo 1
		else return 2; //si cubro pero no son todas, devuelvo 2
	}
}
```

Ejercicio3

DatosEjercicio3.java

```
package _datos;

import java.util.Arrays;
import java.util.Comparator;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.stream.Collectors;

import us.lsi.common.Files2;

public class DatosEjercicio3 {

	public static List<Investigador> investigadores;
	public static record Investigador(String nombre, Integer capacidad, Integer
especialidad) {
		public Investigador of(String nombre, Integer capacidad, Integer especialidad) {
			return new Investigador(nombre, capacidad, especialidad);
		}

		public static Investigador ofFormat(String linea) {
			String[] formato = linea.split(":");
			String nombre = formato[0];
			String segundaParte = formato[1].trim();
			String[] segundaPartePars = segundaParte.split(";");
			Integer capacidad =
Integer.parseInt(segundaPartePars[0].split("=")[1].trim());
			Integer especialidad =
Integer.parseInt(segundaPartePars[1].split("=")[1].trim());
			return new Investigador(nombre,capacidad, especialidad);
		}

	}

	public static List<Trabajo> trabajos;
	public static record Trabajo(String nombre, Integer calidad, Map<Integer, Integer>
reparto) {
		public Trabajo of(String nombre, Integer calidad, Map<Integer, Integer> reparto) {
			return new Trabajo(nombre, calidad, reparto);
		}

		public static Trabajo ofFormat(String linea) {
			String[] formato = linea.split("->");
			String nombre = formato[0].trim();
			String segundaParte = formato[1].trim();
			String[] segundaPartePars = segundaParte.split(";");
			Integer calidad =
Integer.parseInt(segundaPartePars[0].split("=")[1].trim());
			Map<Integer, Integer> reparto =
Arrays.stream(segundaPartePars[1].split("=")[1].split(","))
					.map(pair -> pair.split(":"))
					.collect(Collectors.toMap(
							keyValue -> Integer.parseInt(keyValue[0].replace("(",
"").replace(")", "")),
							keyValue -> Integer.parseInt(keyValue[1].replace("(",
"").replace(")", ""))
					));
```

```
                return new Trabajo(nombre, calidad, reparto);
            }
        }


        public static void iniDatos(String fichero) {
            List<String> lineas = Files2.linesFromFile(fichero);
            investigadores = lineas.stream().filter(l -> l.startsWith("I")).map(x ->
Investigador.ofFormat(x)).toList();
            trabajos = lineas.stream().filter(l -> l.startsWith("T")).map(x ->
Trabajo.ofFormat(x)).toList();
            toConsole();
        }

        //.....................................................
        //head section del .lsi

        public static Integer getTrabajos() {
            return trabajos.size();
        }

        public static Integer getInvestigadores() {
            return investigadores.size();
        }

        public static Integer getCapacidad(Integer i) {
            return investigadores.get(i).capacidad;
        }
        public static Integer getCalidad(Integer i) {
            return trabajos.get(i).calidad;
        }

        public static Integer getMaximoDias() {
            return investigadores.stream().map(Investigador::capacidad).reduce((a, b) ->
a>b?a:b).get();
        }

//      public static Integer compruebaTrabajo(Integer i, Integer k) {
//          Integer checkInv = investigadores.get(i).especialidad();
//          Set<Integer> checkTra = trabajos.get(k).reparto().keySet();
//          return checkTra.contains(checkInv)?1:0;
//      }

        public static Integer diasNecesito(Integer j, Integer k) {
            return trabajos.get(j).reparto().get(k);
        }

        public static Integer getEspecialidades() {
            return
investigadores.stream().map(Investigador::especialidad).distinct().collect(Collectors.toList())
.size();
        }
        public static Integer totalTrabajo(Integer j) {
            return
trabajos.get(j).reparto().values().stream().mapToInt(Integer::intValue).sum();
        }

        public static Integer seleccionaEspecialidad(Integer i, Integer k) {
            return investigadores.get(i).especialidad().equals(k)?1:0;
        }
        //.....................................................
```

```
	public static List<Investigador> getListaInvestigadores(){
		return investigadores;
	}
	public static List<Trabajo> getListaTrabajos(){
		return trabajos;
	}

	public static void toConsole() {
		//String2.toConsole("Conjunto de Entrada Tipos: %s\nConjunto de Entrada
Variedades: %d", tipos, variedades);
		System.out.println(investigadores);
		System.out.println(trabajos);
	}

	// Test de la lectura del fichero
	public static void main(String[] args) {
		iniDatos("ficheros/Ejercicio3DatosEntrada1.txt");
	}
}
```

SolucionEjercicio3.java

```
package _soluciones;

import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

import org.jgrapht.GraphPath;

import _datos.DatosEjercicio3;
import _datos.DatosEjercicio3.Investigador;
import _datos.DatosEjercicio3.Trabajo;
import ejercicios.ejercicio3.Ejercicio3Edge;
import ejercicios.ejercicio3.Ejercicio3Vertex;
import us.lsi.common.List2;
import us.lsi.common.Map2;

public class SolucionEjercicio3 implements Comparable<SolucionEjercicio3>{

	public static SolucionEjercicio3 of_Range(List<Integer> ls) {
		return new SolucionEjercicio3(ls);
	}
	// Ahora en la PI5
		public static SolucionEjercicio3 of(GraphPath<Ejercicio3Vertex, Ejercicio3Edge>
path) {
			List<Integer> ls = path.getEdgeList().stream().map(e ->
e.action()).toList();
			SolucionEjercicio3 res = of_Range(ls);
			res.path = ls;
			return res;
		}

		// Ahora en la PI5
		private List<Integer> path;


	private Integer calidad;
	private List<Investigador> investigadores;
	private List<List<Integer>> solucion;

	//List(0,...,n-1,  n,...2n-1,  ...)
	//     i0,...,in,  i0,...,in,  ...
	//       j0              j1         ...

	private SolucionEjercicio3() {
		calidad = 0;
		investigadores = List2.empty();
		solucion = List2.empty();
	}
	private SolucionEjercicio3(List<Integer> ls) {
		Integer numTrabajos = DatosEjercicio3.getTrabajos();
		Integer numInvestigadores = DatosEjercicio3.getInvestigadores();
		Integer nxm = numTrabajos * numInvestigadores;
		List<List<Integer>> listasDivididas = List2.empty();
		List<List<Integer>> listaReparto = List2.empty();
		Map<Integer, List<Integer>> espeInv = Map2.empty();
		for(int i = 0; i < DatosEjercicio3.getEspecialidades();i++) {
			List<Integer> listaN = List2.empty();
			for(int j=0;j < listasDivididas.size();j++) {
```

```
                        if(DatosEjercicio3.investigadores.get(j).especialidad()==i) {
                              listaN.add(j);
                        }
                  }
                  espeInv.put(i, listaN);
            }

            // Divide la lista en m listas de n elementos cada una
            for (int i = 0; i < numInvestigadores; i++) {
                  int desde = i * numTrabajos;
                  int hasta = (i + 1) * numTrabajos;
                  List<Integer> sublista = ls.subList(desde, hasta);
                  listasDivididas.add(sublista);
            }
            for(Trabajo t:DatosEjercicio3.trabajos) {
                  listaReparto.addAll(List2.of(t.reparto().values().stream().toList()));
            }

            Integer m = DatosEjercicio3.getTrabajos();
            for(int i = 0; i < nxm;i++) {
                  Integer especialidadActual =
DatosEjercicio3.investigadores.get(i/m).especialidad();
                  Integer trabajoActual = i%m;
                  Integer nuevaCap = listaReparto.get(trabajoActual).get(especialidadActual)
- ls.get(i);
                  List<Integer> lista_aux = List2.setElement(listaReparto.get(trabajoActual),
especialidadActual, nuevaCap);
                  listaReparto.set(trabajoActual, lista_aux);
            }
            solucion = listaReparto;
            calidad = IntStream.range(0, listaReparto.size()).boxed().filter(x ->
listaReparto.get(x).stream().allMatch(y -> y == 0)).map(x ->
DatosEjercicio3.trabajos.get(x).calidad()).reduce((a,b) -> a =a+b).orElse(0);
            investigadores = DatosEjercicio3.investigadores;

      }

      public static SolucionEjercicio3 empty() {
            return new SolucionEjercicio3();
      }


      @Override
      public String toString() {
//            System.out.println("reparto de horas: " + solucion);
            String s = investigadores.stream()
                    .map(i -> "INVESTIGADOR " + i.nombre() + ": " + i)
                    .collect(Collectors.joining("\n", "Reparto de horas:\n", "\n"));
            return String.format("%sSuma de las calidades de los trabajos realizados: %d", s,
calidad);
      }
      @Override
      public int compareTo(SolucionEjercicio3 o) {
            // TODO Auto-generated method stub
            return 0;
      }
}
```

Ejercicio3Edge

```
package ejercicios.ejercicio3;


import _datos.DatosEjercicio3;
import us.lsi.graphs.virtual.SimpleEdgeAction;

public record Ejercicio3Edge(Ejercicio3Vertex source, Ejercicio3Vertex target, Integer action,
Double weight)
		implements SimpleEdgeAction<Ejercicio3Vertex,Integer> {

	public static Ejercicio3Edge of(Ejercicio3Vertex v1, Ejercicio3Vertex v2, Integer a) {

		// TODO La arista debe tener peso
		Integer m = DatosEjercicio3.getTrabajos();
		Integer indiceTrabajoActual = v1.index()%m;
		Integer indiceInvActual = v1.index()/m;
		return new Ejercicio3Edge(v1, v2, a, a *
DatosEjercicio3.getCalidad(indiceTrabajoActual) * 1.0);
	}
	@Override
	public String toString() {
		return String.format("%d; %.1f", action, weight);
	}
}
```

Ejercicio3Heuristic

```
package ejercicios.ejercicio3;

import java.util.Iterator;
import java.util.List;
import java.util.function.Predicate;
import java.util.stream.IntStream;

import _datos.DatosEjercicio3;
import us.lsi.common.List2;

public class Ejercicio3Heuristic {

	// Se explica en practicas.
	public static Double heuristic(Ejercicio3Vertex v1, Predicate<Ejercicio3Vertex> goal,
Ejercicio3Vertex v2) {
		Double res = 0.;
		List<Integer> remainingHoras = v1.remainingInv(); //lista de horas disponibles
para cada trabajador
		List<Integer> remainingEspecialidades = List2.ofTam(0,
DatosEjercicio3.getEspecialidades()); //lista de horas disponibles para cada especialidad
		List<List<Integer>> remainingTotal = v1.remainingEsp(); //Lista de lo que falta
para completar cada trabajo
		for(int i=0; i < remainingHoras.size();i++) {
			Integer especialidad =
DatosEjercicio3.investigadores.get(i).especialidad();
			Integer horasAhora = remainingEspecialidades.get(especialidad) +
remainingHoras.get(i);
			remainingEspecialidades.set(especialidad, horasAhora ); //actualizo el
tiempo disponible para cada especialidad
		}

		for(List<Integer> l : remainingTotal) {
			if(termina(remainingEspecialidades, l)) {
				res +=
DatosEjercicio3.trabajos.get(remainingTotal.indexOf(l)).calidad();
			}
		}
		return res;

		//compruebo cuantos trabajos puedo terminar (con remainingInv y remainingEsp) sin
restar las horas que trabajan y sumo la calidad de los trabajos completables


	}

	public static Boolean termina(List<Integer> rE,List<Integer> l ) {
		Boolean res = true;
		for(int i = 0; i < rE.size(); i++) {
			if(l.get(i)-rE.get(i) > 0) res = false;
		}
		return res;
	}

}
```

Ejercicio3Vertex

```
package ejercicios.ejercicio3;

import java.util.Comparator;
import java.util.List;
import java.util.function.Predicate;
import java.util.stream.IntStream;

import _datos.DatosEjercicio3;
import _datos.DatosEjercicio3.Investigador;
import _datos.DatosEjercicio3.Trabajo;
import us.lsi.common.List2;
import us.lsi.graphs.virtual.VirtualVertex;

// Uso el segundo modelo
public record Ejercicio3Vertex(Integer index, List<Integer> remainingInv, List<List<Integer>>
remainingEsp)
     implements VirtualVertex<Ejercicio3Vertex,Ejercicio3Edge,Integer> {

      public static Ejercicio3Vertex of(Integer i, List<Integer> rest, List<List<Integer>>
esp) {
            return new Ejercicio3Vertex(i, rest, esp);
      }

      @Override
      public List<Integer> actions() {
            // TODO Alternativas de un vertice
            List<Integer> res = List2.empty();
            Integer m = DatosEjercicio3.getTrabajos();
            Integer n = DatosEjercicio3.getInvestigadores();
            Integer indiceTrabajoActual = indiceTrabajoActual();
            Integer indiceInvActual = indiceInvActual();

            if(this.index==(n * m)) { //si estoy en el último vértice
                  return res;
            }else if(this.remainingInv.get(indiceInvActual) == 0) { //si no tengo más días
disponibles
                  res.add(0);
                  return res;
            }else {
                  //cuántos movimientos puedo hacer? Dependo de la capacidad que le qude al
investigador actual y de
                  //los días que falten para cubrir la especialidad que trata ese
investigador
                  res.add(0);
                  Investigador investigadorActual =
DatosEjercicio3.investigadores.get(indiceInvActual);
                  Integer especialidadActual = investigadorActual.especialidad();

                  List<Integer> remainingEspecialidades = disponibleEspecialidad();

                  List<Integer> listaTrabajos = this.remainingEsp.get(indiceTrabajoActual);

                  Integer capacidadDisp = this.remainingInv.get(indiceInvActual);
                  Integer capacidadDisp2 = remainingEspecialidades.get(especialidadActual);
                  Integer capacidadNecesito = listaTrabajos.get(especialidadActual);
                  Integer resultado = Math.min(Math.min(capacidadDisp, capacidadDisp2),
capacidadNecesito);
                  List<Integer> listaRes = IntStream.rangeClosed(1,
resultado).boxed().toList();
```

```
                        res.addAll(listaRes);
                }

                return res;
        }

        @Override
        public Ejercicio3Vertex neighbor(Integer a) {
                // TODO Vertice siguiente al actual segun la alternativa a
                List<Integer> remainingInv2 = List2.copy(remainingInv);
                List<List<Integer>> remainingEsp2 = List2.copy(remainingEsp);

                Integer indiceTrabajoActual = indiceTrabajoActual();
                Integer indiceInvActual = indiceInvActual();
                Investigador investigadorActual =
DatosEjercicio3.investigadores.get(indiceInvActual);

                //cuando creo un vecino tengo que actualizar la lista de días disponibles de los
investigadores
                Integer capacidadInvestigador = this.remainingInv.get(indiceInvActual);
                remainingInv2.set(indiceInvActual, capacidadInvestigador-a);
                //cuando creo un vecino tengo que actualizar los días que quedan para completar el
trabajo
                Integer diasQuedan =
remainingEsp.get(indiceTrabajoActual).get(investigadorActual.especialidad());

                List<Integer> lista_aux = List2.setElement(remainingEsp2.get(indiceTrabajoActual),
investigadorActual.especialidad(), diasQuedan - a);
                remainingEsp2.set(indiceTrabajoActual, lista_aux);

                return new Ejercicio3Vertex(index + 1, remainingInv2, remainingEsp2);
        }

        @Override
        public Ejercicio3Edge edge(Integer a) {
                return Ejercicio3Edge.of(this,this.neighbor(a),a);
        }

        // Se explica en practicas.
        public Ejercicio3Edge greedyEdge() {
                Investigador investigadorActual =
DatosEjercicio3.investigadores.get(indiceInvActual());
                List<Integer> remainingEspecialidades = disponibleEspecialidad();
                Integer especialidadActual = investigadorActual.especialidad();
                List<Integer> listaTrabajos = this.remainingEsp.get(indiceTrabajoActual());

                Integer capacidadDisp = this.remainingInv.get(indiceInvActual());
                Integer capacidadDisp2 = remainingEspecialidades.get(especialidadActual);
                Integer capacidadNecesito = listaTrabajos.get(especialidadActual);
                Integer resultado = Math.min(Math.min(capacidadDisp, capacidadDisp2),
capacidadNecesito);

                if(termina(disponibleEspecialidad(), indiceTrabajoActual())) {
                        return edge(resultado);
                }else {
                        Trabajo trabajoTerminaMasCalidad = IntStream.range(indiceInvActual()+1,
DatosEjercicio3.getTrabajos())
                                        .boxed()
                                        .filter(x -> termina(disponibleEspecialidad(), x))
                                        .map(i -> DatosEjercicio3.trabajos.get(i))
```

```
                                .max(Comparator.comparingInt(Trabajo::calidad)).orElse(null);

                        if(trabajoTerminaMasCalidad==null) return edge(resultado);
                        else return edge(0);
                }
        }


        public static Ejercicio3Vertex initial() {
                // TODO Auto-generated method stub
                List<Integer> listaCapacidades =
DatosEjercicio3.investigadores.stream().map(Investigador::capacidad).toList();
                List<List<Integer>> listaEspecialidades =
DatosEjercicio3.trabajos.stream().map(Trabajo::reparto).map(x ->
x.values().stream().toList()).toList();
                return new Ejercicio3Vertex(0,listaCapacidades, listaEspecialidades);
        }

        public static Predicate<Ejercicio3Vertex> goal() {
                // TODO Auto-generated method stub
                Predicate<Ejercicio3Vertex> pred =  p -> p.index() ==
DatosEjercicio3.getTrabajos() * DatosEjercicio3.getInvestigadores(); //n * m
                return pred;
        }

        public static Predicate<Ejercicio3Vertex> goalHasSolution() {
                // TODO Auto-generated method stub
                Predicate<Ejercicio3Vertex> pred =  p -> p.remainingInv.stream().allMatch(x ->
x>=0);
                return pred;
        }

        public Integer indiceTrabajoActual() {
                Integer m = DatosEjercicio3.getTrabajos();
                return this.index%m;
        }

        public Integer indiceInvActual() {
                Integer m = DatosEjercicio3.getTrabajos();
                return this.index/m;
        }


        public Boolean termina(List<Integer> rE, Integer t) {
                Boolean res = true;
                List<Integer> remTrabajoI = this.remainingEsp.get(t); //lista de horas que faltan
para terminar el trabajo actual

                for(int i = 0; i < rE.size(); i++) {
                        if(remTrabajoI.get(i)-rE.get(i) > 0) res = false; //si le resto lo que me
queda disponible y no lo acabo, no se termina
                }
                return res;
        }

        public List<Integer> disponibleEspecialidad(){
                List<Integer> remainingEspecialidades = List2.ofTam(0,
DatosEjercicio3.getEspecialidades()); //lista de horas disponibles para cada especialidad
                for(int i=0; i < this.remainingInv.size();i++) {
                        Integer especialidad =
DatosEjercicio3.investigadores.get(i).especialidad();
```

```
                  Integer horasAhora = remainingEspecialidades.get(especialidad) +
this.remainingInv.get(i);
                  remainingEspecialidades.set(especialidad, horasAhora );//actualizo el
tiempo disponible para cada especialidad
            }
            return remainingEspecialidades;
      }

      public Integer maximoCojo(Integer indiceTrabajoActual) {
            Integer res = 0;
            List<Integer> remainingTrabajo = this.remainingEsp.get(indiceTrabajoActual);
            List<Integer> remainingEsp = disponibleEspecialidad();
            for(int i = 0; i < remainingEsp.size(); i++) {
                  Integer rexAux = remainingEsp.get(i)-remainingTrabajo.get(i);
                  if(rexAux<res) res = rexAux;
            }

            return res;
      }


}
```

Manual

Ejercicio3BT

```
package ejercicios.ejercicio3.manual;

import java.util.Set;

import _soluciones.SolucionEjercicio3;
import us.lsi.common.Set2;

public class Ejercicio3BT {

      private static Double mejorValor;
      private static Ejercicio3State estado;
      private static Set<SolucionEjercicio3> soluciones;

      public static void search() {
            soluciones = Set2.newTreeSet();
            mejorValor = Double.MIN_VALUE; // Estamos maximizando
            estado = Ejercicio3State.initial();
            bt_search();
      }

      private static void bt_search() {
            if (estado.esSolucion()) {
//                System.out.println("1");
                  Double valorObtenido = estado.acumulado;
                  if (valorObtenido > mejorValor) {  // Estamos maximizando
//                      System.out.println("2");
                        mejorValor = valorObtenido;
                        soluciones.add(estado.getSolucion());
                  }
            } else if(!estado.esTerminal()){
//                System.out.println("3");
                  for (Integer a: estado.alternativas()) {
                        if (estado.cota(a) >= mejorValor) {  // Estamos maximizando
//                            System.out.println("4");
```

```
                                estado.forward(a);
                                bt_search();
                                estado.back();
                          }
                    }
              }
        }

        public static Set<SolucionEjercicio3> getSoluciones() {
              return soluciones;
        }

}
```

Ejercicio3State

```
package ejercicios.ejercicio3.manual;

import java.util.List;

import _datos.DatosEjercicio3;
import _datos.DatosEjercicio3.Trabajo;
import _soluciones.SolucionEjercicio3;
import us.lsi.common.List2;

public class Ejercicio3State {

        Ejercicio3Problem actual;
        Double acumulado;
        List<Integer> acciones;
        List<Ejercicio3Problem> anteriores;

        private Ejercicio3State(Ejercicio3Problem p, Double a,
              List<Integer> ls1, List<Ejercicio3Problem> ls2) {
              // TODO Inicializar las propiedades individuales
              actual = p;
              acumulado = a;
              acciones = ls1;
              anteriores =  ls2;
        }

        public static Ejercicio3State initial() {
              // TODO Crear el estado inicial
              return new Ejercicio3State(Ejercicio3Problem.initial(),
0.,List2.empty(),List2.empty());
        }

        public static Ejercicio3State of(Ejercicio3Problem prob, Double acum, List<Integer> lsa,
                    List<Ejercicio3Problem> lsp) {
              return new Ejercicio3State(prob, acum, lsa, lsp);
        }

        public void forward(Integer a) {
              // TODO Avanzar un estado segun la alternativa a
              acciones.add(a);
              anteriores.add(actual);
              acumulado = actual.calidadActual() * 1.0;
              actual = actual.neighbor(a);
        }
```

```
	public void back() {
		// TODO Retroceder al estado anterior
		Ejercicio3Problem prob_anterior = anteriores.get(anteriores.size()-1);
		Integer accion_anterior = acciones.get(anteriores.size()-1);

		actual = prob_anterior;
		acumulado = prob_anterior.calidadActual() * 1.0;
		acciones.remove(accion_anterior);
		anteriores.remove(prob_anterior);
	}

	public List<Integer> alternativas() {
		// TODO Alternativas segun el actual
		return this.actual.actions();
	}

	public Double cota(Integer a) {
		// TODO Cota = acumulado + func(a, actual) + h(vecino(actual, a))
		Integer weight =
DatosEjercicio3.trabajos.stream().map(Trabajo::calidad).max(Integer::compareTo).get();
//		System.out.println("dqdfqwdfwed" +weight);
		return this.acumulado + weight + actual.neighbor(a).heuristic();
	}

	public Boolean esSolucion() {
		// TODO Cuando todos los elementos del universo se han cubierto
		return actual.remainingInv().stream().allMatch(x -> x>=0) && actual.index() ==
DatosEjercicio3.getTrabajos() * DatosEjercicio3.getInvestigadores();
	}

	public Boolean esTerminal() {
		// TODO Cuando se han recorrido todos los Ejercicio3
		return actual.index() == DatosEjercicio3.getTrabajos() *
DatosEjercicio3.getInvestigadores();
	}

	public SolucionEjercicio3 getSolucion() {
		// TODO Aprovechamos lo hecho en la PI4
		return SolucionEjercicio3.of_Range(acciones);
	}

}
```

Ejercicio3Problem

```
package ejercicios.ejercicio3.manual;


import java.util.List;
import java.util.Objects;
import java.util.function.Predicate;
import java.util.stream.IntStream;

import _datos.DatosEjercicio3;
import _datos.DatosEjercicio3.Investigador;
import _datos.DatosEjercicio3.Trabajo;
import us.lsi.common.List2;

public record Ejercicio3Problem(Integer index, List<Integer> remainingInv, List<List<Integer>>
remainingEsp)
```

```
{

	public static Ejercicio3Problem of(Integer i, List<Integer> rest, List<List<Integer>>
esp) {
		return new Ejercicio3Problem(i, rest, esp);
	}

	// TODO Consulte las clases GraphsPI5 y TestPI5
	public static Predicate<Ejercicio3Problem> goal(){
		return p -> p.index() == DatosEjercicio3.getTrabajos() *
DatosEjercicio3.getInvestigadores();
	}

	public static Predicate<Ejercicio3Problem> goalHasSolution(){
		return p -> p.remainingInv.stream().allMatch(x -> x>=0);
	}

	public static Ejercicio3Problem initial() {
		List<Integer> listaCapacidades =
DatosEjercicio3.investigadores.stream().map(Investigador::capacidad).toList();
		List<List<Integer>> listaEspecialidades =
DatosEjercicio3.trabajos.stream().map(Trabajo::reparto).map(x ->
x.values().stream().toList()).toList();
		return of(0,listaCapacidades, listaEspecialidades);
	}


	@Override
	public String toString() {
		return "Ejercicio3Problem [index=" + index + ", remainingInv=" + remainingInv + ",
remainingEsp=" + remainingEsp
				+ "]";
	}

	@Override
	public int hashCode() {
		return Objects.hash(index, remainingEsp, remainingInv);
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Ejercicio3Problem other = (Ejercicio3Problem) obj;
		return Objects.equals(index, other.index) && Objects.equals(remainingEsp,
other.remainingEsp)
				&& Objects.equals(remainingInv, other.remainingInv);
	}

	public List<Integer> actions() {
		// TODO Alternativas de un vertice .
		List<Integer> res = List2.empty();
		Integer m = DatosEjercicio3.getTrabajos();
		Integer n = DatosEjercicio3.getInvestigadores();
		Integer indiceTrabajoActual = indiceTrabajoActual();
		Integer indiceInvActual = indiceInvActual();
```

```
            if(this.index==(n * m)) { //si estoy en el último vértice
                  return res;
            }else if(this.remainingInv.get(indiceInvActual) == 0) { //si no tengo más días
disponibles
                  res.add(0);
                  return res;
            }else {
                  //cuántos movimientos puedo hacer? Dependo de la capacidad que le qude al
investigador actual y de
                  //los días que falten para cubrir la especialidad que trata ese
investigador
                  res.add(0);
                  Investigador investigadorActual =
DatosEjercicio3.investigadores.get(indiceInvActual);
                  Integer especialidadActual = investigadorActual.especialidad();

                  List<Integer> remainingEspecialidades = disponibleEspecialidad();

                  List<Integer> listaTrabajos = this.remainingEsp.get(indiceTrabajoActual);

                  Integer capacidadDisp = this.remainingInv.get(indiceInvActual);
                  Integer capacidadDisp2 = remainingEspecialidades.get(especialidadActual);
                  Integer capacidadNecesito = listaTrabajos.get(especialidadActual);
                  Integer resultado = Math.min(Math.min(capacidadDisp, capacidadDisp2),
capacidadNecesito);
                  List<Integer> listaRes = IntStream.rangeClosed(1,
resultado).boxed().toList();

                  res.addAll(listaRes);
            }

            return res;

      }


      public Ejercicio3Problem neighbor(Integer a) {
            // TODO Vertice siguiente al actual segun la alternativa a

            List<Integer> remainingInv2 = List2.copy(remainingInv);
            List<List<Integer>> remainingEsp2 = List2.copy(remainingEsp);

            Integer indiceTrabajoActual = indiceTrabajoActual();
            Integer indiceInvActual = indiceInvActual();
            Investigador investigadorActual =
DatosEjercicio3.investigadores.get(indiceInvActual);

            //cuando creo un vecino tengo que actualizar la lista de días disponibles de los
investigadores
            Integer capacidadInvestigador = this.remainingInv.get(indiceInvActual);
            remainingInv2.set(indiceInvActual, capacidadInvestigador-a);
            //cuando creo un vecino tengo que actualizar los días que quedan para completar el
trabajo
            Integer diasQuedan =
remainingEsp.get(indiceTrabajoActual).get(investigadorActual.especialidad());

            List<Integer> lista_aux = List2.setElement(remainingEsp2.get(indiceTrabajoActual),
investigadorActual.especialidad(), diasQuedan - a);
            remainingEsp2.set(indiceTrabajoActual, lista_aux);

            return of(index + 1, remainingInv2, remainingEsp2);
```

```
	}

	public Double heuristic() {
		Double res = 0.;
		List<Integer> remainingHoras = remainingInv(); //lista de horas disponibles para cada trabajador
		List<Integer> remainingEspecialidades = List2.ofTam(0, DatosEjercicio3.getEspecialidades()); //lista de horas disponibles para cada especialidad
		List<List<Integer>> remainingTotal = remainingEsp(); //Lista de lo que falta para completar cada trabajo
		for(int i=0; i < remainingHoras.size();i++) {
			Integer especialidad = DatosEjercicio3.investigadores.get(i).especialidad();
			Integer horasAhora = remainingEspecialidades.get(especialidad) + remainingHoras.get(i);
			remainingEspecialidades.set(especialidad, horasAhora ); //actualizo el tiempo disponible para cada especialidad
		}

		for(List<Integer> l : remainingTotal) {
			if(termina(remainingEspecialidades, l)) {
				res += DatosEjercicio3.trabajos.get(remainingTotal.indexOf(l)).calidad();
			}
		}
		return res;
	}

	public Integer calidadActual() {
		Integer res = IntStream.range(0, this.remainingEsp.size())
				.boxed()
				.filter(x -> this.remainingEsp.get(x).stream().allMatch(y -> y == 0))
				.map(x -> DatosEjercicio3.trabajos.get(x).calidad())
				.reduce((a,b) -> a = a+b).orElse(0);
		return res;
	}


	public Integer indiceTrabajoActual() {
		Integer m = DatosEjercicio3.getTrabajos();
		return this.index%m;
	}

	public Integer indiceInvActual() {
		Integer m = DatosEjercicio3.getTrabajos();
		return this.index/m;
	}

	public List<Integer> disponibleEspecialidad(){
		List<Integer> remainingEspecialidades = List2.ofTam(0, DatosEjercicio3.getEspecialidades()); //lista de horas disponibles para cada especialidad
		for(int i=0; i < this.remainingInv.size();i++) {
			Integer especialidad = DatosEjercicio3.investigadores.get(i).especialidad();
			Integer horasAhora = remainingEspecialidades.get(especialidad) + this.remainingInv.get(i);
			remainingEspecialidades.set(especialidad, horasAhora );//actualizo el tiempo disponible para cada especialidad
		}
		return remainingEspecialidades;
	}
```

```
	public static Boolean termina(List<Integer> rE,List<Integer> l ) {
		Boolean res = true;
		for(int i = 0; i < rE.size(); i++) {
			if(l.get(i)-rE.get(i) > 0) res = false;
		}
		return res;
	}
}
```

Ejercicio4

DatosEjercicio4.java

```
package _datos;

import java.util.ArrayList;
import java.util.List;

import org.jgrapht.Graph;

import us.lsi.graphs.Graphs2;
import us.lsi.graphs.GraphsReader;

public class DatosEjercicio4 {

	private static int id_aux = 0;
	public record Conexion(int id, Double distancia) {
		public static Conexion ofFormat(String[] formato) {
			Integer id = id_aux++;
			Double dist = Double.valueOf(formato[2].trim());
			return new Conexion(id, dist);
		}
	}

	public record Cliente(int id, Double beneficio) {
		public static Cliente ofFormat(String[] formato) {
			Integer id = Integer.valueOf(formato[0].trim());
			Double benef = Double.valueOf(formato[1].trim());
			return new Cliente(id, benef);
		}
	}

	public static Graph<Cliente, Conexion> g;
	public static void iniDatos(String fichero) {
		g = GraphsReader.newGraph(fichero, Cliente::ofFormat, Conexion::ofFormat,
				Graphs2::simpleWeightedGraph);
		toConsole();
	}
	public static Integer getNumVertices() {
		return g.vertexSet().size();
	}


	public static Cliente getCliente(Integer i) { //no puedo acceder a los elementos de un
set con i porque no están ordenados, me hace falta un id
		List<Cliente> vertices = new ArrayList<>(g.vertexSet());
		return vertices.stream().filter(x -> x.id()==i).findFirst().get();
	}
	public static Double getBeneficio(Integer i) {
		return getCliente(i).beneficio();
	}
	public static Boolean existeArista(Integer i, Integer j) {
		Cliente c1 = getCliente(i);
		Cliente c2 = getCliente(j);
		return g.containsEdge(c1, c2);
	}
	public static Double getDistancia(Integer i, Integer j) {
		Cliente c1 = getCliente(i);
		Cliente c2 = getCliente(j);
		return g.getEdge(c1, c2).distancia();
```

```
	}
	private static void toConsole() {
		System.out.println(g.vertexSet());
		System.out.println(g.edgeSet());
	}
	public static void main(String[] args) {
		iniDatos("ficheros/Ejercicio4DatosEntrada1.txt");
		System.out.println(getDistancia(2, 4));
	}
}
```

SolucionEjercicio4.java

```
package _soluciones;

import java.util.ArrayList;
import java.util.List;

import _datos.DatosEjercicio4;
import _datos.DatosEjercicio4.Cliente;

public class SolucionEjercicio4 {

	public static SolucionEjercicio4 of_Range(List<Integer> ls) {
		return new SolucionEjercicio4(ls);
	}

	private Double kms;
	private Double benef;
	private List<Cliente> clientes;

	private SolucionEjercicio4() {
		kms = 0.;
		benef = 0.;
		clientes = new ArrayList<>();
		Cliente c0 = DatosEjercicio4.getCliente(0);
		clientes.add(c0);
	}
	private SolucionEjercicio4(List<Integer> ls) {
		kms = 0.;
		benef = 0.;
		clientes = new ArrayList<>();
		Cliente c0 = DatosEjercicio4.getCliente(0);
		clientes.add(c0);
		for (int i = 0; i < ls.size(); i++) {
			Cliente c = DatosEjercicio4.getCliente(ls.get(i));
			clientes.add(c);
			if (i == 0) {
				if (DatosEjercicio4.existeArista(0, ls.get(i))) {
					kms += DatosEjercicio4.getDistancia(0, ls.get(i));
					benef += DatosEjercicio4.getBeneficio(ls.get(i)) - kms;
				}
			} else {
				if (DatosEjercicio4.existeArista(ls.get(i - 1), ls.get(i))) {
					kms += DatosEjercicio4.getDistancia(ls.get(i - 1), ls.get(i));
					benef += DatosEjercicio4.getBeneficio(ls.get(i)) - kms;
				}
			}
		}

	}

	public static SolucionEjercicio4 empty() {
		return new SolucionEjercicio4();
	}

	@Override
	public String toString() {
		List<Integer> ids = clientes.stream().map(c -> c.id()).toList();
		return "Camino a seguir:\n" + ids + "\nDistancia: " + kms + "\nBeneficio: " +
benef;
	}
```

```
}
```

Ejercicio4Edge

```
package ejercicios.ejercicio4;


import _datos.DatosEjercicio4;
import _datos.DatosEjercicio4.Cliente;
import us.lsi.graphs.virtual.SimpleEdgeAction;

public record Ejercicio4Edge(Ejercicio4Vertex source, Ejercicio4Vertex target, Integer action,
Double weight)
		implements SimpleEdgeAction<Ejercicio4Vertex,Integer> {

	public static Ejercicio4Edge of(Ejercicio4Vertex v1, Ejercicio4Vertex v2, Integer a) {

		// TODO La arista debe tener peso
		Cliente clienteActual = DatosEjercicio4.getCliente(v1.index());
		return new Ejercicio4Edge(v1, v2, a, clienteActual.beneficio() -
v1.km_Recorridos()*0.01 );
	}

	@Override
	public String toString() {
		return String.format("%d; %.1f", action, weight);
	}

}
```

Ejercicio4Heuristic

```
package ejercicios.ejercicio4;

import java.util.Comparator;
import java.util.Iterator;
import java.util.List;
import java.util.function.Predicate;
import java.util.stream.IntStream;

import _datos.DatosEjercicio4;
import _datos.DatosEjercicio4.Cliente;
import us.lsi.common.List2;


public class Ejercicio4Heuristic {

	// Se explica en practicas.
	public static Double heuristic(Ejercicio4Vertex v1, Predicate<Ejercicio4Vertex> goal,
Ejercicio4Vertex v2) {
		Double res = 0.;

		List<Integer> clientesPendientes = IntStream.range(0,
DatosEjercicio4.getNumVertices()).boxed().toList();
		List<Integer> clientesPendientes2 = List2.intersection(clientesPendientes,
v1.clientesVisitados());
		List<Cliente> clientes = clientesPendientes2.stream().map(x ->
DatosEjercicio4.getCliente(x)).sorted(Comparator.comparing(Cliente::beneficio).reversed()).toLi
st();
		for(Cliente c:clientes) {
			res+=c.beneficio()-clientes.indexOf(c);
		}
		return res;



	}


	//el orden es lo importante
	//miro los que tengo pendientes
	//a cada pendiente supongo que son adyacentes y cuando llegue me va a dar el beneficio
posible (la penalizacion por el orden es pequeña, "1,2,...").
	//El orden que utilizo es el de la lista del segundo paso


}
```

Ejercicio4Vertex

```
package ejercicios.ejercicio4;

import java.util.Comparator;
import java.util.List;
import java.util.Set;
import java.util.function.Predicate;
import java.util.stream.Collectors;

import org.jgrapht.Graph;

import _datos.DatosEjercicio4;
import _datos.DatosEjercicio4.Cliente;
import _datos.DatosEjercicio4.Conexion;
import us.lsi.common.List2;
import us.lsi.common.Set2;
import us.lsi.graphs.virtual.VirtualVertex;

// Uso el segundo modelo
public record Ejercicio4Vertex(Integer index, Set<Integer> remainingClientes ,List<Integer>
clientesVisitados, Integer km_Recorridos)
implements VirtualVertex<Ejercicio4Vertex,Ejercicio4Edge,Integer> {

	public static Ejercicio4Vertex of(Integer i, Set<Integer> rest ,List<Integer> clien,
Integer kms) {
		return new Ejercicio4Vertex(i, rest, clien, kms);
	}

	@Override
	public List<Integer> actions() { //tengo tantas alternativas como vértices adyacentes, y
el valor será el índice del vértice destino
		// TODO Alternativas de un vertice
		List<Integer> res = List2.empty();
		if(goal().test(this)) return res; //si he terminado no devuelvo nada
		else if(this.clientesVisitados().size()==DatosEjercicio4.getNumVertices()+1) {
//si estoy en el último vértice, vuelvo al inicio
			if(vuelve()) return List2.empty();
			else return res;
		}
		else {
			res = this.remainingClientes.stream()
					.filter(x -> adyacentes(x, this.index) ).toList();

			return res;
		}

	}

	@Override
	public Ejercicio4Vertex neighbor(Integer a) {
		// TODO Vertice siguiente al actual segun la alternativa a
		//actualizar remainingClientes
		Set<Integer> remainingClientes2 = Set2.copy(this.remainingClientes());
		remainingClientes2.remove(a);
		//actualizar clientesVisitados
		List<Integer> clientesVisitados2 = List2.copy(this.clientesVisitados);
		clientesVisitados2.add(a);
		//actualizar km_Recorridos
		Integer kilometros2 = this.km_Recorridos +
DatosEjercicio4.getDistancia(this.index, a).intValue();
```

```
			return new Ejercicio4Vertex(a, remainingClientes2, clientesVisitados2,
kilometros2);
		}

		@Override
		public Ejercicio4Edge edge(Integer a) {
			return Ejercicio4Edge.of(this,this.neighbor(a),a);
		}

		// Se explica en practicas.
		public Ejercicio4Edge greedyEdge() {
			Integer clienteActual = this.index;
			Comparator<Integer> cmp = Comparator.comparing(x ->
DatosEjercicio4.getBeneficio(x)-obtienePesoArista(clienteActual, x));
			List<Integer> interseccion = this.remainingClientes.stream().filter(x ->
adyacentes(x, this.index)).toList();
			Integer max = interseccion.stream().max(cmp).orElse(0);
			return edge(max);
		}

		public Double obtienePesoArista(Integer a, Integer b) {
			Cliente clienteA = DatosEjercicio4.getCliente(a);
			Cliente clienteB = DatosEjercicio4.getCliente(b);
			return DatosEjercicio4.g.getEdge(clienteA, clienteB).distancia();
		}


		public Set<Integer> obtieneVecinos(Integer a){
			Cliente clienteA = DatosEjercicio4.getCliente(a);
			List<Cliente> adyacentes1 =
DatosEjercicio4.g.incomingEdgesOf(clienteA).stream().map(x ->
DatosEjercicio4.g.getEdgeSource(x)).toList();
			List<Cliente> adyacentes2 =
DatosEjercicio4.g.outgoingEdgesOf(clienteA).stream().map(x ->
DatosEjercicio4.g.getEdgeTarget(x)).toList();
			List<Cliente> adyacentes3 = List2.concat(adyacentes1, adyacentes2);
			Set<Integer> adyacentes = adyacentes3.stream().map(x ->
x.id()).collect(Collectors.toSet());
			adyacentes.stream().filter(x -> !this.clientesVisitados.contains(x) && x !=a);
			adyacentes.remove(a);
			adyacentes.removeAll(this.clientesVisitados);
			return adyacentes;
		}

		public static Ejercicio4Vertex initial() {
			// TODO Auto-generated method stub
			Set<Integer> setVertices =
DatosEjercicio4.g.vertexSet().stream().map(Cliente::id).collect(Collectors.toSet());
			return new Ejercicio4Vertex(0,setVertices, List2.of(0), 0);
		}

		public static Predicate<Ejercicio4Vertex> goal() {
			// TODO Auto-generated method stub
			Predicate<Ejercicio4Vertex> pred =  p ->
p.clientesVisitados.size()==DatosEjercicio4.getNumVertices()+1; //paro cuando he recorrido
todos los vértices

			return pred;
		}
```

```
	public static Predicate<Ejercicio4Vertex> goalHasSolution() { //tiene solucion si he recorrido todos los vértices
		// TODO Auto-generated method stub
		Predicate<Ejercicio4Vertex> pred =  p -> p.remainingClientes.isEmpty() && p.clientesVisitados.size()==DatosEjercicio4.getNumVertices()+1;
		return pred;
	}

	public Boolean adyacentes(Integer x, Integer y) {
		return DatosEjercicio4.existeArista(y, x) || DatosEjercicio4.existeArista(x, y);
	}

	public Boolean vuelve() {
		Integer ultimo = this.clientesVisitados.get(DatosEjercicio4.getNumVertices()-1);
		return adyacentes(ultimo, 0);
	}

}
```

Manual

Ejercicio4BT

```
package ejercicios.ejercicio4.manual;

import java.util.Set;

import _soluciones.SolucionEjercicio4;
import us.lsi.common.Set2;

public class Ejercicio4BT {

	private static Double mejorValor;
	private static Ejercicio4State estado;
	private static Set<SolucionEjercicio4> soluciones;

	public static void search() {
		soluciones = Set2.newTreeSet();
		mejorValor = Double.MIN_VALUE; // Estamos maximizando
		estado = Ejercicio4State.initial();
		bt_search();
	}

	private static void bt_search() {
		if (estado.esSolucion()) {
			Double valorObtenido = estado.acumulado;
			if (valorObtenido > mejorValor) {  // Estamos maximizando
				mejorValor = valorObtenido;
				soluciones.add(estado.getSolucion());
			}
		} else if(!estado.esTerminal()){
			for (Integer a: estado.alternativas()) {
				if (estado.cota(a) >= mejorValor) {  // Estamos maximizando
					estado.forward(a);
					bt_search();
					estado.back();
				}
			}
		}
	}
```

```
	public static Set<SolucionEjercicio4> getSoluciones() {
		return soluciones;
	}

}
```

Ejercicio4State

```
package ejercicios.ejercicio4.manual;

import java.util.List;

import _datos.DatosEjercicio4;
import _soluciones.SolucionEjercicio4;
import us.lsi.common.List2;

public class Ejercicio4State {

	Ejercicio4Problem actual;
	Double acumulado;
	List<Integer> acciones;
	List<Ejercicio4Problem> anteriores;

	private Ejercicio4State(Ejercicio4Problem p, Double a,
		List<Integer> ls1, List<Ejercicio4Problem> ls2) {
		// TODO Inicializar las propiedades individuales
		actual = p;
		acumulado = a;
		acciones = ls1;
		anteriores =  ls2;
	}

	public static Ejercicio4State initial() {
		// TODO Crear el estado inicial
		return new
Ejercicio4State(Ejercicio4Problem.initial(),0.,List2.empty(),List2.empty());
	}

	public static Ejercicio4State of(Ejercicio4Problem prob, Double acum, List<Integer> lsa,
			List<Ejercicio4Problem> lsp) {
		return new Ejercicio4State(prob, acum, lsa, lsp);
	}

	public void forward(Integer a) {
		// TODO Avanzar un estado segun la alternativa a
		acciones.add(a);
		anteriores.add(actual);

		acumulado += DatosEjercicio4.getBeneficio(actual.index());
		actual = actual.neighbor(a);
	}

	public void back() {
		// TODO Retroceder al estado anterior
		Integer indice_ultimo_problema = anteriores.size()-1;
		Ejercicio4Problem prob_anterior = anteriores.get(indice_ultimo_problema);

		Integer indice_de_la_decision = prob_anterior.index();

		Integer accion_anterior = acciones.get(indice_ultimo_problema);
```

```
			acumulado =  DatosEjercicio4.getBeneficio(indice_de_la_decision);
			actual = prob_anterior;
			acciones.remove(accion_anterior);
			anteriores.remove(prob_anterior);
		}

		public List<Integer> alternativas() {
			// TODO Alternativas segun el actual
			return this.actual.actions();
		}

		public Double cota(Integer a) {
			// TODO Cota = acumulado + func(a, actual) + h(vecino(actual, a))
			Double weight = DatosEjercicio4.getBeneficio(a);
			return this.acumulado + weight + actual.neighbor(a).heuristic();
		}

		public Boolean esSolucion() {
			// TODO Cuando todos los elementos del universo se han cubierto

			return actual.remainingClientes().isEmpty() &&
actual.clientesVisitados().get(actual.clientesVisitados().size()-1)==0;
		}

		public Boolean esTerminal() {
			// TODO Cuando se han recorrido todos los Ejercicio4
			return actual.clientesVisitados().size()==DatosEjercicio4.getNumVertices()+1;
		}

		public SolucionEjercicio4 getSolucion() {
			// TODO Aprovechamos lo hecho en la PI4
			return SolucionEjercicio4.of_Range(acciones);
		}

}
```

Ejercicio4Problem

```
package ejercicios.ejercicio4.manual;


import java.util.Comparator;
import java.util.List;
import java.util.Objects;
import java.util.Set;
import java.util.function.Predicate;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

import org.jgrapht.Graph;

import _datos.DatosEjercicio4;
import _datos.DatosEjercicio4.Cliente;
import _datos.DatosEjercicio4.Conexion;
import us.lsi.common.List2;
import us.lsi.common.Set2;

public record Ejercicio4Problem(Integer index, Set<Integer> remainingClientes ,List<Integer>
clientesVisitados, Double km_Recorridos)
```

```
{

	public static Ejercicio4Problem of(Integer i, Set<Integer> rest ,List<Integer> clien,
Double kms) {
		return new Ejercicio4Problem(i, rest, clien, kms);
	}

	// TODO Consulte las clases GraphsPI5 y TestPI5
	public static Predicate<Ejercicio4Problem> goal(){
		return p -> p.clientesVisitados.size()==DatosEjercicio4.getNumVertices()+1;
	}

	public static Predicate<Ejercicio4Problem> goalHasSolution(){
		return p -> p.remainingClientes.isEmpty() &&
p.clientesVisitados.get(p.clientesVisitados.size()-1)==0;
	}

	public static Ejercicio4Problem initial() {
		Set<Integer> setVertices =
DatosEjercicio4.g.vertexSet().stream().map(Cliente::id).collect(Collectors.toSet());
		return of(0,setVertices, List2.of(0), 0.);
	}


	@Override
	public String toString() {
		return "Ejercicio4Problem [index=" + index + ", remainingClientes=" +
remainingClientes + ", clientesVisitados="
				+ clientesVisitados + ", km_Recorridos=" + km_Recorridos + "]";
	}

	@Override
	public int hashCode() {
		return Objects.hash(clientesVisitados, index, km_Recorridos, remainingClientes);
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Ejercicio4Problem other = (Ejercicio4Problem) obj;
		return Objects.equals(clientesVisitados, other.clientesVisitados) &&
Objects.equals(index, other.index)
				&& Objects.equals(km_Recorridos, other.km_Recorridos)
				&& Objects.equals(remainingClientes, other.remainingClientes);
	}

	public List<Integer> actions() {
		// TODO Alternativas de un vertice .
		List<Integer> res = List2.empty();
		if(goal().test(this)) return res; //si he terminado no devuelvo nada
		else if(this.clientesVisitados().size()==DatosEjercicio4.getNumVertices()+1) {
//si estoy en el último vértice, vuelvo al inicio
			if(vuelve()) return List2.empty();
			else return res;
		}
		else {
```

```
			res = this.remainingClientes.stream()
					.filter(x -> DatosEjercicio4.existeArista(this.index, x))
					.toList();
			return res;
		}

	}


	public Ejercicio4Problem neighbor(Integer a) {
		// TODO Vertice siguiente al actual segun la alternativa a

		Set<Integer> remainingClientes2 = Set2.copy(this.remainingClientes());
		remainingClientes2.remove(a);
		//actualizar clientesVisitados
		List<Integer> clientesVisitados2 = List2.copy(this.clientesVisitados);
		clientesVisitados2.add(a);
		//actualizar km_Recorridos
		Double kilometros2 = this.km_Recorridos + DatosEjercicio4.getDistancia(this.index,
a);

		return Ejercicio4Problem.of(a, remainingClientes2, clientesVisitados2,
kilometros2);
	}

	public Double heuristic() {
		Double res = 0.;
		List<Integer> clientesPendientes = IntStream.range(0,
DatosEjercicio4.getNumVertices()).boxed().toList();
		List<Integer> clientesPendientes2 = List2.intersection(clientesPendientes,
clientesVisitados());
		List<Cliente> clientes = clientesPendientes2.stream().map(x ->
DatosEjercicio4.getCliente(x)).sorted(Comparator.comparing(Cliente::beneficio).reversed()).toLi
st();
		for(Cliente c:clientes) {
			res+=c.beneficio()-clientes.indexOf(c);
		}
		return res;
	}




	public Boolean adyacentes(Integer x, Integer y) {
		return DatosEjercicio4.existeArista(y, x) || DatosEjercicio4.existeArista(x, y);
	}


	public Boolean existeAristaExtricto(Integer x, Integer y) {
		Graph<Cliente, Conexion> grafo = DatosEjercicio4.g;
		Cliente cliente1 = DatosEjercicio4.getCliente(x);
		Cliente cliente2 = DatosEjercicio4.getCliente(y);
		Boolean res = grafo.outgoingEdgesOf(cliente1).stream().anyMatch(a ->
grafo.getEdgeTarget(a) == cliente2);
		System.out.println("Existe la arista entre " + x + " y " + y + "?: " + res);
		return res;
	}

	public Boolean vuelve() {
		Integer ultimo = this.clientesVisitados.get(DatosEjercicio4.getNumVertices()-1);
		return adyacentes(ultimo, 0);
```

```
        }
}
```

Tests

TestEjercicio1.java

```
package ejercicios.tests;

import java.util.List;
import java.util.function.Predicate;

import _datos.DatosEjercicio1;
import _soluciones.SolucionEjercicio1;
import _utils.GraphsPI5;
import _utils.TestsPI5;
import ejercicios.ejercicio1.Ejercicio1Vertex;

public class TestsEjercicio1 {

	public static void main(String[] args) {
		List.of(1,2,3).forEach(num_test -> {
			TestsPI5.iniTest("Ejercicio1DatosEntrada", num_test,
DatosEjercicio1::iniDatos);

			// TODO Defina un m. factoria para el vertice inicial
			Ejercicio1Vertex v_inicial = Ejercicio1Vertex.initial();
			// TODO Defina un m. static para los vertices finales
			Predicate<Ejercicio1Vertex> es_terminal = Ejercicio1Vertex.goal();

			var gp = TestsPI5.testGreedy(GraphsPI5.greedyEjercicio1Graph(v_inicial,
es_terminal));
			TestsPI5.toConsole("Voraz", gp, SolucionEjercicio1::of);

			var path = TestsPI5.testAStar(GraphsPI5.Ejercicio1Graph(v_inicial,
es_terminal), gp);
			TestsPI5.toConsole("A*", path, SolucionEjercicio1::of);

			path = TestsPI5.testPDR(GraphsPI5.Ejercicio1Graph(v_inicial, es_terminal),
gp);
			TestsPI5.toConsole("PDR", path, SolucionEjercicio1::of);

			path = TestsPI5.testBT(GraphsPI5.Ejercicio1Graph(v_inicial, es_terminal),
gp);
			TestsPI5.toConsole("BT", path, SolucionEjercicio1::of);

			TestsPI5.line("*");

		});
	}

}
```

```
[Tipo[Nombre_Tipo=C01, kgdisponibles=5, id=1], Tipo[Nombre_Tipo=C02, kgdisponibles=4, id=2], Tipo[Nombre_Tipo=C03, kgdisponibles=1, id=3], Tipo[Nombre_Tipo=C04, kgdisponibles=2, id=4], Tipo[Nombre_Tipo=C05, kgdisponibles=8, id=5], Tipo[Nombre_
[Variedad[nombre=P01, id=null, beneficio=20.0, porcentaje={C02=0.4, C01=0.5, C03=0.1}], Variedad[nombre=P02, id=null, beneficio=10.0, porcentaje={C04=0.2, C05=0.8}], Variedad[nombre=P03, id=null, beneficio=5.0, porcentaje={C06=1.0}]]
================================================================================
Solucion Voraz: Solucion = [10, 10, 1]; Tamaño solucion = 3; beneficio total = 305,000000;
Path de la solucion: [10, 10, 1]
--------------------------------------------------------------------------------
Solucion A*: Solucion = [10, 10, 1]; Tamaño solucion = 3; beneficio total = 305,000000;
Path de la solucion: [10, 10, 1]
--------------------------------------------------------------------------------
Solucion PDR: Solucion = [10, 10, 1]; Tamaño solucion = 3; beneficio total = 305,000000;
Path de la solucion: [10, 10, 1]
--------------------------------------------------------------------------------
Solucion BT: Solucion = [10, 10, 1]; Tamaño solucion = 3; beneficio total = 305,000000;
Path de la solucion: [10, 10, 1]
--------------------------------------------------------------------------------
********************************************************************************
[Tipo[Nombre_Tipo=C01, kgdisponibles=11, id=1], Tipo[Nombre_Tipo=C02, kgdisponibles=9, id=2], Tipo[Nombre_Tipo=C03, kgdisponibles=7, id=3], Tipo[Nombre_Tipo=C04, kgdisponibles=12, id=4], Tipo[Nombre_Tipo=C05, kgdisponibles=6, id=5]]
[Variedad[nombre=P01, id=null, beneficio=20.0, porcentaje={C02=0.4, C01=0.2, C05=0.4}], Variedad[nombre=P02, id=null, beneficio=10.0, porcentaje={C02=0.3, C03=0.7}], Variedad[nombre=P03, id=null, beneficio=80.0, porcentaje={C01=0.4, C04=0.6}]]
================================================================================
Solucion Voraz: Solucion = [15, 10, 20]; Tamaño solucion = 3; beneficio total = 2000,000000;
Path de la solucion: [15, 10, 20]
--------------------------------------------------------------------------------
Solucion A*: Solucion = [15, 10, 20]; Tamaño solucion = 3; beneficio total = 2000,000000;
Path de la solucion: [15, 10, 20]
--------------------------------------------------------------------------------
Solucion PDR: Solucion = [15, 10, 20]; Tamaño solucion = 3; beneficio total = 2000,000000;
Path de la solucion: [15, 10, 20]
--------------------------------------------------------------------------------
Solucion BT: Solucion = [15, 10, 20]; Tamaño solucion = 3; beneficio total = 2000,000000;
Path de la solucion: [15, 10, 20]
--------------------------------------------------------------------------------
********************************************************************************
[Tipo[Nombre_Tipo=C01, kgdisponibles=35, id=1], Tipo[Nombre_Tipo=C02, kgdisponibles=4, id=2], Tipo[Nombre_Tipo=C03, kgdisponibles=12, id=3], Tipo[Nombre_Tipo=C04, kgdisponibles=5, id=4], Tipo[Nombre_Tipo=C05, kgdisponibles=30, id=5], Tipo[Nomb
[Variedad[nombre=P01, id=null, beneficio=60.0, porcentaje={C01=0.5, C03=0.4, C07=0.1}], Variedad[nombre=P02, id=null, beneficio=25.0, porcentaje={C02=1.0}], Variedad[nombre=P03, id=null, beneficio=5.0, porcentaje={C02=0.4, C07=0.8}], Variedad[
================================================================================
Solucion Voraz: Solucion = [30, 4, 15, 100]; Tamaño solucion = 4; beneficio total = 12275,000000;
Path de la solucion: [30, 4, 0, 15, 0, 100]
--------------------------------------------------------------------------------
Solucion A*: Solucion = [30, 4, 15, 100]; Tamaño solucion = 4; beneficio total = 12275,000000;
Path de la solucion: [30, 4, 0, 15, 0, 100]
--------------------------------------------------------------------------------
Solucion PDR: Solucion = [30, 4, 15, 100]; Tamaño solucion = 4; beneficio total = 12275,000000;
Path de la solucion: [30, 4, 0, 15, 0, 100]
--------------------------------------------------------------------------------
Solucion BT: Solucion = [30, 4, 15, 100]; Tamaño solucion = 4; beneficio total = 12275,000000;
Path de la solucion: [30, 4, 0, 15, 0, 100]
--------------------------------------------------------------------------------
********************************************************************************
```

TestEjercicio2.java

```java
package ejercicios.tests;

import java.util.List;
import java.util.function.Predicate;

import _datos.DatosEjercicio2;
import _soluciones.SolucionEjercicio2;
import _utils.GraphsPI5;
import _utils.TestsPI5;
import ejercicios.ejercicio2.Ejercicio2Vertex;

public class TestsEjercicio2 {

	public static void main(String[] args) {
		List.of(3).forEach(num_test -> {
			TestsPI5.iniTest("Ejercicio2DatosEntrada", num_test,
DatosEjercicio2::iniDatos);

			// TODO Defina un m. factoria para el vertice inicial
			Ejercicio2Vertex v_inicial = Ejercicio2Vertex.initial();
			// TODO Defina un m. static para los vertices finales
			Predicate<Ejercicio2Vertex> es_terminal = Ejercicio2Vertex.goal();

			var gp = TestsPI5.testGreedy(GraphsPI5.greedyEjercicio2Graph(v_inicial,
es_terminal));
			TestsPI5.toConsole("Voraz", gp, SolucionEjercicio2::of);

			var path = TestsPI5.testAStar(GraphsPI5.Ejercicio2Graph(v_inicial,
es_terminal), gp);
			TestsPI5.toConsole("A*", path, SolucionEjercicio2::of);

			path = TestsPI5.testPDR(GraphsPI5.Ejercicio2Graph(v_inicial, es_terminal),
gp);
			TestsPI5.toConsole("PDR", path, SolucionEjercicio2::of);

			path = TestsPI5.testBT(GraphsPI5.Ejercicio2Graph(v_inicial, es_terminal),
gp);
```

```
                TestsPI5.toConsole("BT", path, SolucionEjercicio2::of);

                TestsPI5.line("*");

            });
        }

    }
```

```
[Curso[id=0, tematicas=[1, 2, 3, 4], coste=10.0, centro=0], Curso[id=1, tematicas=[1, 4], coste=3.0, centro=0], Curso[id=2, tematicas=[5], coste=1.5, centro=1], Curso[id=3, tematicas=[5], coste=5.0, centro=0]]
Max_Centros = 1
********************************************************************************************
Solucion Voraz: Cursos elegidos: {S0, S3}
Coste Total: 15,0
____________________________________________________________________________________________
Solucion A*: Cursos elegidos: {S0, S3}
Coste Total: 15,0
____________________________________________________________________________________________
Solucion PDR: Cursos elegidos: {S0, S3}
Coste Total: 15,0
____________________________________________________________________________________________
Solucion BT: Cursos elegidos: {S0, S3}
Coste Total: 15,0
____________________________________________________________________________________________
********************************************************************************************
```

```
[Curso[id=0, tematicas=[2, 3], coste=2.0, centro=0], Curso[id=1, tematicas=[4], coste=3.0, centro=0], Curso[id=2, tematicas=[1, 5], coste=5.0, centro=0], Curso[id=3, tematicas=[1, 3, 4], coste=3.5, centro=2], Curso[id=4, tematicas=[4, 5], cost
Max_Centros = 2
********************************************************************************************
Solucion Voraz: Cursos elegidos: {S0, S1, S2}
Coste Total: 10,0
____________________________________________________________________________________________
Solucion A*: Cursos elegidos: {S0, S2, S4}
Coste Total: 8,5
____________________________________________________________________________________________
Solucion PDR: Cursos elegidos: {S0, S2, S4}
Coste Total: 8,5
____________________________________________________________________________________________
Solucion BT: Cursos elegidos: {S0, S2, S4}
Coste Total: 8,5
____________________________________________________________________________________________
********************************************************************************************
```

```
[Curso[id=0, tematicas=[2, 6, 7], coste=2.0, centro=2], Curso[id=1, tematicas=[7], coste=3.0, centro=0], Curso[id=2, tematicas=[1, 5], coste=5.0, centro=0], Curso[id=3, tematicas=[1, 3, 4], coste=3.5, centro=2], Curso[id=4, tematicas=[3, 7], c
Max_Centros = 3
********************************************************************************************
Solucion Voraz: Cursos elegidos: {S0, S2, S3}
Coste Total: 10,5
____________________________________________________________________________________________
Solucion A*: Cursos elegidos: {S0, S3, S7}
Coste Total: 6,5
____________________________________________________________________________________________
Solucion PDR: Cursos elegidos: {S0, S3, S7}
Coste Total: 6,5
____________________________________________________________________________________________
Solucion BT: Cursos elegidos: {S0, S3, S7}
Coste Total: 6,5
____________________________________________________________________________________________
********************************************************************************************
```

TestEjercicio3.java

```
package ejercicios.tests;

import java.util.List;
import java.util.function.Predicate;

import _datos.DatosEjercicio3;
import _soluciones.SolucionEjercicio3;
import _utils.GraphsPI5;
import _utils.TestsPI5;
import ejercicios.ejercicio3.Ejercicio3Vertex;

//Clase para todos los tests del ejemplo 3 mediante Greedy, A*, PDR y BT
public class TestsEjercicio3 {


    public static void main(String[] args) {
        List.of(1,2,3).forEach(num_test -> {
            TestsPI5.iniTest("Ejercicio3DatosEntrada", num_test,
DatosEjercicio3::iniDatos);

            // TODO Defina un m. factoria para el vertice inicial
            Ejercicio3Vertex v_inicial = Ejercicio3Vertex.initial();
            // TODO Defina un m. static para los vertices finales
            Predicate<Ejercicio3Vertex> es_terminal = Ejercicio3Vertex.goal();
```

```
                var gp = TestsPI5.testGreedy(GraphsPI5.greedyEjercicio3Graph(v_inicial,
es_terminal));
                TestsPI5.toConsole("Voraz", gp, SolucionEjercicio3::of);

                var path = TestsPI5.testAStar(GraphsPI5.Ejercicio3Graph(v_inicial,
es_terminal), gp);
                TestsPI5.toConsole("A*", path, SolucionEjercicio3::of);

                path = TestsPI5.testPDR(GraphsPI5.Ejercicio3Graph(v_inicial, es_terminal),
gp);
                TestsPI5.toConsole("PDR", path, SolucionEjercicio3::of);

                path = TestsPI5.testBT(GraphsPI5.Ejercicio3Graph(v_inicial, es_terminal),
gp);
                TestsPI5.toConsole("BT", path, SolucionEjercicio3::of);

                TestsPI5.line("*");
            });
    }


}
```

```
[Investigador[nombre=INV1, capacidad=6, especialidad=0], Investigador[nombre=INV2, capacidad=3, especialidad=1], Investigador[nombre=INV3, capacidad=8, especialidad=2]]
[Trabajo[nombre=T01, calidad=5, reparto={0=6, 1=0, 2=0}], Trabajo[nombre=T02, calidad=10, reparto={0=0, 1=3, 2=8}]]
================================================================================================
Solucion Voraz: Reparto de horas:
INVESTIGADOR INV1: Investigador[nombre=INV1, capacidad=6, especialidad=0]
INVESTIGADOR INV2: Investigador[nombre=INV2, capacidad=3, especialidad=1]
INVESTIGADOR INV3: Investigador[nombre=INV3, capacidad=8, especialidad=2]
Suma de las calidades de los trabajos realizados: 15
------------------------------------------------------------------------------------------------
Solucion A*: Reparto de horas:
INVESTIGADOR INV1: Investigador[nombre=INV1, capacidad=6, especialidad=0]
INVESTIGADOR INV2: Investigador[nombre=INV2, capacidad=3, especialidad=1]
INVESTIGADOR INV3: Investigador[nombre=INV3, capacidad=8, especialidad=2]
Suma de las calidades de los trabajos realizados: 15
------------------------------------------------------------------------------------------------
Solucion PDR: Reparto de horas:
INVESTIGADOR INV1: Investigador[nombre=INV1, capacidad=6, especialidad=0]
INVESTIGADOR INV2: Investigador[nombre=INV2, capacidad=3, especialidad=1]
INVESTIGADOR INV3: Investigador[nombre=INV3, capacidad=8, especialidad=2]
Suma de las calidades de los trabajos realizados: 15
------------------------------------------------------------------------------------------------
Solucion BT: Reparto de horas:
INVESTIGADOR INV1: Investigador[nombre=INV1, capacidad=6, especialidad=0]
INVESTIGADOR INV2: Investigador[nombre=INV2, capacidad=3, especialidad=1]
INVESTIGADOR INV3: Investigador[nombre=INV3, capacidad=8, especialidad=2]
Suma de las calidades de los trabajos realizados: 15


------------------------------------------------------------------------------------------------
************************************************************************************************
[Investigador[nombre=INV1, capacidad=10, especialidad=0], Investigador[nombre=INV2, capacidad=5, especialidad=1], Investigador[nombre=INV3, capacidad=8, especialidad=2], Investigador[nombre=INV4, capacidad=2, especialidad=0], Investigador[nomb
[Trabajo[nombre=T01, calidad=7, reparto={0=2, 1=0, 2=5, 3=0}], Trabajo[nombre=T02, calidad=9, reparto={0=8, 1=4, 2=3, 3=0}], Trabajo[nombre=T03, calidad=5, reparto={0=2, 1=0, 2=0, 3=7}]]
================================================================================================
Solucion Voraz: Reparto de horas:
INVESTIGADOR INV1: Investigador[nombre=INV1, capacidad=10, especialidad=0]
INVESTIGADOR INV2: Investigador[nombre=INV2, capacidad=5, especialidad=1]
INVESTIGADOR INV3: Investigador[nombre=INV3, capacidad=8, especialidad=2]
INVESTIGADOR INV4: Investigador[nombre=INV4, capacidad=2, especialidad=0]
INVESTIGADOR INV5: Investigador[nombre=INV5, capacidad=5, especialidad=3]
Suma de las calidades de los trabajos realizados: 16
------------------------------------------------------------------------------------------------
Solucion A*: Reparto de horas:
INVESTIGADOR INV1: Investigador[nombre=INV1, capacidad=10, especialidad=0]
INVESTIGADOR INV2: Investigador[nombre=INV2, capacidad=5, especialidad=1]
INVESTIGADOR INV3: Investigador[nombre=INV3, capacidad=8, especialidad=2]
INVESTIGADOR INV4: Investigador[nombre=INV4, capacidad=2, especialidad=0]
INVESTIGADOR INV5: Investigador[nombre=INV5, capacidad=5, especialidad=3]
Suma de las calidades de los trabajos realizados: 16
------------------------------------------------------------------------------------------------
Solucion PDR: Reparto de horas:
INVESTIGADOR INV1: Investigador[nombre=INV1, capacidad=10, especialidad=0]
INVESTIGADOR INV2: Investigador[nombre=INV2, capacidad=5, especialidad=1]
INVESTIGADOR INV3: Investigador[nombre=INV3, capacidad=8, especialidad=2]
INVESTIGADOR INV4: Investigador[nombre=INV4, capacidad=2, especialidad=0]
INVESTIGADOR INV5: Investigador[nombre=INV5, capacidad=5, especialidad=3]
Suma de las calidades de los trabajos realizados: 16
------------------------------------------------------------------------------------------------
Solucion BT: Reparto de horas:
INVESTIGADOR INV1: Investigador[nombre=INV1, capacidad=10, especialidad=0]
INVESTIGADOR INV2: Investigador[nombre=INV2, capacidad=5, especialidad=1]
INVESTIGADOR INV3: Investigador[nombre=INV3, capacidad=8, especialidad=2]
INVESTIGADOR INV4: Investigador[nombre=INV4, capacidad=2, especialidad=0]
INVESTIGADOR INV5: Investigador[nombre=INV5, capacidad=5, especialidad=3]
Suma de las calidades de los trabajos realizados: 16
------------------------------------------------------------------------------------------------
************************************************************************************************
```

```
================================================================================
[Investigador[nombre=INV1, capacidad=1, especialidad=2], Investigador[nombre=INV2, capacidad=10, especialidad=1], Investigador[nombre=INV3, capacidad=3, especialidad=0], Investigador[nombre=INV4, capacidad=4, especialidad=0], Investigador[nom
[Trabajo[nombre=T01, calidad=8, reparto={0=2, 1=0, 2=2, 3=0}], Trabajo[nombre=T02, calidad=5, reparto={0=8, 1=5, 2=4, 3=2}], Trabajo[nombre=T03, calidad=8, reparto={0=0, 1=5, 2=0, 3=15}], Trabajo[nombre=T04, calidad=5, reparto={0=0, 1=7, 2=8,
================================================================================
Solucion Voraz: Reparto de horas:
INVESTIGADOR INV1: Investigador[nombre=INV1, capacidad=1, especialidad=2]
INVESTIGADOR INV2: Investigador[nombre=INV2, capacidad=10, especialidad=1]
INVESTIGADOR INV3: Investigador[nombre=INV3, capacidad=3, especialidad=0]
INVESTIGADOR INV4: Investigador[nombre=INV4, capacidad=4, especialidad=0]
INVESTIGADOR INV5: Investigador[nombre=INV5, capacidad=10, especialidad=3]
INVESTIGADOR INV6: Investigador[nombre=INV6, capacidad=4, especialidad=3]
INVESTIGADOR INV7: Investigador[nombre=INV7, capacidad=1, especialidad=2]
INVESTIGADOR INV8: Investigador[nombre=INV8, capacidad=30, especialidad=3]
Suma de las calidades de los trabajos realizados: 25
--------------------------------------------------------------------------------
Solucion A*: Reparto de horas:
INVESTIGADOR INV1: Investigador[nombre=INV1, capacidad=1, especialidad=2]
INVESTIGADOR INV2: Investigador[nombre=INV2, capacidad=10, especialidad=1]
INVESTIGADOR INV3: Investigador[nombre=INV3, capacidad=3, especialidad=0]
INVESTIGADOR INV4: Investigador[nombre=INV4, capacidad=4, especialidad=0]
INVESTIGADOR INV5: Investigador[nombre=INV5, capacidad=10, especialidad=3]
INVESTIGADOR INV6: Investigador[nombre=INV6, capacidad=4, especialidad=3]
INVESTIGADOR INV7: Investigador[nombre=INV7, capacidad=1, especialidad=2]
INVESTIGADOR INV8: Investigador[nombre=INV8, capacidad=30, especialidad=3]
Suma de las calidades de los trabajos realizados: 25
--------------------------------------------------------------------------------
Solucion PDR: Reparto de horas:
INVESTIGADOR INV1: Investigador[nombre=INV1, capacidad=1, especialidad=2]
INVESTIGADOR INV2: Investigador[nombre=INV2, capacidad=10, especialidad=1]
INVESTIGADOR INV3: Investigador[nombre=INV3, capacidad=3, especialidad=0]
INVESTIGADOR INV4: Investigador[nombre=INV4, capacidad=4, especialidad=0]
INVESTIGADOR INV5: Investigador[nombre=INV5, capacidad=10, especialidad=3]
INVESTIGADOR INV6: Investigador[nombre=INV6, capacidad=4, especialidad=3]
INVESTIGADOR INV7: Investigador[nombre=INV7, capacidad=1, especialidad=2]
INVESTIGADOR INV8: Investigador[nombre=INV8, capacidad=30, especialidad=3]
Suma de las calidades de los trabajos realizados: 25
--------------------------------------------------------------------------------
Solucion BT: Reparto de horas:
INVESTIGADOR INV1: Investigador[nombre=INV1, capacidad=1, especialidad=2]
INVESTIGADOR INV2: Investigador[nombre=INV2, capacidad=10, especialidad=1]
INVESTIGADOR INV3: Investigador[nombre=INV3, capacidad=3, especialidad=0]
INVESTIGADOR INV4: Investigador[nombre=INV4, capacidad=4, especialidad=0]
INVESTIGADOR INV5: Investigador[nombre=INV5, capacidad=10, especialidad=3]
INVESTIGADOR INV6: Investigador[nombre=INV6, capacidad=4, especialidad=3]
INVESTIGADOR INV7: Investigador[nombre=INV7, capacidad=1, especialidad=2]
INVESTIGADOR INV8: Investigador[nombre=INV8, capacidad=30, especialidad=3]
Suma de las calidades de los trabajos realizados: 25
--------------------------------------------------------------------------------
================================================================================
```


TestEjercicio4.java

```java
package ejercicios.tests;

import java.util.List;
import java.util.function.Predicate;

import _datos.DatosEjercicio4;
import _soluciones.SolucionEjercicio4;
import _utils.GraphsPI5;
import _utils.TestsPI5;
import ejercicios.ejercicio4.Ejercicio4Vertex;

//Clase para todos los tests del ejemplo 3 mediante Greedy, A*, PDR y BT
public class TestsEjercicio4 {


	public static void main(String[] args) {
		List.of(1,2).forEach(num_test -> {
			TestsPI5.iniTest("Ejercicio4DatosEntrada", num_test,
DatosEjercicio4::iniDatos);

			// TODO Defina un m. factoria para el vertice inicial
			Ejercicio4Vertex v_inicial = Ejercicio4Vertex.initial();
			// TODO Defina un m. static para los vertices finales
			Predicate<Ejercicio4Vertex> es_terminal = Ejercicio4Vertex.goal();

			var gp = TestsPI5.testGreedy(GraphsPI5.greedyEjercicio4Graph(v_inicial,
es_terminal));
			TestsPI5.toConsole("Voraz", gp, SolucionEjercicio4::of);

			var path = TestsPI5.testAStar(GraphsPI5.Ejercicio4Graph(v_inicial,
es_terminal), gp);
			TestsPI5.toConsole("A*", path, SolucionEjercicio4::of);

			path = TestsPI5.testPDR(GraphsPI5.Ejercicio4Graph(v_inicial, es_terminal),
gp);
			TestsPI5.toConsole("PDR", path, SolucionEjercicio4::of);
```

```
                path = TestsPI5.testBT(GraphsPI5.Ejercicio4Graph(v_inicial, es_terminal),
gp);
                TestsPI5.toConsole("BT", path, SolucionEjercicio4::of);

                TestsPI5.line("*");
        });
    }
```

```
[Cliente[id=0, beneficio=0.0], Cliente[id=1, beneficio=400.0], Cliente[id=2, beneficio=300.0], Cliente[id=3, beneficio=200.0], Cliente[id=4, beneficio=100.0]]
[Conexion[id=0, distancia=1.0], Conexion[id=1, distancia=100.0], Conexion[id=2, distancia=1.0], Conexion[id=3, distancia=100.0], Conexion[id=4, distancia=1.0], Conexion[id=5, distancia=1.0], Conexion[id=6, distancia=100.0], Conexion[id=7, dis
================================================================================================
Solucion Voraz: Camino a seguir:
[0, 1, 2, 3, 4, 0]
Distancia: 9.0
Beneficio: 981.0
________________________________________________________________________________________________
Solucion A*: Camino a seguir:
[0, 1, 2, 3, 4, 0]
Distancia: 9.0
Beneficio: 981.0
________________________________________________________________________________________________
Solucion PDR: Camino a seguir:
[0, 1, 2, 3, 4, 0]
Distancia: 9.0
Beneficio: 981.0
________________________________________________________________________________________________
Solucion BT: Camino a seguir:
[0, 1, 2, 3, 4, 0]
Distancia: 9.0
Beneficio: 981.0
************************************************************************************************
[Cliente[id=0, beneficio=0.0], Cliente[id=1, beneficio=100.0], Cliente[id=2, beneficio=200.0], Cliente[id=3, beneficio=300.0], Cliente[id=4, beneficio=200.0], Cliente[id=5, beneficio=300.0], Cliente[id=6, beneficio=200.0], Cliente[id=7, benef
[Conexion[id=8, distancia=2.0], Conexion[id=9, distancia=1.0], Conexion[id=10, distancia=1.0], Conexion[id=11, distancia=3.0], Conexion[id=12, distancia=1.0], Conexion[id=13, distancia=1.0], Conexion[id=14, distancia=3.0], Conexion[id=15, dis
================================================================================================
Solucion Voraz: Camino a seguir:
[0, 2, 5, 3, 7, 4, 6, 1, 0]
Distancia: 9.0
Beneficio: 1463.0
________________________________________________________________________________________________
Solucion A*: Camino a seguir:
[0, 2, 5, 3, 7, 4, 6, 1, 0]
Distancia: 9.0
Beneficio: 1463.0
________________________________________________________________________________________________
Solucion PDR: Camino a seguir:
[0, 2, 5, 3, 7, 4, 6, 1, 0]
Distancia: 9.0
Beneficio: 1463.0
________________________________________________________________________________________________
Solucion BT: Camino a seguir:
[0, 2, 5, 3, 7, 4, 6, 1, 0]
Distancia: 9.0
Beneficio: 1463.0
************************************************************************************************
}
```

Manual

EM1.java

```
package ejercicios.tests.manual;

import java.util.List;

import _datos.DatosEjercicio1;
import _utils.TestsPI5;
import ejercicios.ejercicio1.manual.Ejercicio1PDR;
import us.lsi.common.String2;

public class TestsEM1 {

    public static void main(String[] args) {
        List.of(1,2,3).forEach(num_test -> {

    DatosEjercicio1.iniDatos("ficheros/Ejercicio1DatosEntrada"+num_test+".txt");
            String2.toConsole("Solucion obtenida: %s\n", Ejercicio1PDR.search());
            TestsPI5.line("*");
        });
    }

}
```

```
[Tipo[Nombre_Tipo=C01, kgdisponibles=5, id=1], Tipo[Nombre_Tipo=C02, kgdisponibles=4, id=2], Tipo[Nombre_Tipo=C03, kgdisponibles=1, id=3], Tipo[Nombre_Tipo=C04, kgdisponibles=2, id=4], Tipo[Nombre_Tipo=C05, kgdisponibles=8, id=5], Tipo
[Nombre_Tipo=C06, kgdisponibles=1, id=6]]
[Variedad[nombre=P01, id=null, beneficio=20.0, porcentaje={C02=0.4, C01=0.5, C03=0.1}], Variedad[nombre=P02, id=null, beneficio=10.0, porcentaje={C04=0.2, C05=0.8}], Variedad[nombre=P03, id=null, beneficio=5.0, porcentaje={C06=1.0}]]
Solucion obtenida: Solucion = [10, 10, 1]; Tamaño solucion = 3; beneficio total = 305,000000;

**********
[Tipo[Nombre_Tipo=C01, kgdisponibles=11, id=1], Tipo[Nombre_Tipo=C02, kgdisponibles=9, id=2], Tipo[Nombre_Tipo=C03, kgdisponibles=7, id=3], Tipo[Nombre_Tipo=C04, kgdisponibles=12, id=4], Tipo[Nombre_Tipo=C05, kgdisponibles=6, id=5]]
[Variedad[nombre=P01, id=null, beneficio=20.0, porcentaje={C02=0.4, C01=0.2, C05=0.4}], Variedad[nombre=P02, id=null, beneficio=10.0, porcentaje={C02=0.3, C03=0.7}], Variedad[nombre=P03, id=null, beneficio=80.0, porcentaje={C01=0.4, C04=0.6}]]
Solucion obtenida: Solucion = [15, 10, 20]; Tamaño solucion = 3; beneficio total = 2000,000000;

**********
[Tipo[Nombre_Tipo=C01, kgdisponibles=35, id=1], Tipo[Nombre_Tipo=C02, kgdisponibles=4, id=2], Tipo[Nombre_Tipo=C03, kgdisponibles=12, id=3], Tipo[Nombre_Tipo=C04, kgdisponibles=5, id=4], Tipo[Nombre_Tipo=C05, kgdisponibles=30, id=5], Tipo
[Nombre_Tipo=C06, kgdisponibles=42, id=6], Tipo[Nombre_Tipo=C07, kgdisponibles=3, id=7], Tipo[Nombre_Tipo=C08, kgdisponibles=2, id=8], Tipo[Nombre_Tipo=C09, kgdisponibles=20, id=9], Tipo[Nombre_Tipo=C10, kgdisponibles=3, id=10]]
[Variedad[nombre=P01, id=null, beneficio=60.0, porcentaje={C01=0.5, C03=0.4, C07=0.1}], Variedad[nombre=P02, id=null, beneficio=25.0, porcentaje={C02=1.0}], Variedad[nombre=P03, id=null, beneficio=5.0, porcentaje={C02=0.4, C07=0.8}], Variedad
[nombre=P04, id=null, beneficio=25.0, porcentaje={C10=0.2, C06=0.8}], Variedad[nombre=P05, id=null, beneficio=15.0, porcentaje={C03=0.4, C08=0.6}], Variedad[nombre=P06, id=null, beneficio=100.0, porcentaje={C01=0.2, C06=0.3, C05=0.3,
C09=0.2}]]
Solucion obtenida: Solucion = [29, 3, 1, 15, 2, 100]; Tamaño solucion = 6; beneficio total = 12225,000000;

**********
```

EM2.java

```
package ejercicios.tests.manual;

import java.util.List;

import _datos.DatosEjercicio2;
import _utils.TestsPI5;
import ejercicios.ejercicio2.manual.Ejercicio2PDR;
import us.lsi.common.String2;

public class TestsEM2 {

	public static void main(String[] args) {
		List.of(3).forEach(num_test -> {

	DatosEjercicio2.iniDatos("ficheros/Ejercicio2DatosEntrada"+num_test+".txt");
			String2.toConsole("Solucion obtenida: %s\n", Ejercicio2PDR.search());
			TestsPI5.line("*");
		});
	}

}
```

```
[Curso[id=0, tematicas=[1, 2, 3, 4], coste=10.0, centro=0], Curso[id=1, tematicas=[1, 4], coste=3.0, centro=0], Curso[id=2, tematicas=[5], coste=1.5, centro=1], Curso[id=3, tematicas=[5], coste=5.0, centro=0]]
Max_Centros = 1
Solucion obtenida: Cursos elegidos: {S0, S3}
Coste Total: 15,0

**********
```

```
[Curso[id=0, tematicas=[2, 3], coste=2.0, centro=0], Curso[id=1, tematicas=[4], coste=3.0, centro=0], Curso[id=2, tematicas=[1, 5], coste=5.0, centro=0], Curso[id=3, tematicas=[1, 3, 4], coste=3.5, centro=2], Curso[id=4, tematicas=[4, 5],
coste=1.5, centro=1]]
Max_Centros = 2
Solucion obtenida: Cursos elegidos: {S0, S2, S4}
Coste Total: 8,5

**********
```

```
[Curso[id=0, tematicas=[2, 6, 7], coste=2.0, centro=2], Curso[id=1, tematicas=[7], coste=3.0, centro=0], Curso[id=2, tematicas=[1, 5], coste=5.0, centro=0], Curso[id=3, tematicas=[1, 3, 4], coste=3.5, centro=2], Curso[id=4, tematicas=[3, 7],
coste=1.5, centro=1], Curso[id=5, tematicas=[4, 5, 6], coste=4.5, centro=0], Curso[id=6, tematicas=[5, 6], coste=6.0, centro=1], Curso[id=7, tematicas=[2, 3, 5], coste=1.0, centro=1]]
Max_Centros = 3
Solucion obtenida: Cursos elegidos: {S0, S3, S7}
Coste Total: 6,5

**********
```

EM3.java

```
package ejercicios.tests.manual;

import java.util.List;

import _datos.DatosEjercicio3;
import _utils.TestsPI5;
import ejercicios.ejercicio3.manual.Ejercicio3BT;
import us.lsi.common.String2;

public class TestsEM3 {

	public static void main(String[] args) {
		List.of(1,2,3).forEach(num_test -> {

	DatosEjercicio3.iniDatos("ficheros/Ejercicio3DatosEntrada"+num_test+".txt");
			Ejercicio3BT.search();
```

```
                        Ejercicio3BT.getSoluciones().forEach(s -> String2.toConsole("Solucion
obtenida: %s\n", s));
                        TestsPI5.line("*");
                });
        }

}
```

```
[Investigador[nombre=INV1, capacidad=6, especialidad=0], Investigador[nombre=INV2, capacidad=3, especialidad=1], Investigador[nombre=INV3, capacidad=8, especialidad=2]]
[Trabajo[nombre=T01, calidad=5, reparto={0=6, 1=0, 2=0}], Trabajo[nombre=T02, calidad=10, reparto={0=0, 1=3, 2=8}]]
Solucion obtenida: Reparto de horas:
INVESTIGADOR INV1: Investigador[nombre=INV1, capacidad=6, especialidad=0]
INVESTIGADOR INV2: Investigador[nombre=INV2, capacidad=3, especialidad=1]
INVESTIGADOR INV3: Investigador[nombre=INV3, capacidad=8, especialidad=2]
Suma de las calidades de los trabajos realizados: 5

**********
[Investigador[nombre=INV1, capacidad=10, especialidad=0], Investigador[nombre=INV2, capacidad=5, especialidad=1], Investigador[nombre=INV3, capacidad=8, especialidad=2], Investigador[nombre=INV4, capacidad=2, especialidad=0], Investigador
[nombre=INV5, capacidad=5, especialidad=3]]
[Trabajo[nombre=T01, calidad=7, reparto={0=2, 1=0, 2=5, 3=0}], Trabajo[nombre=T02, calidad=9, reparto={0=8, 1=4, 2=3, 3=0}], Trabajo[nombre=T03, calidad=5, reparto={0=2, 1=0, 2=0, 3=7}]]
Solucion obtenida: Reparto de horas:
INVESTIGADOR INV1: Investigador[nombre=INV1, capacidad=10, especialidad=0]
INVESTIGADOR INV2: Investigador[nombre=INV2, capacidad=5, especialidad=1]
INVESTIGADOR INV3: Investigador[nombre=INV3, capacidad=8, especialidad=2]
INVESTIGADOR INV4: Investigador[nombre=INV4, capacidad=2, especialidad=0]
INVESTIGADOR INV5: Investigador[nombre=INV5, capacidad=5, especialidad=3]
Suma de las calidades de los trabajos realizados: 0
```

EM4.java

```
package ejercicios.tests.manual;

import java.util.List;
import _datos.DatosEjercicio4;
import _utils.TestsPI5;
import ejercicios.ejercicio4.manual.Ejercicio4BT;
import us.lsi.common.String2;

public class TestsEM4 {

        public static void main(String[] args) {
                List.of(1,2).forEach(num_test -> {

        DatosEjercicio4.iniDatos("ficheros/Ejercicio4DatosEntrada"+num_test+".txt");
                        Ejercicio4BT.search();
                        Ejercicio4BT.getSoluciones().forEach(s -> String2.toConsole("Solucion
obtenida: %s\n", s));
                        TestsPI5.line("*");
                });
        }

}
```

```
[Cliente[id=0, beneficio=0.0], Cliente[id=1, beneficio=400.0], Cliente[id=2, beneficio=300.0], Cliente[id=3, beneficio=200.0], Cliente[id=4, beneficio=100.0]]
[Conexion[id=0, distancia=1.0], Conexion[id=1, distancia=100.0], Conexion[id=2, distancia=1.0], Conexion[id=3, distancia=100.0], Conexion[id=4, distancia=1.0], Conexion[id=5, distancia=1.0], Conexion[id=6, distancia=100.0], Conexion[id=7,
distancia=5.0]]
Solucion obtenida: Camino a seguir:
[0, 1, 3, 2, 4, 0]
Distancia: 207.0
Beneficio: 387.0

Solucion obtenida: Camino a seguir:
[0, 1, 2, 3, 4, 0]
Distancia: 9.0
Beneficio: 981.0

**********
[Cliente[id=0, beneficio=0.0], Cliente[id=1, beneficio=100.0], Cliente[id=2, beneficio=200.0], Cliente[id=3, beneficio=300.0], Cliente[id=4, beneficio=200.0], Cliente[id=5, beneficio=300.0], Cliente[id=6, beneficio=200.0], Cliente[id=7,
beneficio=200.0]]
[Conexion[id=8, distancia=2.0], Conexion[id=9, distancia=1.0], Conexion[id=10, distancia=1.0], Conexion[id=11, distancia=3.0], Conexion[id=12, distancia=1.0], Conexion[id=13, distancia=1.0], Conexion[id=14, distancia=3.0], Conexion[id=15,
distancia=1.0], Conexion[id=16, distancia=1.0], Conexion[id=17, distancia=3.0], Conexion[id=18, distancia=1.0], Conexion[id=19, distancia=1.0], Conexion[id=20, distancia=1.0]]
Solucion obtenida: Camino a seguir:
[0, 1, 2, 7, 3, 5, 6, 4, 0]
Distancia: 13.0
Beneficio: 1438.0

**********
```

Voraces

TestEjemplos.java

```
package voraces;

import java.util.List;
import java.util.function.Predicate;

import _datos.DatosEjercicio1;
import _datos.DatosEjercicio2;
import _datos.DatosEjercicio3;
import _datos.DatosEjercicio4;
import _soluciones.SolucionEjercicio1;
import _soluciones.SolucionEjercicio2;
import _soluciones.SolucionEjercicio3;
import _soluciones.SolucionEjercicio4;
import ejercicios.ejercicio1.Ejercicio1Vertex;
import ejercicios.ejercicio2.Ejercicio2Vertex;
import ejercicios.ejercicio3.Ejercicio3Vertex;
import ejercicios.ejercicio4.Ejercicio4Vertex;
import us.lsi.common.List2;
import us.lsi.common.String2;

// Voraces de forma manual para los ejemplos. Soluciones iterativas y funcionales
public class TestsEjemplos {

	public static void main(String[] args) {
//		testVorazE1();

//		testVorazE2_Fichero1();
//		testVorazE2_Fichero2();
//		testVorazE2_Fichero3();

//		testVorazE4();

	}

	// Ejemplo1: Voraz Iterativo Manual. Iterando sobre vertices
	private static void testVorazE1() {
		List.of(1,2,3).forEach(num_test -> {

	DatosEjercicio1.iniDatos("ficheros/Ejercicio1DatosEntrada"+num_test+".txt");

			List<Integer> path = List2.empty();

			Ejercicio1Vertex v = Ejercicio1Vertex.initial();
			Predicate<Ejercicio1Vertex> last = Ejercicio1Vertex.goal();
			Predicate<Ejercicio1Vertex> solution = Ejercicio1Vertex.goalHasSolution();
			while(!solution.test(v) && !last.test(v)) {
				var e = v.greedyEdge();
				path.add(e.action());
				v = e.target();
			}
			String2.toConsole("Voraz Manual: %s\n%s",
			SolucionEjercicio1.of_Range(path),String2.linea());
		});
	}
```

```
[Tipo[Nombre_Tipo=C01, kgdisponibles=5, id=1], Tipo[Nombre_Tipo=C02, kgdisponibles=4, id=2], Tipo[Nombre_Tipo=C03, kgdisponibles=1, id=3], Tipo[Nombre_Tipo=C04, kgdisponibles=2, id=4], Tipo[Nombre_Tipo=C05, kgdisponibles=8, id=5], Tipo
[Nombre_Tipo=C06, kgdisponibles=1, id=6]]
[Variedad[nombre=P01, id=null, beneficio=20.0, porcentaje={C02=0.4, C01=0.5, C03=0.1}], Variedad[nombre=P02, id=null, beneficio=10.0, porcentaje={C04=0.2, C05=0.8}], Variedad[nombre=P03, id=null, beneficio=5.0, porcentaje={C06=1.0}]]
Voraz Manual: Solucion = []; Tamaño solucion = 0; beneficio total = 0,000000;
_____
[Tipo[Nombre_Tipo=C01, kgdisponibles=11, id=1], Tipo[Nombre_Tipo=C02, kgdisponibles=9, id=2], Tipo[Nombre_Tipo=C03, kgdisponibles=7, id=3], Tipo[Nombre_Tipo=C04, kgdisponibles=12, id=4], Tipo[Nombre_Tipo=C05, kgdisponibles=6, id=5]]
[Variedad[nombre=P01, id=null, beneficio=20.0, porcentaje={C02=0.4, C01=0.2, C05=0.4}], Variedad[nombre=P02, id=null, beneficio=10.0, porcentaje={C02=0.3, C03=0.7}], Variedad[nombre=P03, id=null, beneficio=80.0, porcentaje={C01=0.4, C04=0.6}]]
Voraz Manual: Solucion = []; Tamaño solucion = 0; beneficio total = 0,000000;
_____
[Tipo[Nombre_Tipo=C01, kgdisponibles=35, id=1], Tipo[Nombre_Tipo=C02, kgdisponibles=4, id=2], Tipo[Nombre_Tipo=C03, kgdisponibles=12, id=3], Tipo[Nombre_Tipo=C04, kgdisponibles=5, id=4], Tipo[Nombre_Tipo=C05, kgdisponibles=30, id=5], Tipo
[Nombre_Tipo=C06, kgdisponibles=42, id=6], Tipo[Nombre_Tipo=C07, kgdisponibles=3, id=7], Tipo[Nombre_Tipo=C08, kgdisponibles=2, id=8], Tipo[Nombre_Tipo=C09, kgdisponibles=20, id=9], Tipo[Nombre_Tipo=C10, kgdisponibles=3, id=10]]
[Variedad[nombre=P01, id=null, beneficio=60.0, porcentaje={C01=0.5, C03=0.4, C07=0.1}], Variedad[nombre=P02, id=null, beneficio=25.0, porcentaje={C02=1.0}], Variedad[nombre=P03, id=null, beneficio=5.0, porcentaje={C02=0.4, C07=0.8}], Variedad
[nombre=P04, id=null, beneficio=25.0, porcentaje={C10=0.2, C06=0.8}], Variedad[nombre=P05, id=null, beneficio=15.0, porcentaje={C03=0.4, C08=0.6}], Variedad[nombre=P06, id=null, beneficio=100.0, porcentaje={C01=0.2, C06=0.3, C05=0.3,
C09=0.2}]]
Voraz Manual: Solucion = []; Tamaño solucion = 0; beneficio total = 0,000000;
_____
```

```
private static void testVorazE2_Fichero1() {
		DatosEjercicio2.iniDatos("ficheros/Ejercicio2DatosEntrada1.txt");
		List<Integer> path = List2.empty();

		Ejercicio2Vertex v = Ejercicio2Vertex.initial();
		Predicate<Ejercicio2Vertex> last = Ejercicio2Vertex.goal();
		Predicate<Ejercicio2Vertex> solution = Ejercicio2Vertex.goalHasSolution();
		while(!solution.test(v) && !last.test(v)) {
			var e = v.greedyEdge();
			path.add(e.action());
			v = e.target();
		}
		String2.toConsole("Voraz Manual: %s\n%s",
		SolucionEjercicio2.of_Range(path),String2.linea());
}
```

```
[Curso[id=0, tematicas=[1, 2, 3, 4], coste=10.0, centro=0], Curso[id=1, tematicas=[1, 4], coste=3.0, centro=0], Curso[id=2, tematicas=[5], coste=1.5, centro=1], Curso[id=3, tematicas=[5], coste=5.0, centro=0]]
Max_Centros = 1
Voraz Manual: Cursos elegidos: {S0, S3}
Coste Total: 15,0
_____
```

```
private static void testVorazE2_Fichero2() {
	DatosEjercicio2.iniDatos("ficheros/Ejercicio2DatosEntrada2.txt");
	List<Integer> path = List2.empty();

	Ejercicio2Vertex v = Ejercicio2Vertex.initial();
	Predicate<Ejercicio2Vertex> last = Ejercicio2Vertex.goal();
	Predicate<Ejercicio2Vertex> solution = Ejercicio2Vertex.goalHasSolution();
	while(!solution.test(v) && !last.test(v)) {
		var e = v.greedyEdge();
		path.add(e.action());
		v = e.target();
	}
	String2.toConsole("Voraz Manual: %s\n%s",
			SolucionEjercicio2.of_Range(path),String2.linea());
}
```

```
[Curso[id=0, tematicas=[2, 3], coste=2.0, centro=0], Curso[id=1, tematicas=[4], coste=3.0, centro=0], Curso[id=2, tematicas=[1, 5], coste=5.0, centro=0], Curso[id=3, tematicas=[1, 3, 4], coste=3.5, centro=2], Curso[id=4, tematicas=[4, 5],
coste=1.5, centro=1]]
Max_Centros = 2
Voraz Manual: Cursos elegidos: {S0, S1, S2}
Coste Total: 10,0
_____
```

```
private static void testVorazE2_Fichero3() {
	DatosEjercicio2.iniDatos("ficheros/Ejercicio2DatosEntrada3.txt");
	List<Integer> path = List2.empty();

	Ejercicio2Vertex v = Ejercicio2Vertex.initial();
	Predicate<Ejercicio2Vertex> last = Ejercicio2Vertex.goal();
	Predicate<Ejercicio2Vertex> solution = Ejercicio2Vertex.goalHasSolution();
	while(!solution.test(v) && !last.test(v)) {
		var e = v.greedyEdge();
		path.add(e.action());
		v = e.target();
	}
```

```
		String2.toConsole("Voraz Manual: %s\n%s",
				SolucionEjercicio2.of_Range(path),String2.linea());
	}
```

```
[Curso[id=0, tematicas=[2, 6, 7], coste=2.0, centro=2], Curso[id=1, tematicas=[7], coste=3.0, centro=0], Curso[id=2, tematicas=[1, 5], coste=5.0, centro=0], Curso[id=3, tematicas=[1, 3, 4], coste=3.5, centro=2], Curso[id=4, tematicas=[3, 7], coste=1.5, centro=1], Curso[id=5, tematicas=[4, 5, 6], coste=4.5, centro=0], Curso[id=6, tematicas=[5, 6], coste=6.0, centro=1], Curso[id=7, tematicas=[2, 3, 5], coste=1.0, centro=1]]
Max_Centros = 3
Voraz Manual: Cursos elegidos: {S0, S2, S3}
Coste Total: 10,5
```

```
	private static void testVorazE4() {
		List.of(1,2).forEach(num_test -> {

	DatosEjercicio4.iniDatos("ficheros/Ejercicio4DatosEntrada"+num_test+".txt");

			List<Integer> path = List2.empty();

			Ejercicio4Vertex v = Ejercicio4Vertex.initial();
			Predicate<Ejercicio4Vertex> last = Ejercicio4Vertex.goal();
			Predicate<Ejercicio4Vertex> solution = Ejercicio4Vertex.goalHasSolution();
			while(!solution.test(v) && !last.test(v)) {
				var e = v.greedyEdge();
				path.add(e.action());
				v = e.target();
			}
			String2.toConsole("Voraz Manual: %s\n%s",
			SolucionEjercicio4.of_Range(path),String2.linea());
		});
	}
}
```

```
[Cliente[id=0, beneficio=0.0], Cliente[id=1, beneficio=400.0], Cliente[id=2, beneficio=300.0], Cliente[id=3, beneficio=200.0], Cliente[id=4, beneficio=100.0]]
[Conexion[id=0, distancia=1.0], Conexion[id=1, distancia=100.0], Conexion[id=2, distancia=1.0], Conexion[id=3, distancia=100.0], Conexion[id=4, distancia=1.0], Conexion[id=5, distancia=1.0], Conexion[id=6, distancia=100.0], Conexion[id=7, distancia=5.0]]
Voraz Manual: Camino a seguir:
[0, 1, 2, 3, 4, 0]
Distancia: 9.0
Beneficio: 981.0
```

```
[Cliente[id=0, beneficio=0.0], Cliente[id=1, beneficio=100.0], Cliente[id=2, beneficio=200.0], Cliente[id=3, beneficio=300.0], Cliente[id=4, beneficio=200.0], Cliente[id=5, beneficio=300.0], Cliente[id=6, beneficio=200.0], Cliente[id=7, beneficio=200.0]]
[Conexion[id=8, distancia=2.0], Conexion[id=9, distancia=1.0], Conexion[id=10, distancia=1.0], Conexion[id=11, distancia=3.0], Conexion[id=12, distancia=1.0], Conexion[id=13, distancia=1.0], Conexion[id=14, distancia=3.0], Conexion[id=15, distancia=1.0], Conexion[id=16, distancia=1.0], Conexion[id=17, distancia=3.0], Conexion[id=18, distancia=1.0], Conexion[id=19, distancia=1.0], Conexion[id=20, distancia=1.0]]
Voraz Manual: Camino a seguir:
[0, 2, 5, 3, 7, 4, 6, 1, 0]
Distancia: 9.0
Beneficio: 1463.0
```

Utils

GraphsPI5.java

```
package _utils;

import java.util.function.Predicate;

import ejercicios.ejercicio1.Ejercicio1Edge;
import ejercicios.ejercicio1.Ejercicio1Heuristic;
import ejercicios.ejercicio1.Ejercicio1Vertex;
import ejercicios.ejercicio2.Ejercicio2Edge;
import ejercicios.ejercicio2.Ejercicio2Heuristic;
import ejercicios.ejercicio2.Ejercicio2Vertex;
import ejercicios.ejercicio3.Ejercicio3Edge;
import ejercicios.ejercicio3.Ejercicio3Heuristic;
import ejercicios.ejercicio3.Ejercicio3Vertex;
import ejercicios.ejercicio4.Ejercicio4Edge;
import ejercicios.ejercicio4.Ejercicio4Heuristic;
import ejercicios.ejercicio4.Ejercicio4Vertex;
import us.lsi.graphs.virtual.EGraph;
import us.lsi.graphs.virtual.EGraph.Type;
import us.lsi.path.EGraphPath.PathType;

// Clase Factoria para crear los grafos de los ejemplos y ejercicios
public class GraphsPI5 {

	// Ejercicio1: Grafo NO Greedy
	public static EGraph<Ejercicio1Vertex, Ejercicio1Edge>
	Ejercicio1Graph(Ejercicio1Vertex v_inicial, Predicate<Ejercicio1Vertex> es_terminal) {
		return EGraph.virtual(v_inicial, es_terminal, PathType.Sum, Type.Max)
				.goalHasSolution(Ejercicio1Vertex.goalHasSolution())
				.heuristic(Ejercicio1Heuristic::heuristic).build();
	}

	// Ejercicio1: Grafo Greedy
	public static EGraph<Ejercicio1Vertex, Ejercicio1Edge>
	greedyEjercicio1Graph(Ejercicio1Vertex v_inicial, Predicate<Ejercicio1Vertex>
es_terminal) {
		return EGraph.virtual(v_inicial,es_terminal,PathType.Sum, Type.Max)
				.greedyEdge(Ejercicio1Vertex::greedyEdge)
				.goalHasSolution(Ejercicio1Vertex.goalHasSolution())
				.heuristic(Ejercicio1Heuristic::heuristic).build();
	}

	// Ejercicio2: Grafo NO Greedy
	public static EGraph<Ejercicio2Vertex, Ejercicio2Edge>
	Ejercicio2Graph(Ejercicio2Vertex v_inicial, Predicate<Ejercicio2Vertex> es_terminal) {
		return EGraph.virtual(v_inicial, es_terminal, PathType.Sum, Type.Min)
				.goalHasSolution(Ejercicio2Vertex.goalHasSolution())
				.heuristic(Ejercicio2Heuristic::heuristic).build();
	}

	// Ejercicio2: Grafo Greedy
	public static EGraph<Ejercicio2Vertex, Ejercicio2Edge>
	greedyEjercicio2Graph(Ejercicio2Vertex v_inicial, Predicate<Ejercicio2Vertex>
es_terminal) {
		return EGraph.virtual(v_inicial,es_terminal,PathType.Sum, Type.Min)
				.greedyEdge(Ejercicio2Vertex::greedyEdge)
				.goalHasSolution(Ejercicio2Vertex.goalHasSolution())
				.heuristic(Ejercicio2Heuristic::heuristic).build();
```

```
	}

	// Ejercicio3: Grafo NO Greedy
	public static EGraph<Ejercicio3Vertex, Ejercicio3Edge>
	Ejercicio3Graph(Ejercicio3Vertex v_inicial, Predicate<Ejercicio3Vertex> es_terminal) {
		return EGraph.virtual(v_inicial, es_terminal, PathType.Sum, Type.Max)
				.goalHasSolution(Ejercicio3Vertex.goalHasSolution())
				.heuristic(Ejercicio3Heuristic::heuristic).build();
	}

	// Ejercicio3: Grafo Greedy
	public static EGraph<Ejercicio3Vertex, Ejercicio3Edge>
	greedyEjercicio3Graph(Ejercicio3Vertex v_inicial, Predicate<Ejercicio3Vertex>
es_terminal) {
		return EGraph.virtual(v_inicial,es_terminal,PathType.Sum, Type.Max)
				.greedyEdge(Ejercicio3Vertex::greedyEdge)
				.goalHasSolution(Ejercicio3Vertex.goalHasSolution())
				.heuristic(Ejercicio3Heuristic::heuristic).build();
	}

	// Ejercicio4: Grafo NO Greedy
	public static EGraph<Ejercicio4Vertex, Ejercicio4Edge>
	Ejercicio4Graph(Ejercicio4Vertex v_inicial, Predicate<Ejercicio4Vertex> es_terminal) {
		return EGraph.virtual(v_inicial, es_terminal, PathType.Sum, Type.Max)
				.goalHasSolution(Ejercicio4Vertex.goalHasSolution())
				.heuristic(Ejercicio4Heuristic::heuristic).build();
	}

	// Ejercicio4: Grafo Greedy
	public static EGraph<Ejercicio4Vertex, Ejercicio4Edge>
	greedyEjercicio4Graph(Ejercicio4Vertex v_inicial, Predicate<Ejercicio4Vertex>
es_terminal) {
		return EGraph.virtual(v_inicial,es_terminal,PathType.Sum, Type.Max)
				.greedyEdge(Ejercicio4Vertex::greedyEdge)
				.goalHasSolution(Ejercicio4Vertex.goalHasSolution())
				.heuristic(Ejercicio4Heuristic::heuristic).build();
	}
}
```

TestsPI.java

```
package _utils;

import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

import org.jgrapht.Graph;
import org.jgrapht.GraphPath;

import us.lsi.colors.GraphColors;
import us.lsi.colors.GraphColors.Style;
import us.lsi.common.String2;
import us.lsi.graphs.alg.AStar;
import us.lsi.graphs.alg.BT;
import us.lsi.graphs.alg.DPR;
import us.lsi.graphs.alg.GreedyOnGraph;
import us.lsi.graphs.virtual.EGraph;
```

```
// Reune y simplifica todos los tests.
public class TestsPI5 {

	public static<V,E> GraphPath<V, E> testGreedy(EGraph<V, E> graph) {
		var alg_voraz = GreedyOnGraph.of(graph);
		GraphPath<V, E> path = alg_voraz.path();
		return alg_voraz.isSolution(path)? path: null;
	}

	public static<V,E> GraphPath<V, E> testAStar(EGraph<V, E> graph, GraphPath<V, E> gp) {

		var alg_star = gp==null? AStar.of(graph):
			AStar.of(graph, null, gp.getWeight(), gp);
		var res = alg_star.search().orElse(null);
		toDot("AStar", alg_star.outGraph(), res);
		return res;
	}

	public static<V,E> GraphPath<V, E> testPDR(EGraph<V, E> graph, GraphPath<V, E> gp) {
		var alg_pdr = gp==null? DPR.of(graph):
			DPR.of(graph, null, gp.getWeight(), gp, true);

		var res = alg_pdr.search().orElse(null);
		toDot("PDR", alg_pdr.outGraph(), res);
		return res;
	}

	public static<V,E,S> GraphPath<V, E> testBT(EGraph<V, E> graph, GraphPath<V, E> gp) {
		var alg_bt = gp==null? BT.of(graph):
			BT.of(graph, null, gp.getWeight(), gp, true);

		var res = alg_bt.search().orElse(null);
		toDot("BT", alg_bt.outGraph(), res);
		return res;
	}

	public static<V,E> void toConsole(String type, GraphPath<V, E> path,
Function<GraphPath<V, E>, ?> fSolution) {
		if(path!=null)
			String2.toConsole("Solucion %s: %s", type, fSolution.apply(path));
		else
			String2.toConsole("%s no obtuvo solucion", type);

		String2.toConsole(String2.linea());
	}


	private static String file;
	private static Integer num_test;

	public static void iniTest(String f, Integer t, Consumer<String> iniDatos) {
		file = f;
		num_test = t;
		iniDatos.accept("ficheros/"+f+t+".txt");
		line("=");
	}

	public static void line(String s) {
		String2.toConsole(IntStream.range(0, 100).mapToObj(i -
>s).collect(Collectors.joining()));
```

```
	}

	private static <V,E> void toDot(String type, Graph<V,E> g, GraphPath<V,E> sol){
		Predicate<V> vs = v -> sol.getVertexList().contains(v);
		Predicate<E> es = e -> sol.getEdgeList().contains(e);
		GraphColors.toDot(g, "grafos/"+type+"-"+file+num_test+".gv",
				V::toString,
				E::toString,
				v -> GraphColors.styleIf(Style.bold, vs.test(v)),
				e -> GraphColors.styleIf(Style.bold, es.test(e)));
	}

}
```

Problemas

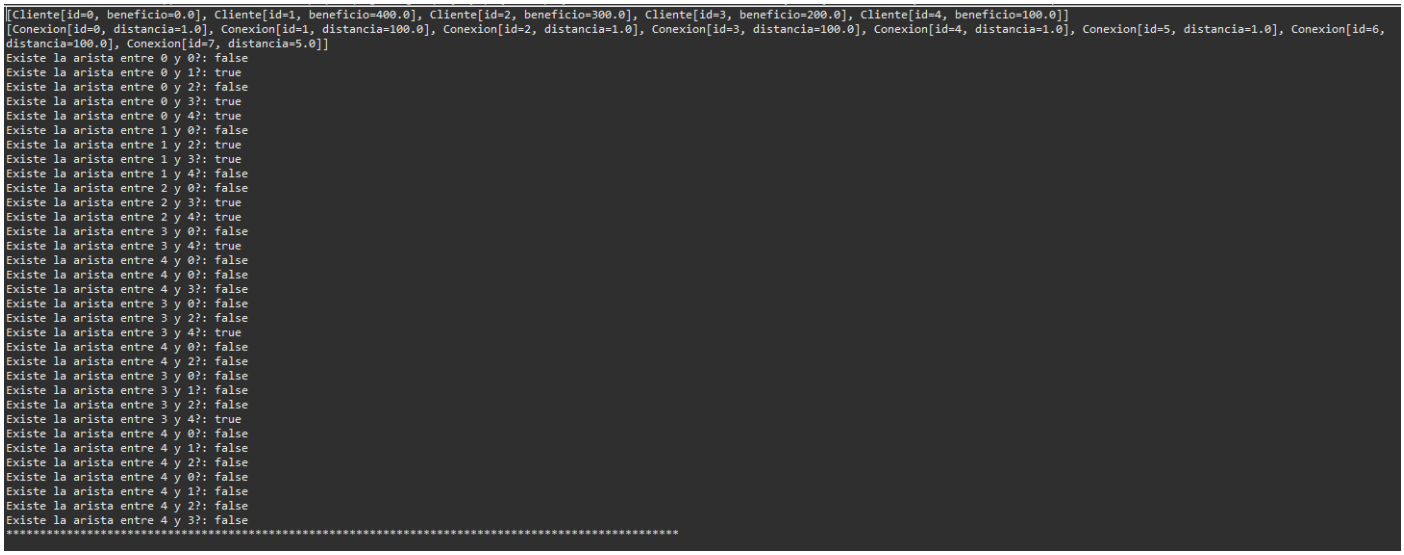
ejercicio1Voraz: ejecuta pero no obtiene valores

ejercicio3BT: ejecuta pero no da la solución correcta. Creo que es por el archivo SolucionEjercicio3.java, ya que con la version automática también me pasaba lo mismo y modificándolo pude resolver el problema.

ejercicio4BT:no me da las soluciones correctas porque no determina bien las adyacencias de los vértices.

- Si el grafo fuese dirigido, el orden de las aristas importa, sin embargo, no lo tengo declarado como dirigido y tampoco he visto ningún documento donde lo especifique.

- He probado a hacer una función auxiliar para calcular las adyacencias teniendo en cuenta el orden, pero no da solución (existeAristaExtricto(Integer x, Integer y))



esta foto muestra como evalúa las conexiones para el primer fichero de datos