

# **MEMORIA**

# **PRÁCTICA INDIVIDUAL 5**

Jaime Linares Barrera

2º Ingeniería Informática - Ingeniería del Software, Grupo 4

# 1. EJERCICIO 1

## 1.1. Modelo

- **Modelo**
- $x_i$ : variable entera, indica cuántos kilogramos de la variedad  $i$  serán fabricados,  $i \in [0, m]$

$$\begin{aligned} & \max \sum_{i=0}^{m-1} b_i x_i \\ & \sum_{i=0}^{m-1} p_{ij} x_i \leq c_j, j \in [0, n] \\ & \text{int } x_i, i \in [0, m] \end{aligned}$$

## 1.2. DatosCafes

```
package _datos;

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

import us.lsi.common.Files2;

public class DatosCafes {

    public static List<Cafe> cafes;
    public static List<Variedad> variedades;

    // Funcion que realiza la lectura de fichero y crea un tipo DatosCafes
    public static void iniDatos(String fichero) {
        List<Cafe> listaCafes = new ArrayList<>();
        List<Variedad> listaVariedades = new ArrayList<>();

        List<String> filas = Files2.linesFromFile(fichero);
        List<String> cafesString = filas.sublist(1, filas.indexOf("// VARIEDADES"));
        List<String> variedadesString = filas.subList(filas.indexOf("// VARIEDADES") + 1, filas.size());

        // Transformamos una cadena en el tipo Cafe y lo añadimos a la lista de cafes
        for(String linea: cafesString) {
            Cafe c = Cafe.parseaCafe(linea);
            listaCafes.add(c);
        }
        cafes = listaCafes;

        // Transformamos una cadena en el tipo Variedad y lo añadimos a la lista de variedades
        for(String linea: variedadesString) {
            Variedad v = Variedad.parseaVariedad(linea);
            listaVariedades.add(v);
        }
        variedades = listaVariedades;

        toConsole();
    }

    // Para mostar el resultado por pantalla
    private static void toConsole() {
        System.out.println("- Tipos de cafe leidos: ");
        for(Cafe c: cafes) {
            System.out.println(" " + c);
        }
        System.out.println("- Tipo de variedades leidas: ");
        for(Variedad v: variedades) {
            System.out.println(" " + v);
        }
    }

    // Funcion que me devuelve los tipos de cafe
    public static List<Cafe> getTiposCafe() {
        return cafes;
    }

    // Funcion que calcula el numero de tipo de cafe
    public static IntegergetNumTiposCafe() {
        return cafes.size();
    }

    // Funcion que me devuelve las variedades
    public static List<Variedad> getVariedades() {
        return variedades;
    }

    // Funcion que calcula el numero de variedades
    public static IntegergetNumVariedades() {
        return variedades.size();
    }

    // Funcion que calcula los kg disponibles de cafe del tipo j
    public static Integer getCantidadCafe(Integer j) {
        return cafes.get(j).kgDisponibles();
    }
}
```

```

// Funcion que calcula el beneficio de venta por cada kg de la variedad i
public static Double getBeneficioVentaKg(Integer i) {
    return variedades.get(i).beneficio();
}

// Funcion que calcula el porcentaje del cafe j necesarios para un kg de la variedad i
public static Double getPorcentajeCafeKg(Integer i, Integer j) {
    String nombreCafe = cafes.get(j).nombre();
    return variedades.get(i).comp().stream()
        .filter(x -> x.nombreCafe().equals(nombreCafe))
        .map(x -> x.cantidadKg())
        .findFirst()
        .orElse(0.);
}

}

// TEST
public static void main(String[] args) {
    System.out.println(" TEST DatosCafes *\n");
    iniDatos("ficheros/Ejercicio1DatosEntrada1.txt");
}
}

```

### 1.3. SolucionCafes

```

package _soluciones;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.jgrapht.GraphPath;

import _datos.DatosCafes;
import ejercicio1.CafeEdge;
import ejercicio1.CafeVertex;

public class SolucionCafes {

    // Propiedades
    private Double beneficio;
    private Map<String, Integer> solucion;
    private List<Integer> path;

    // Metodos de factoria
    public static SolucionCafes of(List<Integer> ls) {
        return new SolucionCafes(ls);
    }

    public static SolucionCafes of(GraphPath<CafeVertex, CafeEdge> path) {
        List<Integer> ls = path.getEdgeList().stream().map(x -> x.action()).toList();
        SolucionCafes res = of(ls);
        res.path = ls;
        return res;
    }

    // Constructor
    private SolucionCafes(List<Integer> ls) {
        Double beneficioTotal = 0.;
        Map<String, Integer> mp = new HashMap<>();
        for(int i=0; i<ls.size(); i++) {
            if(ls.get(i) > 0) {
                beneficioTotal += ls.get(i) * DatosCafes.getBeneficioVentaKg(i);
                mp.put(DatosCafes.getVariedades().get(i).nombre(), ls.get(i));
            }
        }
        beneficio = beneficioTotal;
        solucion = mp;
    }

    // Metodo toString que devuelve una cadena con los kg de cada variedad y el beneficio final obtenido
    @Override
    public String toString() {
        String res = String.format("\nVariedades = %s; Beneficio = %f;", this.solucion, this.beneficio);
        return path==null? res: String.format("%s\nPath de la solucion = %s", res, this.path);
    }
}

```

## **1.4. Ejercicio 1 – Automático**

### **1.4.1. CafeVertex**

```
package ejercicio1;

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

import _datos.DatosCafes;
import us.lsi.common.List2;
import us.lsi.graphs.virtual.VirtualVertex;

public record CafeVertex(Integer index, List<Double> cantidadesCafes)
    implements VirtualVertex<CafeVertex, CafeEdge, Integer> {

    // Metodo de factoria
    public static CafeVertex of(Integer index, List<Double> cantidadesCafes) {
        return new CafeVertex(index, cantidadesCafes);
    }

    // Vertice inicial: indice sera 0 y las cantidades de cafe sera una lista donde
    // cada posicion correspondera a la cantidad de cafe disponible incialmente
    public static CafeVertex initial() {
        List<Double> cantidadesInicialCafe = new ArrayList<>();
        for(int j=0; j<DatosCafes.getNumTiposCafe(); j++) {
            Double cantidadCafe = DatosCafes.getCantidadCafe(j) + 0.;
            cantidadesInicialCafe.add(j, cantidadCafe);
        }
        return of(0, cantidadesInicialCafe);
    }

    // goal: predicado el cual es verdadero si ya hemos recorrido todas las variedades
    public static Predicate<CafeVertex> goal() {
        return x -> x.index() == DatosCafes.getNumVariedades();
    }

    // goalHasSolution: predicado el cual es verdadero si ya hemos recorrido todas
    // las variedades
    public static Predicate<CafeVertex> goalHasSolution() {
        return x -> x.index() == DatosCafes.getNumVariedades();
    }

    // Alternativas a las que podemos ir desde ese vertice, es decir, el rango de cantidad de
    // kg de la variedad podemos coger
    @Override
    public List<Integer> actions() {
        List<Integer> alternativas = new ArrayList<>();

        // en caso de que se hayan recorrido todas las variedades no habra alternativas
        if(index() < DatosCafes.getNumVariedades()) {

            // obtengo una lista con las maxima cantidad que se puede coger de cada cafe
            // en tal variedad siempre y cuando se necesite tal cafe
            List<Double> cantidadesMaximasUsables = new ArrayList<>();
            for(int j=0; j<DatosCafes.getNumTiposCafe(); j++) {
                Double cantidadDisponibleCafe = cantidadesCafes().get(j);
                Double porcentajeCafeEnVariedad = DatosCafes.getPorcentajeCafeKg(index(), j);
                if(porcentajeCafeEnVariedad > 0.) { // si se necesita el cafe en la variedad
                    Double cantidadMaxPuedeUsar = cantidadDisponibleCafe/porcentajeCafeEnVariedad;
                    cantidadesMaximasUsables.add(cantidadMaxPuedeUsar);
                }
            }

            // si ocurre que estamos en la ultima variedad cogemos el maximo que podemos coger,
            // es decir, el minimo de la lista
            Integer alternativa = cantidadesMaximasUsables.stream()
                .min(Comparator.naturalOrder()).orElse(0.).intValue();
            if(index().equals(DatosCafes.getNumVariedades()-1)) {
                alternativas = List2.of(alternativa);
            } else { // sino alternativas = [minimo de la lista .. 0]
                alternativas = List2.rangeList(0, alternativa+1); // +1 porque es rango abierto
            }
        }

        return alternativas;
    }
}
```

```

// Vertice vecino al que vamos, es decir, aumentar el indice y restar a la cantidad de cafe
// disponible el numero de variedades por cantidad de kg necesarios si ese tipo de cafe es
// necesario en tal variedad
@Override
public CafeVertex neighbor(Integer a) {
    List<Double> listaCantidades = List2.copyOf(cantidadesCafes());

    for(int j=0; j<DatosCafes.getNumTiposCafe(); j++) {
        Double cantidadInicial = listaCantidades.get(j);
        Double porcentajeCafeEnVariedad = DatosCafes.getPorcentajeCafeKg(index(), j);
        Double cantidadAEliminar = a*porcentajeCafeEnVariedad;
        if (cantidadAEliminar > 0.) {
            Double cantidadFinal = cantidadInicial - cantidadAEliminar;
            listaCantidades.set(j, cantidadFinal);
        }
    }

    return of(index()+1, listaCantidades);
}

// Arista que construimos, en este caso, la accion sera el numero de kg de la variedad con la
// que estamos trabajando y el peso sera el beneficio por kg de tal variedad por el numero de
// kgs seleccionados de tal variedad
@Override
public CafeEdge edge(Integer a) {
    return CafeEdge.of(this, neighbor(a), a);
}

// Metodo toString que define la representacion en forma de cadena de un vertice
@Override
public String toString() {
    return String.format("%d; %s", this.index, this.cantidadesCafes());
}
}

```

#### 1.4.2. CafeEdge

```

package ejercicio1;

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

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

    // Metodo de factoria
    public static CafeEdge of(CafeVertex source, CafeVertex target, Integer action) {
        Double weight = DatosCafes.getBeneficioVentaKg(source.index()) * action;
        return new CafeEdge(source, target, action, weight);
    }

    // Metodo toString que define la representacion en forma de cadena de una arista
    @Override
    public String toString() {
        return String.format("%d; %.2f", this.action, this.weight);
    }
}

```

### 1.4.3. CafeHeuristic

```
package ejercicio1;

import java.util.ArrayList;

public class CafeHeuristic {

    // Heuristica: lo mejor que puede pasar en un vertice es que se cojan las cantidades maximas
    // posibles de cada variedad que queda por recorrer
    public static Double heuristic(CafeVertex v1, Predicate<CafeVertex> goal, CafeVertex v2) {
        Double res = 0.;

        for(int i=v1.index(); i<DatosCafes.getNumVariedades(); i++) {
            List<Double> cantidadesMaximasUsables = new ArrayList<>();
            // obtengo una lista con las maxima cantidad que se puede coger de cada cafe
            // en tal variedad siempre y cuando se necesite tal cafe
            for(int j=0; j<DatosCafes.getNumTiposCafe(); j++) {
                Double cantidadCafeDisponible = v1.cantidadesCafes().get(j);
                Double porcentajeCafeEnVariedad = DatosCafes.getPorcentajeCafeKg(i, j);
                if(porcentajeCafeEnVariedad > 0.) { // si se necesita el cafe en la variedad
                    Double cantidadMaxPuedeUsar = cantidadCafeDisponible/porcentajeCafeEnVariedad;
                    cantidadesMaximasUsables.add(cantidadMaxPuedeUsar);
                }
            }

            // cogemos el minimo de la lista que sera el maximo de kg que podemos coger de la variedad
            // que estamos recorriendo
            Integer maxPossible = cantidadesMaximasUsables.stream()
                .min(Comparator.naturalOrder()).get().intValue();

            // multiplicamos el maximo de kg que podemos coger por el beneficio por kg de tal variedad
            Double beneficio = DatosCafes.getBeneficioVentaKg(i);
            res += maxPossible*beneficio;
        }

        return res;
    }
}
```

### 1.4.4. TestsEjercicio1

```
package ejercicios.tests;

import java.util.List;

import _datos.DatosCafes;
import _soluciones.SolucionCafes;
import _utils.GraphsPI5;
import _utils.TestsPI5;
import ejercicio1.CafeVertex;

public class TestsEjercicio1 {

    public static void main(String[] args) {
        System.out.println("* TESTS EJERCICIO 1 *");
        TestsPI5.Line("_");
        TestsPI5.Line("*");

        List.of(1,2,3).forEach(num_test -> {
            TestsPI5.iniTest("Ejercicio1DatosEntrada", num_test, DatosCafes::iniDatos);
            TestsPI5.tests(
                CafeVertex.initial(), // Vertice Inicial
                CafeVertex.goal(), // Predicado para un vertice final
                GraphsPI5::cafeBuilder, // Referencia al Builder del grafo
                SolucionCafes::of); // Referencia al metodo factoria para la solucion
        });
    }
}
```

## 1.4.5. Volcado de pantalla

```
* TESTS EJERCICIO 1 *
=====
- Tipos de cafe leídos:
Cafe[nombre=C01, kgDisponibles=5]
Cafe[nombre=C02, kgDisponibles=4]
Cafe[nombre=C03, kgDisponibles=1]
Cafe[nombre=C04, kgDisponibles=2]
Cafe[nombre=C05, kgDisponibles=8]
Cafe[nombre=C06, kgDisponibles=1]
- Tipo de variedades leídas:
Variedad[nombre=P01, beneficio=20.0, comp=[TuplaEj1[nombreCafe=C01, cantidadKg=0.5], TuplaEj1[nombreCafe=C02, cantidadKg=0.4], TuplaEj1[nombreCafe=C03, cantidadKg=0.1]]]
Variedad[nombre=P02, beneficio=10.0, comp=[TuplaEj1[nombreCafe=C04, cantidadKg=0.2], TuplaEj1[nombreCafe=C05, cantidadKg=0.8]]]
Variedad[nombre=P03, beneficio=5.0, comp=[TuplaEj1[nombreCafe=C06, cantidadKg=1.0]]]
=====
Solucion A*:
Variedades = {P01=10, P03=1, P02=10}; Beneficio = 305,000000;
Path de la solucion = [10, 10, 1]
=====
Solucion PDR:
Variedades = {P01=10, P03=1, P02=10}; Beneficio = 305,000000;
Path de la solucion = [10, 10, 1]
=====
Solucion BT:
Variedades = {P01=10, P03=1, P02=10}; Beneficio = 305,000000;
Path de la solucion = [10, 10, 1]
=====
- Tipos de cafe leídos:
Cafe[nombre=C01, kgDisponibles=11]
Cafe[nombre=C02, kgDisponibles=9]
Cafe[nombre=C03, kgDisponibles=7]
Cafe[nombre=C04, kgDisponibles=12]
Cafe[nombre=C05, kgDisponibles=6]
- Tipo de variedades leídas:
Variedad[nombre=P01, beneficio=20.0, comp=[TuplaEj1[nombreCafe=C01, cantidadKg=0.2], TuplaEj1[nombreCafe=C02, cantidadKg=0.4], TuplaEj1[nombreCafe=C05, cantidadKg=0.4]]]
Variedad[nombre=P02, beneficio=10.0, comp=[TuplaEj1[nombreCafe=C02, cantidadKg=0.3], TuplaEj1[nombreCafe=C03, cantidadKg=0.7]]]
Variedad[nombre=P03, beneficio=80.0, comp=[TuplaEj1[nombreCafe=C01, cantidadKg=0.4], TuplaEj1[nombreCafe=C04, cantidadKg=0.6]]]
=====
Solucion A*:
Variedades = {P03=20, P01=15, P02=10}; Beneficio = 2000,000000;
Path de la solucion = [15, 10, 20]
=====
Solucion PDR:
Variedades = {P03=20, P01=15, P02=10}; Beneficio = 2000,000000;
Path de la solucion = [15, 10, 20]
=====
Solucion BT:
Variedades = {P03=20, P01=15, P02=10}; Beneficio = 2000,000000;
Path de la solucion = [15, 10, 20]
=====
- Tipos de cafe leídos:
Cafe[nombre=C01, kgDisponibles=35]
Cafe[nombre=C02, kgDisponibles=4]
Cafe[nombre=C03, kgDisponibles=12]
Cafe[nombre=C04, kgDisponibles=5]
Cafe[nombre=C05, kgDisponibles=30]
Cafe[nombre=C06, kgDisponibles=42]
Cafe[nombre=C07, kgDisponibles=3]
Cafe[nombre=C08, kgDisponibles=2]
Cafe[nombre=C09, kgDisponibles=20]
Cafe[nombre=C10, kgDisponibles=3]
- Tipo de variedades leídas:
Variedad[nombre=P01, beneficio=60.0, comp=[TuplaEj1[nombreCafe=C01, cantidadKg=0.5], TuplaEj1[nombreCafe=C03, cantidadKg=0.4], TuplaEj1[nombreCafe=C07, cantidadKg=0.1]]]
Variedad[nombre=P02, beneficio=25.0, comp=[TuplaEj1[nombreCafe=C02, cantidadKg=1.0]]]
Variedad[nombre=P03, beneficio=5.0, comp=[TuplaEj1[nombreCafe=C02, cantidadKg=0.4], TuplaEj1[nombreCafe=C07, cantidadKg=0.8]]]
Variedad[nombre=P04, beneficio=25.0, comp=[TuplaEj1[nombreCafe=C06, cantidadKg=0.8], TuplaEj1[nombreCafe=C10, cantidadKg=0.2]]]
Variedad[nombre=P05, beneficio=15.0, comp=[TuplaEj1[nombreCafe=C03, cantidadKg=0.4], TuplaEj1[nombreCafe=C08, cantidadKg=0.6]]]
Variedad[nombre=P06, beneficio=100.0, comp=[TuplaEj1[nombreCafe=C01, cantidadKg=0.2], TuplaEj1[nombreCafe=C05, cantidadKg=0.3], TuplaEj1[nombreCafe=C06, cantidadKg=0.3], TuplaEj1[nombreCafe=C09, cantidadKg=0.2]]]
=====
Solucion A*:
Variedades = {P01=30, P02=4, P04=15, P06=100}; Beneficio = 12275,000000;
Path de la solucion = [30, 4, 0, 15, 0, 100]
=====
Solucion PDR:
Variedades = {P01=30, P02=4, P04=15, P06=100}; Beneficio = 12275,000000;
Path de la solucion = [30, 4, 0, 15, 0, 100]
```

## 1.5. Ejercicio 1 – Manual (PDR)

### 1.5.1. CafeProblem

```
package ejercicio1.manual;

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

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

public record CafeProblem(Integer index, List<Double> cantidadesCafes) {

    // Metodo de factoria
    public static CafeProblem of(Integer index, List<Double> cantidadesCafes) {
        return new CafeProblem(index, cantidadesCafes);
    }

    // Vertice inicial: indice sera 0 y las cantidades de cafe sera una lista donde
    // cada posicion corresponda a la cantidad de cafe disponible incialmente
    public static CafeProblem initial() {
        List<Double> cantidadesInicialCafe = new ArrayList<>();
        for (int j = 0; j < DatosCafes.getNumTiposCafe(); j++) {
            Double cantidadCafe = DatosCafes.getCantidadCafe(j) + 0.;
            cantidadesInicialCafe.add(j, cantidadCafe);
        }
        return of(0, cantidadesInicialCafe);
    }

    // Alternativas a las que podemos ir desde ese vertice, es decir, el rango de cantidad de
    // kg de la variedad podemos coger
    public List<Integer> actions() {
        List<Integer> alternativas = new ArrayList<>();

        // en caso de que se hayan recorrido todas las variedades no habra alternativas
        if (index() < DatosCafes.getNumVariedades()) {
            // obtengo una lista con las maxima cantidad que se puede coger de cada cafe
            // en tal variedad siempre y cuando se necesite tal cafe
            List<Double> cantidadesMaximasUsables = new ArrayList<>();
            for (int j = 0; j < DatosCafes.getNumTiposCafe(); j++) {
                Double cantidadDisponibleCafe = cantidadesCafes().get(j);
                Double porcentajeCafeEnVariedad = DatosCafes.getPorcentajeCafeKg(index(), j);
                if (porcentajeCafeEnVariedad > 0.) { // si se necesita el cafe en la variedad
                    Double cantidadMaxPuedeUsar = cantidadDisponibleCafe / porcentajeCafeEnVariedad;
                    cantidadesMaximasUsables.add(cantidadMaxPuedeUsar);
                }
            }

            // si ocurre que estamos en la ultima variedad cogemos el maximo que podemos coger,
            // es decir, el minimo de la lista
            Integer alternativa = cantidadesMaximasUsables.stream().min(Comparator.naturalOrder()).orElse(0.)
                .intValue();
            if (index().equals(DatosCafes.getNumVariedades() - 1)) {
                alternativas = List2.of(alternativa);
            } else { // sino alternativas = [minimo de la lista .. 0]
                alternativas = List2.rangeList(0, alternativa + 1); // +1 porque es rango abierto
            }
        }
        return alternativas;
    }

    // Vertice vecino al que vamos, es decir, aumentar el indice y restar a la cantidad de cafe
    // disponible el numero de variedades por cantidad de kg necesarios si ese tipo de cafe es
    // necesario en tal variedad
    public CafeProblem neighbor(Integer a) {
        List<Double> listaCantidades = List2.copy(cantidadesCafes());

        for (int j = 0; j < DatosCafes.getNumTiposCafe(); j++) {
            Double cantidadInicial = listaCantidades.get(j);
            Double porcentajeCafeEnVariedad = DatosCafes.getPorcentajeCafeKg(index(), j);
            Double cantidadAEliminar = a * porcentajeCafeEnVariedad;
            if (cantidadAEliminar > 0.) {
                Double cantidadFinal = cantidadInicial - cantidadAEliminar;
                listaCantidades.set(j, cantidadFinal);
            }
        }

        return of(index() + 1, listaCantidades);
    }

    // Heuristica: lo mejor que puede pasar en un vertice es que se cojan las cantidades maximas
    // posibles de cada variedad que queda por recorrer
    public Double heuristic() {
        Double res = 0.;
        for (int i = index(); i < DatosCafes.getNumVariedades(); i++) {
            List<Double> cantidadesMaximasUsables = new ArrayList<>();
            // obtengo una lista con las maxima cantidad que se puede coger de cada cafe
            // en tal variedad siempre y cuando se necesite tal cafe
            for (int j = 0; j < DatosCafes.getNumTiposCafe(); j++) {
                Double cantidadCafeDisponible = cantidadesCafes().get(j);
                Double porcentajeCafeEnVariedad = DatosCafes.getPorcentajeCafeKg(i, j);
                if (porcentajeCafeEnVariedad > 0.) { // si se necesita el cafe en la variedad
                    Double cantidadMaxPuedeUsar = cantidadCafeDisponible / porcentajeCafeEnVariedad;
                    cantidadesMaximasUsables.add(cantidadMaxPuedeUsar);
                }
            }

            // cogemos el minimo de la lista que sera el maximo de kg que podemos coger de la variedad
            // que estamos recorriendo
            Integer maxPossible = cantidadesMaximasUsables.stream()
                .min(Comparator.naturalOrder()).get().intValue();

            // multiplicamos el maximo de kg que podemos coger por el beneficio por kg de tal variedad
            Double beneficio = DatosCafes.getBeneficioVentaKg(i);
            res += maxPossible * beneficio;
        }
        return res;
    }
}
```

## 1.5.2. CafePDR

```

package ejercicio1.manual;

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

import _datos.DatosCafes;
import _soluciones.SolucionCafes;

public class CafePDR {

    // Creamos el tipo que sera la solucion parcial, que es el par formado por la alternativa
    // elegida y el valor de la propiedad a optimizar
    public static record Spm(Integer action, Double weight) implements Comparable<Spm> {
        public static Spm of(Integer action, Double weight) {
            return new Spm(action, weight);
        }

        @Override
        public int compareTo(Spm o) {
            return this.weight.compareTo(o.weight);
        }
    }

    // Propiedades
    public static Map<CafeProblem, Spm> memory;
    public static Double mejorValor;

    // Funcion que se encarga de obtener la solucion optima
    public static SolucionCafes search() {
        memory = new HashMap<>();
        mejorValor = 0.; // estamos maximizando
        pdr_search(CafeProblem.initial(), 0., memory); // llamada al algoritmo
        return getSolucion();
    }

    // Funcion que es basicamente el algoritmo
    private static Spm pdr_search(CafeProblem prob, Double acumulado, Map<CafeProblem, Spm> memoria) {
        Spm res;
        Boolean esTerminal = prob.index() == DatosCafes.getNumVariedades();

        if(memoria.containsKey(prob)) { // comprobamos si se encuentra en memoria
            res = memoria.get(prob);
        } else if(esTerminal) { // si es terminal...
            res = Spm.of(null, 0.);
            memoria.put(prob, res);
            if(acumulado > mejorValor) { // porque estamos maximizando
                mejorValor = acumulado;
            }
        } else { // si no es terminal ni se encuentra en memoria...
            List<Spm> soluciones = new ArrayList<>();
            for(Integer accion: prob.actions()) {
                Double cota = acatar(acumulado, prob, accion);
                if(cota < mejorValor) {
                    continue;
                }
                CafeProblem vecino = prob.neighbor(accion);
                Spm s = pdr_search(vecino, acumulado + accion*DatosCafes.getBeneficioVentaKg(prob.index()), memoria);
                if(s != null) {
                    Spm amp = Spm.of(accion, s.weight() + accion*DatosCafes.getBeneficioVentaKg(prob.index()));
                    soluciones.add(amp);
                }
            }
            // estamos maximizando
            res = soluciones.stream()
                .max(Comparator.naturalOrder()).orElse(null);
            if(res != null) {
                memoria.put(prob, res);
            }
        }
        return res;
    }

    // Funcion acatar: sirve para saber que es lo mejor que puede pasar desde un vertice
    // tomando la alternativa a
    private static Double acatar(Double acum, CafeProblem p, Integer a) {
        return acum + a*DatosCafes.getBeneficioVentaKg(p.index()) + p.neighbor(a).heuristic();
    }

    // Funcion que te da la solucion tras aplicar el algoritmo
    public static SolucionCafes getSolucion() {
        List<Integer> acciones = new ArrayList<>();
        CafeProblem prob = CafeProblem.initial();
        Spm spm = memoria.get(prob);
        while (spm != null & spm.action != null) {
            CafeProblem old = prob;
            acciones.add(spm.action);
            prob = old.neighbor(spm.action);
            spm = memoria.get(prob);
        }
        return SolucionCafes.of(acciones);
    }
}

```

### **1.5.3. TestsEjercicio1PDR**

```
package ejercicios.tests.manual;

import java.util.List;

import _datos.DatosCafes;
import _utils.TestsPI5;
import ejercicio1.manual.CafePDR;
import us.lsi.common.String2;

public class TestsEjercicio1PDR {

    public static void main(String[] args) {
        System.out.println("* TESTS EJERCICIO 1 - PDR MANUAL *\n");
        TestsPI5.Line("**");

        List.of(1,2,3).forEach(num_test -> {
            DatosCafes.iniDatos("ficheros/Ejercicio1DatosEntrada"+num_test+".txt");
            String2.toConsole("\n- Solucion obtenida: %s\n", CafePDR.search());
            TestsPI5.line("**");
        });
    }
}
```

### **1.5.4. Volcado de pantalla**

```
* TESTS EJERCICIO 1 - PDR MANUAL *

*****
- Tipos de cafe leídos:
Cafe[nombre=C01, kgDisponibles=5]
Cafe[nombre=C02, kgDisponibles=4]
Cafe[nombre=C03, kgDisponibles=1]
Cafe[nombre=C04, kgDisponibles=2]
Cafe[nombre=C05, kgDisponibles=8]
Cafe[nombre=C06, kgDisponibles=1]

- Tipo de variedades leídas:
Variedad[nombre=P01, beneficio=20.0, comp=[TuplaEj1[nombreCafe=C01, cantidadKg=0.5], TuplaEj1[nombreCafe=C02, cantidadKg=0.4], TuplaEj1[nombreCafe=C03, cantidadKg=0.1]]]
Variedad[nombre=P02, beneficio=10.0, comp=[TuplaEj1[nombreCafe=C04, cantidadKg=0.2], TuplaEj1[nombreCafe=C05, cantidadKg=0.8]]]
Variedad[nombre=P03, beneficio=5.0, comp=[TuplaEj1[nombreCafe=C06, cantidadKg=1.0]]]

- Solucion obtenida:
Variedades = {P01=10, P03=1, P02=10}; Beneficio = 305,000000;

*****
- Tipos de cafe leídos:
Cafe[nombre=C01, kgDisponibles=11]
Cafe[nombre=C02, kgDisponibles=9]
Cafe[nombre=C03, kgDisponibles=7]
Cafe[nombre=C04, kgDisponibles=12]
Cafe[nombre=C05, kgDisponibles=6]

- Tipo de variedades leídas:
Variedad[nombre=P01, beneficio=20.0, comp=[TuplaEj1[nombreCafe=C01, cantidadKg=0.2], TuplaEj1[nombreCafe=C02, cantidadKg=0.4], TuplaEj1[nombreCafe=C05, cantidadKg=0.4]]]
Variedad[nombre=P02, beneficio=10.0, comp=[TuplaEj1[nombreCafe=C02, cantidadKg=0.3], TuplaEj1[nombreCafe=C03, cantidadKg=0.7]]]
Variedad[nombre=P03, beneficio=80.0, comp=[TuplaEj1[nombreCafe=C01, cantidadKg=0.4], TuplaEj1[nombreCafe=C04, cantidadKg=0.6]]]

- Solucion obtenida:
Variedades = {P01=15, P03=20, P02=10}; Beneficio = 2000,000000;

*****
- Tipos de cafe leídos:
Cafe[nombre=C01, kgDisponibles=35]
Cafe[nombre=C02, kgDisponibles=4]
Cafe[nombre=C03, kgDisponibles=12]
Cafe[nombre=C04, kgDisponibles=5]
Cafe[nombre=C05, kgDisponibles=30]
Cafe[nombre=C06, kgDisponibles=42]
Cafe[nombre=C07, kgDisponibles=3]
Cafe[nombre=C08, kgDisponibles=2]
Cafe[nombre=C09, kgDisponibles=20]
Cafe[nombre=C10, kgDisponibles=3]

- Tipo de variedades leídas:
Variedad[nombre=P01, beneficio=60.0, comp=[TuplaEj1[nombreCafe=C01, cantidadKg=0.5], TuplaEj1[nombreCafe=C03, cantidadKg=0.4], TuplaEj1[nombreCafe=C07, cantidadKg=0.1]]]
Variedad[nombre=P02, beneficio=25.0, comp=[TuplaEj1[nombreCafe=C02, cantidadKg=1.0]]]
Variedad[nombre=P03, beneficio=5.0, comp=[TuplaEj1[nombreCafe=C02, cantidadKg=0.4], TuplaEj1[nombreCafe=C07, cantidadKg=0.8]]]
Variedad[nombre=P04, beneficio=25.0, comp=[TuplaEj1[nombreCafe=C06, cantidadKg=0.8], TuplaEj1[nombreCafe=C10, cantidadKg=0.2]]]
Variedad[nombre=P05, beneficio=15.0, comp=[TuplaEj1[nombreCafe=C03, cantidadKg=0.4], TuplaEj1[nombreCafe=C08, cantidadKg=0.6]]]
Variedad[nombre=P06, beneficio=100.0, comp=[TuplaEj1[nombreCafe=C01, cantidadKg=0.2], TuplaEj1[nombreCafe=C05, cantidadKg=0.3], TuplaEj1[nombreCafe=C06, cantidadKg=0.3], TuplaEj1[nombreCafe=C09, cantidadKg=0.2]]]

- Solucion obtenida:
Variedades = {P01=30, P02=4, P04=15, P06=100}; Beneficio = 12275,000000;
```

## 2. EJERCICIO 2

### 2.1. Modelo

- Modelo alternativo:
  - $x_i$ : variable binaria, indica si el curso  $i$  se selecciona,  $i \in [0, n]$

$$\begin{aligned} & \min \sum_{i=0}^{n-1} p_i x_i \\ & |U_{i=0|x_i=1}^{n-1} t_i| = m \\ & CD_{i=0|x_i=1}^{n-1} c_i \leq mc \\ & \text{bin } x_i, \quad i \in [0, n) \end{aligned}$$

### 2.2. DatosCursos

```
package _datos;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import us.lsi.common.Files2;

public class DatosCursos {

    public static Integer maxCentros;
    public static List<Curso> cursos;

    // Funcion que realiza la lectura de fichero y crea un tipo DatosCursos
    public static void iniDatos(String fichero) {
        List<Curso> listaCursos = new ArrayList<>();

        List<String> filas = Files2.linesFromFile(fichero);
        List<String> cursosString = filas.subList(1, filas.size());

        // La primera linea del fichero es el numero maximo de centros
        String primeraLinea = filas.get(0);
        String[] partes = primeraLinea.split("=");
        Integer numMaxCentros = Integer.valueOf(partes[1].trim());
        maxCentros = numMaxCentros;

        // Transformamos una cadena en el tipo Curso y lo añadimos a la lista de cursos
        for(String linea: cursosString) {
            Curso c = Curso.parseaCurso(linea);
            listaCursos.add(c);
        }
        cursos = listaCursos;

        toConsole();
    }

    // Para mostar el resultado por pantalla
    private static void toConsole() {
        System.out.println("- Numero maximo de centros leido: " + maxCentros);
        System.out.println("- Cursos leidos:");
        for(Curso c: cursos) {
            System.out.println(" " + c);
        }
    }

    // Funcion que devuelve la lista de centros
    public static List<Curso> getCursos() {
        return cursos;
    }

    // Funcion que devuelve el numero de cursos
    public static Integer getNumCursos() {
        return cursos.size();
    }

    // Funcion que devuelve las tematicas del problema
    public static Set<Integer> getAllTematicas() {
        Set<Integer> tematicas = new HashSet<>();
        for(Curso c: cursos) {
            Set<Integer> t = c.tematicas();
            tematicas.addAll(t);
        }
        return tematicas;
    }

    // Funcion que devuelve el numero de tematicas
    public static Integer getNumTematicas() {
        return getAllTematicas().size();
    }
}
```

```

// Funcion que devuelve el numero de centros
public static Integer getNumCentros() {
    Set<Integer> centros = new HashSet<>();
    for (Curso c : cursos) {
        Integer centro = c.centro();
        centros.add(centro);
    }
    return centros.size();
}

// Funcion que devuelve el numero maximo de centros
public static Integer getMaxCentros() {
    return maxCentros;
}

// Funcion que devuelve el precio de la inscripcion del curso i
public static Double getPrecioInscripcion(Integer i) {
    return cursos.get(i).coste();
}

// Funcion que devuelve las tematicas del curso i
public static Set<Integer> getTematicasCurso(Integer i) {
    return cursos.get(i).tematicas();
}

// Funcion que devuelve el centro donde se imparte el curso i
public static Integer getCentroCurso(Integer i) {
    return cursos.get(i).centro();
}

// TEST
public static void main(String[] args) {
    System.out.println("** TEST DatosCursos *\n");
    iniDatos("ficheros/Ejercicio2DatosEntrada1.txt");
}
}

```

### **2.3. SolucionCursos**

```

package _soluciones;

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

import org.jgrapht.GraphPath;

import _datos.DatosCursos;
import ejercicio2.CursosEdge;
import ejercicio2.CursosVertex;

public class SolucionCursos {

    // Propiedades
    private Double coste;
    private List<String> cursos;
    private List<Integer> path;

    // Metodos de factoria
    public static SolucionCursos of(List<Integer> ls) {
        return new SolucionCursos(ls);
    }

    public static SolucionCursos of(GraphPath<CursosVertex, CursosEdge> path) {
        List<Integer> ls = path.getEdgeList().stream().map(x -> x.action()).toList();
        SolucionCursos res = of(ls);
        res.path = ls;
        return res;
    }
}

```

```

// Constructor
public SolucionCursos(List<Integer> ls) {
    Double costeTotal = 0.;
    List<String> cursosElegidos = new ArrayList<>();
    for(int i=0; i<ls.size(); i++) {
        if(ls.get(i) > 0) {
            costeTotal += DatosCursos.getPrecioInscripcion(i);
            String curso = DatosCursos.getCursos().get(i).id();
            cursosElegidos.add(curso);
        }
    }
    coste = costeTotal;
    cursos = cursosElegidos;
}

// Metodo toString que devuelve una cadena con los cursos y el coste total de elegir esos cursos
@Override
public String toString() {
    String res = String.format("\nCursos seleccionados = %s; Coste = %f;",
        this.cursos, this.coste);
    return path==null? res: String.format("%s\nPath de la solucion = %s", res, this.path);
}
}

```

## 2.4. Ejercicio 2 – Automático

### 2.4.1. CursosVertex

```

package ejercicio2;

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

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

public record CursosVertex(Integer index, Set<Integer> tematicasRestantes, Set<Integer> centrosUtilizados)
    implements VirtualVertex<CursosVertex, CursosEdge, Integer> {

    // Metodo de factoria
    public static CursosVertex of(Integer index, Set<Integer> tematicasRestantes,
        Set<Integer> centrosUtilizados) {
        return new CursosVertex(index, tematicasRestantes, centrosUtilizados);
    }

    // Vertice inicial: indice sera 0, las tematicas restantes seran todas las tematicas
    // que hay en el problema y los cursos utilizados sera un conjunto vacio
    public static CursosVertex initial() {
        Set<Integer> tematicasRestantes = DatosCursos.getAllTematicas();
        Set<Integer> centrosUtilizados = new HashSet<>();
        return of(0, tematicasRestantes, centrosUtilizados);
    }

    // goal: predicado el cual es verdadero cuando ya hemos recorrido todos los cursos
    public static Predicate<CursosVertex> goal() {
        return x -> x.index() == DatosCursos.getNumCursos();
    }
}

```

```

// goalHasSolution: predicado el cual es verdadero si no quedan tematicas restantes y
// el numero de centros utilizado es menor o igual al maximo de centros posibles
public static Predicate<CursosVertex> goalHasSolution() {
    return x -> x.tematicasRestantes().isEmpty() &&
        x.centrosUtilizados().size() <= DatosCursos.getMaxCentros();
}

// Alternativas a las que podemos ir desde ese vertice, es decir, si podemos coger
// el curso o no
@Override
public List<Integer> actions() {
    List<Integer> alternativas = new ArrayList<>();

    // en caso de que se hayan recorrido todos los cursos, no habra alternativas
    if(index() < DatosCursos.getNumCursos()) {
        // en caso de que ya sea solucion, es decir, que no haya que cubrir tematicas y
        // que no se supera el numero maximo de centros, la alternativa sera no cogerlo
        // ya que nuestra intencion es minimizar el coste de los precios de inscripcion
        if(tematicasRestantes().isEmpty() &&
            centrosUtilizados().size() <= DatosCursos.getMaxCentros()) {
            alternativas = List2.of(0);

        } else {    // si no es solucion, miramos los siguientes casos
            // obtenemos el conjunto de tematicas que quedarían y los centros seleccionados
            Set<Integer> restTematicas = Set2.difference(tematicasRestantes(),
                DatosCursos.getTematicasCurso(index()));
            Set<Integer> centrosUsados = Set2.copy(centrosUtilizados());
            centrosUsados.add(DatosCursos.getCentroCurso(index()));
            // si estamos en el ultimo curso e inscribiéndonos en tal curso no es solucion,
            // no habra alternativas
            if(index() == DatosCursos.getNumCursos()-1) {
                if(restTematicas.isEmpty() && centrosUsados.size() <= DatosCursos.getMaxCentros()) {
                    alternativas = List2.of(1);
                } else {
                    alternativas = List2.empty();
                }
            } else {
                // si las tematicas restantes se quedan igual o el numero de centros es mayor al
                // maximo posible no se seleccionara el curso (alternativa: no inscribirse)
                if(restTematicas.equals(tematicasRestantes()) ||
                    centrosUsados.size() > DatosCursos.getMaxCentros()) {
                    alternativas = List2.of(0);
                }
                // si no se cumple lo anterior, las alternativas son inscribirse o
                // no inscribirse en el curso
                else {
                    alternativas = List2.of(0, 1);
                }
            }
        }
    }
    return alternativas;
}

// Vertice vecino al que vamos, es decir, aumentar el indice, quitar las tematicas
// que hayamos ya utilizado y añadir al conjunto de centros el centro seleccionado
@Override
public CursosVertex neighbor(Integer a) {
    Set<Integer> tematicasVecino = null;
    Set<Integer> centrosVecino = null;

    // si seleccionamos el curso quitamos las tematicas que hemos usado y añadimos el centro del curso seleccionado
    if(a != 0) {
        tematicasVecino = Set2.difference(tematicasRestantes(),
            DatosCursos.getTematicasCurso(index()));
        centrosVecino = Set2.copy(centrosUtilizados());
        centrosVecino.add(DatosCursos.getCentroCurso(index()));

    } else {    // si no hemos seleccionado el curso las tematicas y los centros se quedan igual
        tematicasVecino = Set2.copy(tematicasRestantes());
        centrosVecino = Set2.copy(centrosUtilizados());
    }

    return of(index()+1, tematicasVecino, centrosVecino);
}

// Arista que construimos, en este caso, la accion sera 1 o 0 dependiendo si hemos
// seleccionado o no ese curso y el peso sera el precio de inscripcion del curso por
// 0 o 1 (ya que si no se selecciona el peso sera 0)
@Override
public CursosEdge edge(Integer a) {
    return CursosEdge.of(this, neighbor(a), a);
}

// Metodo toString que define la representacion en forma de cadena de un vertice
@Override
public String toString() {
    return String.format("%d; %s; %s", this.index, this.tematicasRestantes, this.centrosUtilizados);
}

```

## 2.4.2. CursosEdge

```
package ejercicio2;

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

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

    // Metodo factorial
    public static CursosEdge of(CursosVertex source, CursosVertex target, Integer action) {
        Double weight = DatosCursos.getPrecioInscripcion(source.index()) * action;
        return new CursosEdge(source, target, action, weight);
    }

    // Metodo toString que define la representacion en forma de cadena de una arista
    @Override
    public String toString() {
        return String.format("%d; %.2f", this.action, this.weight);
    }
}
```

## 2.4.3. CursosHeuristic

```
package ejercicio2;

import java.util.ArrayList;
public class CursosHeuristic {

    // Heuristica: lo mejor que puede pasar en un vertice es que cojamos el curso con menor
    // peso (si las tematicas restantes cambian y no se supera el numero maximo de centros utilizados)
    public static Double heuristic(CursosVertex v1, Predicate<CursosVertex> goal, CursosVertex v2) {
        Double res = 0.;
        List<Double> ls = new ArrayList<>();
        if(!v1.tematicasRestantes().isEmpty() && v1.centrosUtilizados().size() <= DatosCursos.getMaxCentros()) {
            for (int i=v1.index(); i<DatosCursos.getNumCursos(); i++) {
                // obtenemos el conjunto de tematicas que quedarían y los centros seleccionados
                Boolean tematicas = !List2.intersection(v1.tematicasRestantes(), DatosCursos.getTematicasCurso(i))
                    .isEmpty();
                Set<Integer> centrosUsados = Set2.copy(v1.centrosUtilizados());
                centrosUsados.add(DatosCursos.getCentroCurso(i));
                // si las tematicas restantes cambian y no se supera el numero maximo de centros utilizados
                // entonces tendremos en cuenta el precio de la inscripcion de ese curso añadiendolo a la lista
                if(tematicas && centrosUsados.size() <= DatosCursos.getMaxCentros()) {
                    ls.add(DatosCursos.getPrecioInscripcion(i));
                }
            }
            // devolvemos el precio de inscripcion menor de los cursos seleccionados
            res = ls.stream().min(Comparator.naturalOrder()).orElse(100.);
        }
        return res;
    }
}
```

## 2.4.4. TestsEjercicio2

```
package ejercicios.tests;

import java.util.List;

import _datos.DatosCursos;
import _soluciones.SolucionCursos;
import _utils.GraphsPI5;
import _utils.TestsPI5;
import ejercicio2.CursosVertex;

public class TestsEjercicio2 {

    public static void main(String[] args) {
        System.out.println("* TESTS EJERCICIO 2 *");
        TestsPI5.line("_");
        TestsPI5.line("/*");

        List.of(1,2,3).forEach(num_test -> {
            TestsPI5.iniTest("Ejercicio2DatosEntrada", num_test, DatosCursos::iniDatos);
            TestsPI5.tests(
                CursosVertex.initial(),           // Vertice Inicial
                CursosVertex.goal(),             // Predicado para un vertice final
                GraphsPI5:cursosBuilder,        // Referencia al Builder del grafo
                SolucionCursos::of);            // Referencia al metodo factorial para la solucion
        ));
    }
}
```

## 2.4.5. Volcado de pantalla

\* TESTS EJERCICIO 2 \*

\*\*\*\*\*  
- Numero maximo de centros leido: 1  
- Cursos leidos:

Curso[id=C0, tematicas=[1, 2, 3, 4], coste=10.0, centro=0]  
Curso[id=C1, tematicas=[1, 4], coste=3.0, centro=0]  
Curso[id=C2, tematicas=[5], coste=1.5, centro=1]  
Curso[id=C3, tematicas=[5], coste=5.0, centro=0]

=====

Solucion A\*:

Cursos seleccionados = [C0, C3]; Coste = 15,000000;  
Path de la solucion = [1, 0, 0, 1]

Solucion PDR:

Cursos seleccionados = [C0, C3]; Coste = 15,000000;  
Path de la solucion = [1, 0, 0, 1]

Solucion BT:

Cursos seleccionados = [C0, C3]; Coste = 15,000000;  
Path de la solucion = [1, 0, 0, 1]

\*\*\*\*\*

- Numero maximo de centros leido: 2

- Cursos leidos:

Curso[id=C4, tematicas=[2, 3], coste=2.0, centro=0]  
Curso[id=C5, tematicas=[4], coste=3.0, centro=0]  
Curso[id=C6, tematicas=[1, 5], coste=5.0, centro=0]  
Curso[id=C7, tematicas=[1, 3, 4], coste=3.5, centro=2]  
Curso[id=C8, tematicas=[4, 5], coste=1.5, centro=1]

=====

Solucion A\*:

Cursos seleccionados = [C4, C6, C8]; Coste = 8,500000;  
Path de la solucion = [1, 0, 1, 0, 1]

Solucion PDR:

Cursos seleccionados = [C4, C6, C8]; Coste = 8,500000;  
Path de la solucion = [1, 0, 1, 0, 1]

Solucion BT:

Cursos seleccionados = [C4, C6, C8]; Coste = 8,500000;  
Path de la solucion = [1, 0, 1, 0, 1]

\*\*\*\*\*

- Numero maximo de centros leido: 3

- Cursos leidos:

Curso[id=C9, tematicas=[2, 6, 7], coste=2.0, centro=2]  
Curso[id=C10, tematicas=[7], coste=3.0, centro=0]  
Curso[id=C11, tematicas=[1, 5], coste=5.0, centro=0]  
Curso[id=C12, tematicas=[1, 3, 4], coste=3.5, centro=2]  
Curso[id=C13, tematicas=[3, 7], coste=1.5, centro=1]  
Curso[id=C14, tematicas=[4, 5, 6], coste=4.5, centro=0]  
Curso[id=C15, tematicas=[5, 6], coste=6.0, centro=1]  
Curso[id=C16, tematicas=[2, 3, 5], coste=1.0, centro=1]

=====

Solucion A\*:

Cursos seleccionados = [C9, C12, C16]; Coste = 6,500000;  
Path de la solucion = [1, 0, 0, 1, 0, 0, 0, 1]

Solucion PDR:

Cursos seleccionados = [C9, C12, C16]; Coste = 6,500000;  
Path de la solucion = [1, 0, 0, 1, 0, 0, 0, 1]

Solucion BT:

Cursos seleccionados = [C9, C12, C16]; Coste = 6,500000;  
Path de la solucion = [1, 0, 0, 1, 0, 0, 0, 1]

\*\*\*\*\*

Si nos damos cuenta salen las id seguidas, pero la solucin da correctamente

## 2.5. Ejercicio 2 – Manual (PDR)

### 2.5.1. CursosProblem

```
package ejercicio2.manual;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import _datos.DatosCursos;
import us.lsi.common.List2;
import us.lsi.common.Set2;

public record CursosProblem(Integer index, Set<Integer> tematicasRestantes, Set<Integer> centrosUtilizados) {

    // Metodo de factorial
    public static CursosProblem of(Integer index, Set<Integer> tematicasRestantes,
                                    Set<Integer> centrosUtilizados) {
        return new CursosProblem(index, tematicasRestantes, centrosUtilizados);
    }

    // Vertice inicial: indice sera 0, las tematicas restantes seran todas las tematicas
    // que hay en el problema y los cursos utilizados sera un conjunto vacio
    public static CursosProblem initial() {
        Set<Integer> tematicasRestantes = DatosCursos.getAllTematicas();
        Set<Integer> centrosUtilizados = new HashSet<>();
        return of(0, tematicasRestantes, centrosUtilizados);
    }

    // Alternativas a las que podemos ir desde ese vertice, es decir, si podemos coger
    // el curso o no
    public List<Integer> actions() {
        List<Integer> alternativas = new ArrayList<>();

        // en caso de que se hayan recorrido todos los cursos, no habra alternativas
        if(index() < DatosCursos.getNumCursos()) {
            // en caso de que ya sea solucion, es decir, que no haya que cubrir tematicas y
            // que no se supera el numero maximo de centros, la alternativa sera no cogerlo
            // ya que nuestra intencion es minimizar el coste de los precios de inscripcion
            if(tematicasRestantes().isEmpty() &&
               centrosUtilizados().size() <= DatosCursos.getMaxCentros()) {
                alternativas = List2.of(0);
            } else { // si no es solucion, miramos los siguientes casos
                // obtenemos el conjunto de tematicas que quedarian y los centros seleccionados
                Set<Integer> restTematicas = Set2.difference(tematicasRestantes(),
                                                               DatosCursos.getTematicasCurso(index()));
                Set<Integer> centrosUsados = Set2.copy(centrosUtilizados());
                centrosUsados.add(DatosCursos.getCentroCurso(index()));

                // si estamos en el ultimo curso e inscribiendonos en tal curso no es solucion,
                // no habra alternativas
                if(index() == DatosCursos.getNumCursos()-1) {
                    if(restTematicas.isEmpty() && centrosUsados.size() <= DatosCursos.getMaxCentros()) {
                        alternativas = List2.of(1);
                    } else {
                        alternativas = List2.empty();
                    }
                } else {
                    // si las tematicas restantes se quedan igual o el numero de centros es mayor al
                    // maximo posible no se seleccionara el curso (alternativa: no inscribirse)
                    if(restTematicas.equals(tematicasRestantes()) ||
                       centrosUsados.size() > DatosCursos.getMaxCentros()) {
                        alternativas = List2.of(0);
                    }
                    // si no se cumple lo anterior, las alternativas son inscribirse o
                    // no inscribirse en el curso
                    else {
                        alternativas = List2.of(0, 1);
                    }
                }
            }
        }
        return alternativas;
    }
}
```

```

// Vertice vecino al que vamos, es decir, aumentar el indice, quitar las tematicas
// que hayamos ya utilizado y añadir al conjunto de centros el centro seleccionado
public CursosProblem neighbor(Integer a) {
    Set<Integer> tematicasVecino = null;
    Set<Integer> centrosVecino = null;

    // si seleccionamos el curso quitamos las tematicas que hemos usado y añadimos el centro del curso seleccionado
    if(a != 0) {
        tematicasVecino = Set2.difference(tematicasRestantes(),
            DatosCursos.getTematicasCurso(index()));
        centrosVecino = Set2.copy(centrosUtilizados());
        centrosVecino.add(DatosCursos.getCentroCurso(index()));

    } else { // si no hemos seleccionado el cursos las tematicas y los centros se quedan igual
        tematicasVecino = Set2.copy(tematicasRestantes());
        centrosVecino = Set2.copy(centrosUtilizados());
    }

    return of(index()+1, tematicasVecino, centrosVecino);
}

// Heuristica: lo mejor que puede pasar en un vertice es que cojamos el curso con menor
// peso (si las tematicas restantes cambian y no se supera el numero maximo de centros utilizados)
public Double heuristic() {
    Double res = 0.;
    List<Double> ls = new ArrayList<>();

    if(!tematicasRestantes().isEmpty() && centrosUtilizados().size() <= DatosCursos.getMaxCentros()) {
        for (int i=index(); i<DatosCursos.getNumCursos(); i++) {
            // obtenemos el conjunto de tematicas que quedarían y los centros seleccionados
            Boolean tematicas = !List2.intersection(tematicasRestantes(), DatosCursos.getTematicasCurso(i))
                .isEmpty();
            Set<Integer> centrosUsados = Set2.copy(centrosUtilizados());
            centrosUsados.add(DatosCursos.getCentroCurso(i));
            // si las tematicas restantes cambian y no se supera el numero maximo de centros utilizados
            // entonces tendremos en cuenta el precio de la inscripción de ese curso añadiéndolo a la lista
            if(tematicas && centrosUsados.size() <= DatosCursos.getMaxCentros()) {
                ls.add(DatosCursos.getPrecioInscripcion(i));
            }
        }
        // devolvemos el precio de inscripción menor de los cursos seleccionados
        res = ls.stream().min(Comparator.naturalOrder()).orElse(100.);
    }
}

return res;
}

```

}

## 2.5.2. CursosPDR

```

package ejercicio2.manual;

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

import _datos.DatosCursos;
import _soluciones.SolucionCursos;

public class CursosPDR {

    // Creamos el tipo que sera la solucion parcial, que es el par formado por la alternativa
    // elegida y el valor de la propiedad a optimizar
    public static record Spm(Integer action, Double weight) implements Comparable<Spm> {
        public static Spm of(Integer action, Double weight) {
            return new Spm(action, weight);
        }

        @Override
        public int compareTo(Spm o) {
            return this.weight.compareTo(o.weight);
        }
    }

    // Propiedades
    public static Map<CursosProblem,Spm> memory;
    public static Double mejorValor;

    // Funcion que se encarga de obtener la solucion optima
    public static SolucionCursos search() {
        memory = new HashMap<>();
        mejorValor = Double.MAX_VALUE; // estamos minimizando
        pdr_search(CursosProblem.initial(), 0., memory); // llamada al algoritmo
        return getSolucion();
    }
}

```

```

// Funcion que es basicamente el algoritmo
private static Spm pdr_search(CursosProblem prob, Double acumulado, Map<CursosProblem, Spm> memoria) {
    Spm res;
    Boolean esTerminal = prob.index() == DatosCursos.getNumCursos();
    Boolean esSolucion = prob.tematicasRestantes().isEmpty() &&
        prob.centrosUtilizados().size() <= DatosCursos.getMaxCentros();

    if(memory.containsKey(prob)) { // comprobamos si se encuentra en memoria
        res = memory.get(prob);
    } else if(esTerminal && esSolucion) { // si es terminal...
        res = Spm.of(null, 0.);
        memoria.put(prob, res);
        if(acumulado < mejorValor) { // porque estamos minimizando
            mejorValor = acumulado;
        }
    } else { // si no es terminal y solucion ni se encuentra en memoria...
        List<Spm> soluciones = new ArrayList<>();
        for(Integer accion: prob.actions()) {
            Double cota = acotar(acumulado, prob, accion);
            if(cota > mejorValor) {
                continue;
            }
            CursosProblem vecino = prob.neighbor(accion);
            Spm s = pdr_search(vecino, acumulado + accion*DatosCursos.getPrecioInscripcion(prob.index()), memoria);
            if(s != null) {
                Spm amp = Spm.of(accion, s.weight() + accion*DatosCursos.getPrecioInscripcion(prob.index()));
                soluciones.add(amp);
            }
        }
        // estamos minimizando
        res = soluciones.stream()
            .min(Comparator.naturalOrder()).orElse(null);
        if(res != null) {
            memoria.put(prob, res);
        }
    }
    return res;
}

// Funcion acotar: sirve para saber que es lo mejor que puede pasar desde un vertice
// tomando la alternativa a
private static Double acotar(Double acum, CursosProblem p, Integer a) {
    return acum + a*DatosCursos.getPrecioInscripcion(p.index()) + p.neighbor(a).heuristic();
}

// Funcion que te da la solucion tras aplicar el algoritmo
public static SolucionCursos getSolucion() {
    List<Integer> acciones = new ArrayList<>();
    CursosProblem prob = CursosProblem.initial();
    Spm spm = memoria.get(prob);
    while(spm != null && spm.action != null) {
        CursosProblem old = prob;
        acciones.add(spm.action);
        prob = old.neighbor(spm.action);
        spm = memoria.get(prob);
    }
    return SolucionCursos.of(acciones);
}
}

```

### 2.5.3. TestsEjercicio2PDR

```

package ejercicios.tests.manual;

import java.util.List;

import _datos.DatosCursos;
import _utils.TestsPI5;
import ejercicio2.manual.CursosPDR;
import us.lsi.common.String2;

public class TestsEjercicio2PDR {

    public static void main(String[] args) {
        System.out.println("* TESTS EJERCICIO 2 - PDR MANUAL *\n");
        TestsPI5.line("*");

        List.of(1,2,3).forEach(num_test -> {
            DatosCursos.iniDatos("ficheros/Ejercicio2DatosEntrada"+num_test+".txt");
            String2.toConsole("\n- Solucion obtenida: %s\n", CursosPDR.search());
            TestsPI5.line("*");
        });
    }
}

```

## 2.5.4. Volcado de pantalla

\* TESTS EJERCICIO 2 - PDR MANUAL \*

```
*****
- Numero maximo de centros leido: 1
- Cursos leidos:
  Curso[id=C0, tematicas=[1, 2, 3, 4], coste=10.0, centro=0]
  Curso[id=C1, tematicas=[1, 4], coste=3.0, centro=0]
  Curso[id=C2, tematicas=[5], coste=1.5, centro=1]
  Curso[id=C3, tematicas=[5], coste=5.0, centro=0]

- Solucion obtenida:
Cursos seleccionados = [C0, C3]; Coste = 15,000000;

*****
- Numero maximo de centros leido: 2
- Cursos leidos:
  Curso[id=C4, tematicas=[2, 3], coste=2.0, centro=0]
  Curso[id=C5, tematicas=[4], coste=3.0, centro=0]
  Curso[id=C6, tematicas=[1, 5], coste=5.0, centro=0]
  Curso[id=C7, tematicas=[1, 3, 4], coste=3.5, centro=2]
  Curso[id=C8, tematicas=[4, 5], coste=1.5, centro=1]

- Solucion obtenida:
Cursos seleccionados = [C4, C6, C8]; Coste = 8,500000;

*****
- Numero maximo de centros leido: 3
- Cursos leidos:
  Curso[id=C9, tematicas=[2, 6, 7], coste=2.0, centro=2]
  Curso[id=C10, tematicas=[7], coste=3.0, centro=0]
  Curso[id=C11, tematicas=[1, 5], coste=5.0, centro=0]
  Curso[id=C12, tematicas=[1, 3, 4], coste=3.5, centro=2]
  Curso[id=C13, tematicas=[3, 7], coste=1.5, centro=1]
  Curso[id=C14, tematicas=[4, 5, 6], coste=4.5, centro=0]
  Curso[id=C15, tematicas=[5, 6], coste=6.0, centro=1]
  Curso[id=C16, tematicas=[2, 3, 5], coste=1.0, centro=1]

- Solucion obtenida:
Cursos seleccionados = [C9, C12, C16]; Coste = 6,500000;
```

## 3. EJERCICIO 3

### 3.1. Modelo

- Modelo alternativo
  - $x_{ij}$ : variable entera, días que el investigador  $i$  dedica al trabajo  $j$ ,  $i \in [0, n]$ ,  $j \in [0, m]$

Un trabajo  $j$  se realiza si alguna  $x_{ij} > 0$ , por lo que podemos hacer un modelo sólo con  $x_{ij}$

$$\begin{aligned}
 & \max \sum_{j=0}^{m-1} c_j \\
 & \sum_{j=0}^{m-1} x_{ij} \leq dd_i, \quad i \in [0, n] \\
 & \sum_{l=0|e_l=k}^{n-1} x_{lj} = dn_{jk}, \quad j \in [0, m] \mid \exists_{l=0}^n x_{lj} > 0, k \in [0, e], \\
 & \quad \text{int } x_{lj}, \quad i \in [0, n], j \in [0, m]
 \end{aligned}$$

11

### **3.2. DatosInvestigadores**

```
package _datos;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import us.lsi.common.Files2;

public class DatosInvestigadores {

    public static List<Investigador> investigadores;
    public static List<Trabajo> trabajos;

    // Funcion que realiza la lectura de fichero y crea un tipo DatosInvestigadores
    public static void iniDatos(String fichero) {
        List<Investigador> listaInvestigadores = new ArrayList<>();
        List<Trabajo> listaTrabajos = new ArrayList<>();

        List<String> filas = Files2.linesFromFile(fichero);
        List<String> investigadoresString = filas.subList(1, filas.indexOf("// TRABAJOS"));
        List<String> trabajosString = filas.subList(filas.indexOf("// TRABAJOS") + 1, filas.size());

        // Transformamos una linea en el tipo Investigador y lo añadimos a la lista de investigadores
        for(String linea: investigadoresString) {
            Investigador inv = Investigador.parseaInvestigador(linea);
            listaInvestigadores.add(inv);
        }
        investigadores = listaInvestigadores;

        // Transformamos una linea en el tipo Trabajo y lo añadimos a la lista de trabajos
        for(String linea: trabajosString) {
            Trabajo t = Trabajo.parseaTrabajo(linea);
            listaTrabajos.add(t);
        }
        trabajos = listaTrabajos;

        toConsole();
    }

    // Para mostar el resultado por pantalla
    private static void toConsole() {
        System.out.println("- Investigadores leidos: ");
        for(Investigador inv: investigadores) {
            System.out.println(" " + inv);
        }
        System.out.println("- Trabajos leidos: ");
        for(Trabajo t: trabajos) {
            System.out.println(" " + t);
        }
    }

    // Funcion que me devuelve los investigadores
    public static List<Investigador> getInvestigadores() {
        return investigadores;
    }

    // Funcion que me devuelve el numero de investigadores
    public static Integer getNumInvestigadores() {
        return investigadores.size();
    }

    // Funcion que me devuelve el numero de especialidades
    public static Integer getNumEspecialidades() {
        Set<Integer> especialidades = new HashSet<>();
        for(Investigador inv: investigadores) {
            Integer especialidad = inv.especialidad();
            especialidades.add(especialidad);
        }
        return especialidades.size();
    }

    // Funcion que me devuelve los trabajos
    public static List<Trabajo> getTrabajos() {
        return trabajos;
    }
}
```

```

// Funcion que me devuelve el numero de trabajos
public static Integer getNumTrabajos() {
    return trabajos.size();
}

// Funcion que me devuelve un 1 si el investigador i tiene la especialidad k, sino devuelve un 0
public static Integer getInvestigadorTieneEspecialidad(Integer i, Integer k) {
    return investigadores.get(i).especialidad().equals(k)? 1:0;
}

// Funcion que me devuelve los dias disponibles del investigador i
public static Integer getDiasDisponiblesInvestigador(Integer i) {
    return investigadores.get(i).capacidad();
}

// Funcion que me devuelve los dias necesarios para el trabajo j del investigador con especialidad k
public static Integer getDiasNecesariosTrabajosEspecialidad(Integer j, Integer k) {
    return trabajos.get(j).repartos().stream()
        .filter(x -> x.especialidad().equals(k))
        .findFirst()
        .get()
        .dias();
}

// Funcion que me devuelve la calidad del trabajo j
public static Integer getCalidadTrabajo(Integer j) {
    return trabajos.get(j).calidad();
}

}

// TEST
public static void main(String[] args) {
    System.out.println("* TEST DatosInvestigadores *\n");
    iniDatos("ficheros/Ejercicio3DatosEntrada1.txt");
}
}

```

### 3.3. SolucionInvestigadores

```

package _soluciones;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.jgrapht.GraphPath;

import _datos.DatosInvestigadores;
import ejercicio3.InvestigadoresEdge;
import ejercicio3.InvestigadoresVertex;
import us.lsi.common.List2;

public class SolucionInvestigadores {

    // Propiedades
    private Map<String, List<Integer>> solucion;
    private Integer goal;
    private List<Integer> path;

    // getter del goal
    public Integer getGoal() {
        return goal;
    }

    // Metodo de factoria
    public static SolucionInvestigadores of(List<Integer> ls) {
        return new SolucionInvestigadores(ls);
    }

    public static SolucionInvestigadores of(GraphPath<InvestigadoresVertex, InvestigadoresEdge> path) {
        List<Integer> ls = path.getEdgeList().stream().map(x -> x.action()).toList();
        SolucionInvestigadores res = of(ls);
        res.path = ls;
        return res;
    }
}

```

```

// Constructor
public SolucionInvestigadores(List<Integer> ls) {
    List<Integer> lsCopia = List2.copy(ls);
    // Hallamos la suma de las calidades de los trabajos que se realizan
    Integer calidades = 0;
    for(int j=0; j<DatosInvestigadores.getNumTrabajos(); j++) {
        Boolean realiza = true;
        for(int k=0; k<DatosInvestigadores.getNumEspecialidades(); k++) {
            Integer diasPorEspecialidad = 0;
            for(int i=0; i<ls.size(); i++) {
                Integer numInvestigador = i/DatosInvestigadores.getNumTrabajos();
                Integer numTrabajo = i%DatosInvestigadores.getNumTrabajos();
                if(numTrabajo.equals(j)) &&
                    DatosInvestigadores.getInvestigadorTieneEspecialidad(numInvestigador, k).equals(1) {
                    diasPorEspecialidad += ls.get(i);
                }
            }
            if(diasPorEspecialidad != DatosInvestigadores.getDiasNecesariosTrabajosEspecialidad(j, k)) {
                realiza = false;
            }
        }
        if(realiza) {
            calidades += DatosInvestigadores.getCalidadTrabajo(j);
        }
    }
}

// Hallamos las horas que cada investigador dedica a cada trabajo
Map<String, List<Integer>> mp = new HashMap<>();
for(int i=0; i<DatosInvestigadores.getNumInvestigadores(); i++) {
    // Dividimos el cromosoma por investigadores
    Integer inicio = i*DatosInvestigadores.getNumTrabajos();
    List<Integer> valoresInvestigador = lsCopia.subList(inicio, inicio+DatosInvestigadores.getNumTrabajos());
    mp.put("INV"+(i+1), valoresInvestigador);
}

solucion = mp;
goal = calidades;
}

// Metodo toString que devuelve una cadena con lo que cada investigador dedica a cada trabajo y la calidad obtenida
@Override
public String toString() {
    String res = String.format("Reparto obtenido = %s;\n- Suma de las calidades de los trabajos realizados = %d",
        this.solucion, this.goal);
    return path==null? res: String.format("%s\nPath de la solucion = %s", res, this.path);
}
}

```

## 3.4. Ejercicio 3 – Automático

### 3.4.1. InvestigadoresVertex

```

package ejercicio3;

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

import _datos.DatosInvestigadores;
import us.lsi.common.List2;
import us.lsi.graphs.virtual.VirtualVertex;

public record InvestigadoresVertex(Integer index, List<Integer> days, List<List<Integer>> distribution)
    implements VirtualVertex<InvestigadoresVertex, InvestigadoresEdge, Integer> {

    // Metodo de factorial
    public static Investigadoresvertex of(Integer index, List<Integer> days,
        List<List<Integer>> distribution) {
        return new InvestigadoresVertex(index, days, distribution);
    }

    // Vertice inicial: el indice sera 0, los dias seran los dias que tiene disponible cada
    // investigador inicialmente y la distribucion sera una lista con m lista de enteros donde
    // cada lista tendra los dias necesarios inicialmente de cada especialidad en ese trabajo
    public static InvestigadoresVertex initial() {
        Integer index = 0;

        List<Integer> days = new ArrayList<>();
        for(int i=0; i<DatosInvestigadores.getNumInvestigadores(); i++) {
            Integer diasDisponibles = DatosInvestigadores.getDiasDisponiblesInvestigador(i);
            days.add(i, diasDisponibles);
        }

        List<List<Integer>> distribution = new ArrayList<>();
        for(int j=0; j<DatosInvestigadores.getNumTrabajos(); j++) {
            List<Integer> distributionTrabajo = new ArrayList<>();
            for(int k=0; k<DatosInvestigadores.getNumEspecialidades(); k++) {
                Integer diasEspecialidadTrabajo = DatosInvestigadores.getDiasNecesariosTrabajosEspecialidad(j, k);
                distributionTrabajo.add(k, diasEspecialidadTrabajo);
            }
            distribution.add(j, distributionTrabajo);
        }

        return of(index, days, distribution);
    }
}

```

```

// goal: predicado el cual es verdadero cuando ya hemos recorrido todas las variables, es decir,
// la n*m variables
public static Predicate<InvestigadoresVertex> goal() {
    return x -> x.index() == DatosInvestigadores.getNumInvestigadores()*DatosInvestigadores.getNumTrabajos();
}

// goalHasSolution: predicado el cual es verdadero cuando ya hemos recorrido todas los variables,
// es decir, la n*m variables
public static Predicate<InvestigadoresVertex> goalHasSolution() {
    return x -> x.index() == DatosInvestigadores.getNumInvestigadores()*DatosInvestigadores.getNumTrabajos();
}

// Alternativas a las que podemos ir desde ese vertice, es decir, el rango de dias que
// un investigador puede hacer en un trabajo determinado
@Override
public List<Integer> actions() {
    List<Integer> alternativas = new ArrayList<>();
    Integer numVariables = DatosInvestigadores.getNumInvestigadores()*DatosInvestigadores.getNumTrabajos();

    // en caso de que ya hayamos recorrido todas las variables no habra alternativas
    if(index() < numVariables) {
        List<Integer> dias = new ArrayList<>();

        // obtenemos los dias restantes del investigador y lo añadimos a la lista de dias
        Integer indiceInvestigador = index()/DatosInvestigadores.getNumTrabajos();
        Integer diasRestantesInvestigador = days().get(indiceInvestigador);
        dias.add(diasRestantesInvestigador);

        // obtenemos los dias restantes de la especialidad del investigador en el trabajo y
        // lo añadimos a la lista de dias
        Integer especialidad = DatosInvestigadores.getInvestigadores()
            .get(indiceInvestigador).especialidad();
        Integer indiceTrabajo = index()%DatosInvestigadores.getNumTrabajos();
        Integer diasRestantesEspecialidadTrabajo = distribution().get(indiceTrabajo).get(especialidad);
        dias.add(diasRestantesEspecialidadTrabajo);

        // cogemos el minimo de la lista dias que sera lo maximo que puede trabajar el
        // investigador en ese trabajo
        Integer maxDias = dias.stream().min(comparator.naturalOrder()).orElse(0);

        // las alternativas seran desde 0 hasta el maximo de dias posibles
        alternativas = List2.rangeList(0, maxDias+1); // +1 porque es rango abierto
    }

    return alternativas;
}

// Vertice vecino al que vamos, es decir, se incrementa el indice y si resulta que se le
// dedican mas de 0 dias se los restamos de los dias disponibles del investigador y se lo
// restamos a las horas necesarias de la especialidad de tal investigador en tal trabajo
@Override
public Investigadoresvertex neighbor(Integer a) {
    List<Integer> daysNeighbor = List2.copy(days());
    List<List<Integer>> distributionNeighbor = new ArrayList<>();
    for(int j=0; j<distribution().size(); j++) {
        List<Integer> trabajoNeighbor = List2.copy(distribution().get(j));
        distributionNeighbor.add(trabajoNeighbor);
    }

    if(a > 0) {
        // le restamos a las que haya trabajado al investigador
        Integer indiceInvestigador = index()/DatosInvestigadores.getNumTrabajos();
        Integer horasInvestigadorVecino = days().get(indiceInvestigador) - a;
        daysNeighbor.set(indiceInvestigador, horasInvestigadorVecino);

        // le restamos a las horas necesarias para la especialidad del investigador
        // en tal trabajo
        Integer especialidad = DatosInvestigadores.getInvestigadores()
            .get(indiceInvestigador).especialidad();
        Integer indiceTrabajo = index()%DatosInvestigadores.getNumTrabajos();
        List<Integer> diasEspecialidadTrabajo = distributionNeighbor.get(indiceTrabajo);
        Integer horasTrabajoEspecialidadVecino = diasEspecialidadTrabajo.get(especialidad) - a;
        diasEspecialidadTrabajo.set(especialidad, horasTrabajoEspecialidadVecino);
        distributionNeighbor.set(indiceTrabajo, diasEspecialidadTrabajo);
    }

    return of(index()+1, daysNeighbor, distributionNeighbor);
}

// Arista que construimos, en este caso, la accion sera el numero de horas que el investigador
// le ha dedicado al trabajo y el peso sera 0 (si no se completa un trabajo entre un vertice
// y el otro) o la calidad del trabajo (si entre un vertice y el otro si que se completa el trabajo)
@Override
public InvestigadoresEdge edge(Integer a) {
    return InvestigadoresEdge.of(this, neighbor(a), a);
}

// Metodo toString que define la representacion en forma de cadena de un vertice
@Override
public String toString() {
    return String.format("%d; %s; %s", this.index, this.days, this.distribution);
}
}

```

### **3.4.2. InvestigadoresEdge**

```
package ejercicio3;

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

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

    // Metodo de factorial
    public static InvestigadoresEdge of(InvestigadoresVertex source, InvestigadoresVertex target, Integer action) {
        Double weight = 0;

        // comprobamos si el trabajo que se esta haciendo se ha acaba o no
        Integer indiceTrabajo = source.index()%DatosInvestigadores.getNumTrabajos();
        Integer sumDiasTrabajoSource = source.distribution().get(indiceTrabajo).stream().mapToInt(x -> x).sum();
        Integer sumDiasTrabajoTarget = target.distribution().get(indiceTrabajo).stream().mapToInt(x -> x).sum();
        if(sumDiasTrabajoSource > 0 && sumDiasTrabajoTarget == 0) {
            weight += DatosInvestigadores.getCalidadTrabajo(indiceTrabajo);
        }

        return new InvestigadoresEdge(source, target, action, weight);
    }

    // Metodo toString que define la representacion en forma de cadena de una arista
    @Override
    public String toString() {
        return String.format("%d; %.2f", this.action, this.weight);
    }
}
```

### **3.4.3. InvestigadoresHeuristic**

```
package ejercicio3;

import java.util.function.Predicate;
import _datos.DatosInvestigadores;
public class InvestigadoresHeuristic {

    // Heuristica: lo mejor que puede pasar a partir de un vertice es que se cumplan todos los trabajos
    // restantes a partir de ese vertice (incluido el del vertice en el que estamos)
    public static Double heuristic(InvestigadoresVertex v1, Predicate<InvestigadoresVertex> goal,
        InvestigadoresVertex v2) {
        Double res = 0;

        Integer indiceInvestigador = v1.index()%DatosInvestigadores.getNumTrabajos();
        Integer ultimoInv = DatosInvestigadores.getNumInvestigadores()-1;
        // si el investigador es el ultimo miramos por que trabajo vamos para solo sumar a partir de ese
        if(indiceInvestigador == ultimoInv) {
            Integer indiceTrabajo = v1.index()%DatosInvestigadores.getNumTrabajos();
            for(int j=indiceTrabajo; j<DatosInvestigadores.getNumTrabajos(); j++) {
                res += DatosInvestigadores.getCalidadTrabajo(j);
            }
        } else { // si el investigador no es el ultimo se deben sumar de todos los trabajos
            for(int j=0; j<DatosInvestigadores.getNumTrabajos(); j++) {
                res += DatosInvestigadores.getCalidadTrabajo(j);
            }
        }
        return res;
    }
}
```

### **3.4.4. TestsEjercicio3**

```
package ejercicios.tests;

import java.util.List;

import _datos.DatosInvestigadores;
import _soluciones.SolucionInvestigadores;
import _utils.GraphsPI5;
import _utils.TestsPI5;
import ejercicio3.Investigadoresvertex;

public class TestsEjercicio3 {

    public static void main(String[] args) {
        System.out.println("TESTS EJERCICIO 3");
        TestsPI5.line(".");
        TestsPI5.line("...");

        List.of(1,2).forEach(num_test -> {
            TestsPI5.initTest("Ejercicio3DatosEntrada", num_test, DatosInvestigadores::iniDatos);
            TestsPI5.tests(
                Investigadoresvertex.initial(),           // Vertice Inicial
                Investigadoresvertex.goal(),              // Predicado para un vertice final
                GraphsPI5::investigadoresBuilder,         // Referencia al Builder del grafo
                SolucionInvestigadores::of);             // Referencia al metodo factoria para la solucion
        });
    }
}
```

### **3.4.5. Volcado de pantalla**

No para los datos de entrada 3 porque sale una excepción de falta de memoria

```
* TESTS EJERCICIO 3 *
=====
- Investigadores leidos:
Investigador[nombre=INV1, capacidad=6, especialidad=0]
Investigador[nombre=INV2, capacidad=3, especialidad=1]
Investigador[nombre=INV3, capacidad=8, especialidad=2]
- Trabajos leidos:
Trabajo[nombre=T01, calidad=5, repartos=[TuplaEj3[especialidad=0, dias=6], TuplaEj3[especialidad=1, dias=0], TuplaEj3[especialidad=2, dias=0]]]
Trabajo[nombre=T02, calidad=10, repartos=[TuplaEj3[especialidad=0, dias=0], TuplaEj3[especialidad=1, dias=3], TuplaEj3[especialidad=2, dias=8]]]
=====
Solucion A*: Reparto obtenido = {INV3=[0, 0], INV1=[6, 0], INV2=[0, 0]};
- Suma de las calidades de los trabajos realizados = 5
Path de la solucion = [6, 0, 0, 0, 0]
=====
Solucion PDR: Reparto obtenido = {INV3=[0, 8], INV1=[6, 0], INV2=[0, 3]};
- Suma de las calidades de los trabajos realizados = 15
Path de la solucion = [6, 0, 0, 0, 3, 0, 8]
=====
Solucion BT: Reparto obtenido = {INV3=[0, 8], INV1=[6, 0], INV2=[0, 3]};
- Suma de las calidades de los trabajos realizados = 15
Path de la solucion = [6, 0, 0, 0, 3, 0, 8]
=====
- Investigadores leidos:
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]
- Trabajos leidos:
Trabajo[nombre=T01, calidad=7, repartos=[TuplaEj3[especialidad=0, dias=2], TuplaEj3[especialidad=1, dias=0], TuplaEj3[especialidad=2, dias=5], TuplaEj3[especialidad=3, dias=0]]]
Trabajo[nombre=T02, calidad=9, repartos=[TuplaEj3[especialidad=0, dias=8], TuplaEj3[especialidad=1, dias=4], TuplaEj3[especialidad=2, dias=3], TuplaEj3[especialidad=3, dias=0]]]
Trabajo[nombre=T03, calidad=5, repartos=[TuplaEj3[especialidad=0, dias=2], TuplaEj3[especialidad=1, dias=0], TuplaEj3[especialidad=2, dias=0], TuplaEj3[especialidad=3, dias=7]]]
=====
Solucion A*: Reparto obtenido = {INV3=[5, 3, 0], INV4=[2, 0, 0], INV1=[0, 8, 2], INV2=[0, 4, 0], INV5=[0, 0, 0]};
- Suma de las calidades de los trabajos realizados = 16
Path de la solucion = [0, 8, 2, 0, 4, 0, 5, 3, 0, 2, 0, 0, 0, 0, 0]
=====
Solucion PDR: Reparto obtenido = {INV3=[5, 3, 0], INV4=[2, 0, 0], INV1=[0, 8, 0], INV2=[0, 4, 0], INV5=[0, 0, 0]};
- Suma de las calidades de los trabajos realizados = 16
Path de la solucion = [0, 8, 0, 0, 4, 0, 5, 3, 0, 2, 0, 0, 0, 0, 0]
=====
```

## 3.5. Ejercicio 3 – Manual (BT)

### 3.5.1. InvestigadoresProblem

```
package ejercicio3.manual;

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

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

public record InvestigadoresProblem(Integer index, List<Integer> days, List<List<Integer>> distribution) {

    // Metodo de factorial
    public static InvestigadoresProblem of(Integer index, List<Integer> days, List<List<Integer>> distribution) {
        return new InvestigadoresProblem(index, days, distribution);
    }

    // Vertice inicial: el indice sera 0, los dias seran los dias que tiene disponible cada
    // investigador inicialmente y la distribucion sera una lista con m lista de enteros donde
    // cada lista tendra los dias necesarios inicialmente de cada especialidad en ese trabajo
    public static InvestigadoresProblem inicial() {
        Integer index = 0;

        List<Integer> days = new ArrayList<>();
        for (int i = 0; i < DatosInvestigadores.getNumInvestigadores(); i++) {
            Integer diasDisponibles = DatosInvestigadores.getDiasDisponiblesInvestigador(i);
            days.add(i, diasDisponibles);
        }

        List<List<Integer>> distribution = new ArrayList<>();
        for (int j = 0; j < DatosInvestigadores.getNumTrabajos(); j++) {
            List<Integer> distributionTrabajo = new ArrayList<>();
            for (int k = 0; k < DatosInvestigadores.getNumEspecialidades(); k++) {
                Integer diasEspecialidadTrabajo = DatosInvestigadores.getDiasNecesariosTrabajosEspecialidad(j, k);
                distributionTrabajo.add(k, diasEspecialidadTrabajo);
            }
            distribution.add(j, distributionTrabajo);
        }

        return of(index, days, distribution);
    }

    // Alternativas a las que podemos ir desde ese vertice, es decir, el rango de dias que
    // un investigador puede hacer en un trabajo determinado
    public List<Integer> actions() {
        List<Integer> alternativas = new ArrayList<>();
        Integer numVariables = DatosInvestigadores.getNumInvestigadores() * DatosInvestigadores.getNumTrabajos();

        // en caso de que ya hayamos recorrido todas las variables no habra alternativas
        if (index() < numVariables) {
            List<Integer> dias = new ArrayList<>();

            // obtenemos los dias restantes del investigador y lo añadimos a la lista de dias
            Integer indiceInvestigador = index() / DatosInvestigadores.getNumTrabajos();
            Integer diasRestantesInvestigador = days().get(indiceInvestigador);
            dias.add(diasRestantesInvestigador);

            // obtenemos los dias restantes de la especialidad del investigador en el trabajo y
            // lo añadimos a la lista de dias
            Integer especialidad = DatosInvestigadores.getInvestigadores().get(indiceInvestigador).especialidad();
            Integer indiceTrabajo = index() % DatosInvestigadores.getNumTrabajos();
            Integer diasRestantesEspecialidadTrabajo = distribution().get(indiceTrabajo).get(especialidad);
            dias.add(diasRestantesEspecialidadTrabajo);

            // cogemos el minimo de la lista dias que sera lo maximo que puede trabajar el
            // investigador en ese trabajo
            Integer maxDias = dias.stream().min(Comparator.naturalOrder()).orElse(0);

            // las alternativas seran desde 0 hasta el maximo de dias posibles
            alternativas = List2.rangeList(0, maxDias + 1); // +1 porque es rango abierto
        }

        return alternativas;
    }

    // Vertice vecino al que vamos, es decir, se incrementa el indice y si resulta que se le
    // dedican mas de 0 dias se los restamos de los dias disponibles del investigador y se lo
    // restamos a las horas necesarias de la especialidad de tal investigador en tal trabajo
    public InvestigadoresProblem neighbor(Integer a) {
        List<Integer> daysNeighbor = List2.copy(days());
        List<List<Integer>> distributionNeighbor = List2.copy(distribution());

        if (a > 0) {
            // le restamos a las que haya trabajado al investigador
            Integer indiceInvestigador = index() / DatosInvestigadores.getNumTrabajos();
            Integer horasInvestigadorVecino = days().get(indiceInvestigador) - a;
            daysNeighbor.set(indiceInvestigador, horasInvestigadorVecino);

            // le restamos a las horas necesarias para la especialidad del investigador
            // en tal trabajo
            Integer especialidad = DatosInvestigadores.getInvestigadores().get(indiceInvestigador).especialidad();
            Integer indiceTrabajo = index() % DatosInvestigadores.getNumTrabajos();
            List<Integer> diasEspecialidadTrabajo = List2.copy(distribution()).get(indiceTrabajo);
            Integer horasTrabajoEspecialidadVecino = diasEspecialidadTrabajo.get(especialidad) - a;
            diasEspecialidadTrabajo.set(especialidad, horasTrabajoEspecialidadVecino);
            distributionNeighbor.set(indiceTrabajo, diasEspecialidadTrabajo);
        }

        return of(index() + 1, daysNeighbor, distributionNeighbor);
    }
}
```

```

// Heuristica: lo mejor que puede pasar a partir de un vertice es que se cumplan todos los trabajos
// restantes a partir de ese vertice (incluido el del vertice en el que estamos)
public Double heuristic() {
    Double res = 0.;

    Integer indiceInvestigador = index() / DatosInvestigadores.getNumTrabajos();
    Integer ultimoInv = DatosInvestigadores.getNumInvestigadores() - 1;
    // si el investigador es el ultimo miramos por que trabajo vamos para solo sumar
    // a partir de ese
    if (indiceInvestigador == ultimoInv) {
        Integer indiceTrabajo = index() % DatosInvestigadores.getNumTrabajos();
        for (int j = indiceTrabajo; j < DatosInvestigadores.getNumTrabajos(); j++) {
            res += DatosInvestigadores.getCalidadTrabajo(j);
        }
    } else { // si el investigador no es el ultimo se deben sumar de todos los trabajos
        for (int j = 0; j < DatosInvestigadores.getNumTrabajos(); j++) {
            res += DatosInvestigadores.getCalidadTrabajo(j);
        }
    }

    return res;
}

```

### 3.5.2. InvestigadoresState

```

package ejercicio3.manual;

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

import _datos.DatosInvestigadores;
import _soluciones.SolucionInvestigadores;

public class InvestigadoresState {

    // Propiedades del estado
    InvestigadoresProblem actual;
    Double acumulado;
    List<Integer> acciones;
    List<InvestigadoresProblem> anteriores;

    // Constructor
    private InvestigadoresState(InvestigadoresProblem prob, Double ac, List<Integer> ls1,
                                List<InvestigadoresProblem> ls2) {
        actual = prob;
        acumulado = ac;
        acciones = ls1;
        anteriores = ls2;
    }

    // Metodo de factorya
    public static InvestigadoresState of(InvestigadoresProblem prob, Double ac, List<Integer> ls1,
                                         List<InvestigadoresProblem> ls2) {
        return new InvestigadoresState(prob, ac, ls1, ls2);
    }

    // Estado inicial: lo obtenemos a partir del problema inicial
    public static InvestigadoresState initial() {
        InvestigadoresProblem probInicial = InvestigadoresProblem.initial();
        List<Integer> acciones = new ArrayList<>();
        List<InvestigadoresProblem> anteriores = new ArrayList<>();
        return of(probInicial, 0., acciones, anteriores);
    }

    // forward: como cambia el estado al avanzar, es decir, al acumulado sumarle la calidad del trabajo
    // si se ha acabado el trabajo, añadir a la lista de acciones la accion que tomamos, añadir a
    // anteriores el problema actual y el problema actual ahora sera el vecino
    public void forward(Integer a) {
        Double calidad = 0.;
        // comprobamos si el trabajo que se esta haciendo se acaba o no
        Integer indiceTrabajo = actual.index() % DatosInvestigadores.getNumTrabajos();
        Integer sumDiasTrabajoSource = actual.distribution().get(indiceTrabajo).stream()
            .mapToInt(x -> x).sum();
        Integer sumDiasTrabajoTarget = actual.neighbor(a).distribution().get(indiceTrabajo).stream()
            .mapToInt(x -> x).sum();
        if (sumDiasTrabajoSource > 0 && sumDiasTrabajoTarget == 0) {
            calidad += DatosInvestigadores.getCalidadTrabajo(indiceTrabajo);
        }

        acumulado += calidad;
        acciones.add(a);
        anteriores.add(actual);
        actual = actual.neighbor(a);
    }

    // back: como cambia el estado al retroceder, es decir, al acumulado hay que restarle la calidad
    // si es que el trabajo se habia terminado en ese paso, quitar la ultima accion de la lista de
    // acciones, quitar de la lista de anteriores el ultimo visto y el actual pasa a ser este ultimo
    // que hemos quitado
    public void back() {
        int last = acciones.size() - 1;
        InvestigadoresProblem prob_ant = anteriores.get(last);

        Double calidad = 0.;
        // comprobamos si el trabajo que se habia acabado en el anterior paso
        Integer indiceTrabajo = prob_ant.index() % DatosInvestigadores.getNumTrabajos();
        Integer sumDiasTrabajoActual = actual.distribution().get(indiceTrabajo).stream()
            .mapToInt(x -> x).sum();
        Integer sumDiasTrabajoAnterior = prob_ant.distribution().get(indiceTrabajo).stream()
            .mapToInt(x -> x).sum();
        if (sumDiasTrabajoAnterior > 0 && sumDiasTrabajoActual == 0) {
            calidad += DatosInvestigadores.getCalidadTrabajo(indiceTrabajo);
        }

        acumulado -= calidad;
        acciones.remove(last);
        anteriores.remove(last);
        actual = prob_ant;
    }
}

```

```

}

// Alternativas: alternativas del estado actual
public List<Integer> alternativas() {
    return actual.actions();
}

// Cota: sirve para saber que es lo mejor que puede pasar desde un investigador en un trabajo
// tomando la alternativa a
public Double cota(Integer a) {
    Double weight = 0.;
    // comprobamos si el trabajo que se esta haciendo se acaba o no
    Integer indiceTrabajo = actual.index()%DatosInvestigadores.getNumTrabajos();
    Integer sumDiasTrabajoSource = actual.distribution().get(indiceTrabajo).stream()
        .mapToInt(x -> x).sum();
    Integer sumDiasTrabajoTarget = actual.neighbor(a).distribution().get(indiceTrabajo).stream()
        .mapToInt(x -> x).sum();
    if(sumDiasTrabajoSource > 0 && sumDiasTrabajoTarget == 0) {
        weight += DatosInvestigadores.getCalidadTrabajo(indiceTrabajo);
    }
    return acumulado + weight + actual.neighbor(a).heuristic();
}

// esSolucion: recorrer todos los variables posibles (todos los trabajos por todos los investigadores
public Boolean esSolucion() {
    return actual.index() == DatosInvestigadores.getNumInvestigadores()*DatosInvestigadores.getNumTrabajos();
}

// esTerminal: recorrer todos los variables posibles (todos los trabajos por todos los investigadores
public Boolean esTerminal() {
    return actual.index() == DatosInvestigadores.getNumInvestigadores()*DatosInvestigadores.getNumTrabajos();
}

// getSolucion: devuelve la solucion del estado en el que estamos
public SolucionInvestigadores getSolucion() {
    return SolucionInvestigadores.of(acciones);
}
}

```

### 3.5.3. InvestigadoresBT

```

package ejercicio3.manual;

import java.util.HashSet;

public class InvestigadoresBT {

    // Propiedades
    public static Double mejorValor;
    private static InvestigadoresState estado;
    public static Set<SolucionInvestigadores> soluciones;

    // Funcion que da los valores iniciales y llama al algoritmo en si
    public static void search() {
        soluciones = new HashSet<>();
        mejorValor = 0.; // estamos maximizando
        estado = InvestigadoresState.initial();
        bt_search();
    }

    // Funcion que es basicamente el algoritmo
    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();
                }
            }
        }
    }

    // Funcion que devuelve las soluciones del problema
    public static Set<SolucionInvestigadores> getSoluciones() {
        return soluciones;
    }
}

```

### 3.5.4. TestsEjercicio3BT

```
package ejercicios.tests.manual;

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

import _datos.DatosInvestigadores;
import _soluciones.SolucionInvestigadores;
import _utils.TestsPIS;
import ejercicios.manual.InvestigadoresBT;

public class TestsEjercicio3BT {

    public static void main(String[] args) {
        System.out.println(" TESTS EJERCICIO 3 - BT MANUAL *\n");
        TestsPIS.Line("**");

        List.of(3).forEach(num_test -> {
            DatosInvestigadores.iniDatos("ficheros/Ejercicio3DatosEntrada"+num_test+".txt");
            InvestigadoresBT.search();
            SolucionInvestigadores sol = InvestigadoresBT.getSoluciones().stream().max(Comparator.comparing(SolucionInvestigadores::getGoal)).orElse(null);
            String res = "\n- Solucion obtenida: "+ sol + "\n";
            System.out.println(res);
            TestsPIS.Line("**");
        });
    }
}
```

### 3.5.5. Volcado de pantalla

No para los datos de entrada 3 porque sale una excepción de falta de memoria

```
* TESTS EJERCICIO 3 - BT MANUAL *

*****
- Investigadores leidos:
Investigador[nombre=INV1, capacidad=6, especialidad=0]
Investigador[nombre=INV2, capacidad=3, especialidad=1]
Investigador[nombre=INV3, capacidad=8, especialidad=2]
- Trabajos leidos:
Trabajo[nombre=T01, calidad=5, repartos=[TuplaEj3[especialidad=0, dias=6], TuplaEj3[especialidad=1, dias=0], TuplaEj3[especialidad=2, dias=0]]]
Trabajo[nombre=T02, calidad=10, repartos=[TuplaEj3[especialidad=0, dias=0], TuplaEj3[especialidad=1, dias=3], TuplaEj3[especialidad=2, dias=8]]]
- Solucion obtenida: Reparto obtenido = {INV3=[0, 8], INV1=[6, 0], INV2=[0, 3]};
- Suma de las calidades de los trabajos realizados = 15

*****
- Investigadores leidos:
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]
- Trabajos leidos:
Trabajo[nombre=T01, calidad=7, repartos=[TuplaEj3[especialidad=0, dias=2], TuplaEj3[especialidad=1, dias=0], TuplaEj3[especialidad=2, dias=5], TuplaEj3[especialidad=3, dias=0]]]
Trabajo[nombre=T02, calidad=9, repartos=[TuplaEj3[especialidad=0, dias=8], TuplaEj3[especialidad=1, dias=4], TuplaEj3[especialidad=2, dias=3], TuplaEj3[especialidad=3, dias=0]]]
Trabajo[nombre=T03, calidad=5, repartos=[TuplaEj3[especialidad=0, dias=2], TuplaEj3[especialidad=1, dias=0], TuplaEj3[especialidad=2, dias=0], TuplaEj3[especialidad=3, dias=7]]]
- Solucion obtenida: Reparto obtenido = {INV3=[5, 3, 0], INV4=[2, 0, 0], INV1=[0, 8, 0], INV2=[0, 4, 0], INV5=[0, 0, 0]};
- Suma de las calidades de los trabajos realizados = 16

*****
```

## 4. EJERCICIO 4

### 4.1. Modelo

#### Modelo alternativo

$x_i$ : variable entera cuyo valor será el índice del vértice que ocupa la posición  $i$  en el camino ( $x_0$  es el almacen)

$$\max \sum_{i=1}^{n-1} b_i - \sum_{i=0}^{n-1} (n-i) w(x_i, x_{(i+1)\%n})$$

$$P_{i=0}^{n-1}(x_i, i)$$

$$x_0 = a$$

$$CP_{i=0|g}^{n-1} x_i$$

$$x_i < n, i \in [0, n]$$

$$\text{int } x_i, i \in [0, n]$$

$CP_{i=0|g}^{n-1} f(i)$ : *Closed Path*. Los valores de  $f(i)$  son los vértices de un grafo  $g$  y forman un camino cerrado en el mismo

$P_{i=0}^{n-1}(f(i), v(i))$ : *Permutación*. Es una restricción que obliga a la lista de  $f(i)$  a sea una permutación de  $v(i)$ .

### 4.2. DatosRepartidor

```
package _datos;

import org.jgrapht.Graph;
import us.lsi.graphs.Graphs2;
import us.lsi.graphs.GraphsReader;

public class DatosRepartidor {

    public static Graph<Cliente, Conexion> grafo;

    // Funcion que realiza la lectura de fichero y crea un tipo DatosCafes
    public static void iniDatos(String fichero) {
        grafo = GraphsReader.newGraph(fichero,
            Cliente::ofFormat, Conexion::ofFormat,
            Graphs2::simpleWeightedGraph);

        toConsole();
    }

    // Para mostar el resultado por pantalla
    private static void toConsole() {
        System.out.println("- Número de clientes leidos: " + grafo.vertexSet().size());
        System.out.println("- Clientes leidos: ");
        for(Cliente c: grafo.vertexSet()) {
            System.out.println(" " + c);
        }
        System.out.println("\n- Número de conexiones leidas: " + grafo.edgeSet().size());
        System.out.println("- Conexiones leidas: ");
        for(Conexion co: grafo.edgeSet()) {
            System.out.println(" " + co);
        }
    }
}
```

```

// Funcion auxiliar que devuelve el cliente i
private static Cliente getCliente(Integer i) {
    return grafo.vertexSet().stream()
        .filter(x -> x.clienteId().equals(i))
        .findFirst().orElse(null);
}

// Funcion que devuelve el numero de vertices
public static IntegergetNumVertices() {
    return grafo.vertexSet().size();
}

// Funcion que devuelve el peso de la arista que une el Cliente i y el Cliente j
public static Double getPesoArista(Integer i, Integer j) {
    return grafo.getEdge(getCliente(i), getCliente(j)).distanciaKm();
}

// Funcion que devuelve el beneficio que da el Cliente i
public static Double getBeneficioCliente(Integer i) {
    return grafo.vertexSet().stream()
        .filter(x -> x.clienteId().equals(i))
        .findFirst().get()
        .beneficio();
}

// Funcion que dice si existe una arista en el grafo
public static Boolean existeArista(Integer i, Integer j) {
    return grafo.containsEdge(getCliente(i), getCliente(j));
}

// TEST
public static void main(String[] args) {
    System.out.println("** TEST DatosRepartidor *\n");
    iniDatos("ficheros/Ejercicio4DatosEntrada1.txt");
}
}

```

### **4.3. SolucionRepartidor**

```

package _soluciones;

import java.util.List;

import org.jgrapht.GraphPath;

import _datos.DatosRepartidor;
import ejercicio4.RepartidorEdge;
import ejercicio4.RepartidorVertex;
import us.lsi.common.List2;

public class SolucionRepartidor {

    // Propiedades
    private List<Integer> camino;
    private Double distancia;
    private Double beneficio;
    private List<Integer> path;

    // getter del beneficio
    public Double getBeneficio() {
        return beneficio;
    }

    // Metodo de factory
    public static SolucionRepartidor of(List<Integer> ls) {
        List<Integer> lsCopy = List2.copy(ls); // para que me deje añadirle el 0 en la primera posicion
        return new SolucionRepartidor(lsCopy);
    }

    public static SolucionRepartidor of(GraphPath<RepartidorVertex, RepartidorEdge> path) {
        List<Integer> ls = path.getEdgeList().stream().map(x -> x.action()).toList();
        List<Integer> camino = List2.copy(ls); // para que me deje añadirle el 0 en la primera posicion
        SolucionRepartidor res = of(camino);
        res.path = ls;
        return res;
    }
}

```

```

// Constructor
public SolucionRepartidor(List<Integer> ls) {
    Double km = 0.;
    Double beneficios = 0.;

    // Recorremos el cromosoma para hallar la distancia recorrida y el beneficio
    for(int i=0; i<ls.size(); i++) {
        // Beneficio += beneficio - cada minuto que se pase (ponemos los km, ya que tarda 1 minuto en hacer 1km)
        // Si no existe la arista aumentamos el error
        if(i==0) {
            km += DatosRepartidor.getPesoArista(0, ls.get(i));
            beneficios += DatosRepartidor.getBeneficioCliente(ls.get(i)) - km;
        } else {
            km += DatosRepartidor.getPesoArista(ls.get(i-1), ls.get(i));
            beneficios += DatosRepartidor.getBeneficioCliente(ls.get(i)) - km;
        }
    }
    // Añadimos a la posicion 0 el inicial, es decir, el 0
    ls.add(0, 0);

    camino = ls;
    distancia = km;
    beneficio = beneficios;
}

// Metodo toString que devuelve una cadena con el orden de clientes que ha seguido,
// la distancia recorrida en km y el beneficio total
@Override
public String toString() {
    String res = String.format("\nCamino desde 0 hasta 0: %s;\nKms: %f;\nBeneficio: %f;",
        this.camino, this.distancia, this.beneficio);
    return path==null? res: String.format("%s\nPath de la solucion = %s", res, this.path);
}
}

```

## 4.4. Ejercicio 4 – Automático

### 4.4.1. RepartidorVertex

```

package ejercicio4;

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

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

public record RepartidorVertex(Integer cliente, Set<Integer> pendientes, List<Integer> visitados,
    Double kms) implements VirtualVertex<RepartidorVertex, RepartidorEdge, Integer> {

    // Metodo de factoria
    public static RepartidorVertex of(Integer cliente, Set<Integer> pendientes, List<Integer> visitados,
        Double kms) {
        return new RepartidorVertex(cliente, pendientes, visitados, kms);
    }

    // Vertice inicial: el cliente es el 0, los pendientes son todos los vertices que hay en el grafo,
    // los visitados es una lista vacia y los km recorridos son 0
    public static RepartidorVertex initial() {
        List<Integer> visitados = new ArrayList<>();

        Set<Integer> pendientes = new HashSet<>();
        for(int i=0; i<DatosRepartidor.getNumVertices(); i++) {
            pendientes.add(i);
        }

        return of(0, pendientes, visitados, 0.);
    }

    // goal: recorrer todos los vertices del grafo, es decir, que el cliente final sea el numero de
    // vertices que hay en el grafo
    public static Predicate<RepartidorVertex> goal() {
        return x -> x.cliente() == DatosRepartidor.getNumVertices();
    }

    // goalHasSolution: que no queden vertices por visitar, es decir, que los pendientes sea un conjunto
    // vacio
    public static Predicate<RepartidorVertex> goalHasSolution() {
        return x -> x.pendientes().size() == 0;
    }
}

```

#### 4.4.2.

```
// Alternativas a las que podemos ir, es decir, a los vertices (del grafo) a los que podemos ir desde
// el ultimo que hemos visitado
@Override
public List<Integer> actions() {
    List<Integer> alternativas = new ArrayList<>();
    List<Integer> pendientesVisitar = pendientes().stream().toList();

    // si ya hemos recorrido todos los vertices no habra alternativas
    if(cliente() < DatosRepartidor.getNumVertices()) {
        // si estamos en el cliente 0, tenemos que ver aquellos que esten conectados al vertice
        // origen, es decir, el 0, quitando obviamente el 0
        if(cliente() == 0) {
            for(Integer vertice: pendientesVisitar) {
                if(vertice != 0 && DatosRepartidor.existeArista(0, vertice)) {
                    alternativas.add(vertice);
                }
            }
        } // si estamos en el ultimo cliente, tenemos que ver que el ultimo visitado pueda conectarse
        // con el destino que es el 0
        else if(cliente() == DatosRepartidor.getNumVertices()-1) {
            if(pendientesVisitar.contains(0) &&
                DatosRepartidor.existeArista(visitados().get(visitados().size()-1), 0)) {
                alternativas.add(0);
            }
        } // si estamos en el cualquier otro vertice, tenemos que mirar aquellos que queden que puedan
        // estar conectados con el ultimo vertice visitado quitando el 0 que sera el destino final
        else {
            for(Integer vertice: pendientesVisitar) {
                if(vertice != 0 &&
                    DatosRepartidor.existeArista(visitados().get(visitados().size()-1), vertice)) {
                        alternativas.add(vertice);
                }
            }
        }
    }
    return alternativas;
}

// vecino al que vamos, es decir, incrementamos el indice, quitamos la accion (vertice al que vamos)
// del conjunto de pendientes, añadimos la accion al conjunto de vertices visitados y aumentamos los
// kms con el peso de la arista entre vertices
@Override
public RepartidorVertex neighbor(Integer a) {
    // eliminamos de los pendientes
    Set<Integer> pendientesVecino = Set2.copyOf(pendientes());
    pendientesVecino.remove(a);

    // añadimos a los visitados
    List<Integer> visitadosVecino = List2.copyOf(visitados());
    visitadosVecino.add(a);

    // añadimos kms recorridos
    Double kmsVecino = kms();
    if(cliente() == 0) {
        kmsVecino += DatosRepartidor.getPesoArista(0, a);
    } else {
        kmsVecino += DatosRepartidor.getPesoArista(visitados().get(visitados().size()-1), a);
    }

    return of(cliente()+1, pendientesVecino, visitadosVecino, kmsVecino);
}

// Arista que construimos, en este caso, donde la accion sera el cliente al que vamos y donde el
// peso sera el beneficio de ir a ese cliente menos los kms que recorremos hasta llegar a ese
// vertice (es decir, los que llevamos vamos mas los que hay al que vamos a ir)
@Override
public RepartidorEdge edge(Integer a) {
    return RepartidorEdge.of(this, neighbor(a), a);
}

// Metodo toString que define la representacion en forma de cadena de un vertice
@Override
public String toString() {
    return String.format("%d; %s; %s; %f", this.cliente, this.pendientes, this.visitados, this.kms);
}
```

#### 4.4.3. RepartidorEdge

```

package ejercicio4;

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

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

    // Metodo de factorial
    public static RepartidorEdge of(RepartidorVertex source, RepartidorVertex target, Integer action) {
        Double weight = DatosRepartidor.getBeneficioCliente(action);

        Double kmsRecorridos = source.kms();
        if(source.cliente() == 0) {
            kmsRecorridos += DatosRepartidor.getPesoArista(0, action);
        } else {
            kmsRecorridos += DatosRepartidor.getPesoArista(source.visitados().get(source.visitados().size()-1), action);
        }

        weight -= kmsRecorridos;
    }

    return new RepartidorEdge(source, target, action, weight);
}

// Metodo toString que define la representacion en forma de cadena de una arista
@Override
public String toString() {
    return String.format("%d; %.2f", this.action, this.weight);
}
}

```

#### **4.4.4. RepartidorHeuristic**

```

package ejercicio4;

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

import _datos.DatosRepartidor;

public class RepartidorHeuristic {

    // Heuristica: lo mejor que puede pasar a partir de un vertice es que se consigan todos los beneficios
    // de los clientes (vertices) que quedan por ir y restandole los kms suponiendo que la distancia minima
    // es 1
    public static Double heuristic(RepartidorVertex v1, Predicate<RepartidorVertex> goal, RepartidorVertex v2) {
        Double beneficio = 0.;

        List<Integer> pendientes = v1.pendientes().stream().toList(); // para poder recorrerlo
        for(Integer vertice: pendientes) {
            beneficio += DatosRepartidor.getBeneficioCliente(vertice);
        }
        Double kms = IntStream.range(1, v1.pendientes().size()).mapToDouble(x -> v1.kms() + x).sum();

        return beneficio - kms - 1;
    }
}

```

#### **4.4.5. TestsEjercicio4**

```

package ejercicios.tests;

import java.util.List;

import _datos.DatosRepartidor;
import _soluciones.SolucionRepartidor;
import _utils.GraphsPIS;
import _utils.TestsPIS;
import ejercicio4.RepartidorVertex;

public class TestsEjercicio4 {

    public static void main(String[] args) {
        System.out.println("* TESTS EJERCICIO 4 *");
        TestsPIS.line(" ");
        TestsPIS.line(" ");

        List.of(1,2).forEach(num_test -> {
            TestsPIS.initTest("Ejercicio4DatosEntrada", num_test, DatosRepartidor::iniDatos);
            TestsPIS.tests(
                RepartidorVertex.initial(), // Vertice Inicial
                RepartidorVertex.goal(), // Predicado para un vertice final
                GraphsPIS::repartidorBuilder, // Referencia al Builder del grafo
                SolucionRepartidor::of); // Referencia al metodo factorial para la solucion
        });
    }
}

```

#### **4.4.6. Volcado de pantalla**

\* TESTS EJERCICIO 4 \*

\*\*\*\*\*

- Numero de clientes leidos: 5  
- Clientes leidos:  
  Cliente[clienteId=0, beneficio=0.0]  
  Cliente[clienteId=1, beneficio=400.0]  
  Cliente[clienteId=2, beneficio=300.0]  
  Cliente[clienteId=3, beneficio=200.0]  
  Cliente[clienteId=4, beneficio=100.0]

- Numero de conexiones leidas: 8  
- Conexiones leidas:  
  Conexion[conexionId=0, distanciaKm=1.0]  
  Conexion[conexionId=1, distanciaKm=100.0]  
  Conexion[conexionId=2, distanciaKm=1.0]  
  Conexion[conexionId=3, distanciaKm=100.0]  
  Conexion[conexionId=4, distanciaKm=1.0]  
  Conexion[conexionId=5, distanciaKm=1.0]  
  Conexion[conexionId=6, distanciaKm=100.0]  
  Conexion[conexionId=7, distanciaKm=5.0]

=====

Solucion A\*:  
Camino desde 0 hasta 0: [0, 1, 2, 3, 4, 0];  
Kms: 9,000000;  
Beneficio: 981,000000;  
Path de la solucion = [1, 2, 3, 4, 0]

Solucion PDR:  
Camino desde 0 hasta 0: [0, 1, 2, 3, 4, 0];  
Kms: 9,000000;  
Beneficio: 981,000000;  
Path de la solucion = [1, 2, 3, 4, 0]

Solucion BT:  
Camino desde 0 hasta 0: [0, 1, 2, 3, 4, 0];  
Kms: 9,000000;  
Beneficio: 981,000000;  
Path de la solucion = [1, 2, 3, 4, 0]

\*\*\*\*\*

- Numero de clientes leidos: 8  
- Clientes leidos:  
  Cliente[clienteId=0, beneficio=0.0]  
  Cliente[clienteId=1, beneficio=100.0]  
  Cliente[clienteId=2, beneficio=200.0]  
  Cliente[clienteId=3, beneficio=300.0]  
  Cliente[clienteId=4, beneficio=200.0]  
  Cliente[clienteId=5, beneficio=300.0]  
  Cliente[clienteId=6, beneficio=200.0]  
  Cliente[clienteId=7, beneficio=200.0]

- Numero de conexiones leidas: 13  
- Conexiones leidas:  
  Conexion[conexionId=8, distanciaKm=2.0]  
  Conexion[conexionId=9, distanciaKm=1.0]  
  Conexion[conexionId=10, distanciaKm=1.0]  
  Conexion[conexionId=11, distanciaKm=3.0]  
  Conexion[conexionId=12, distanciaKm=1.0]  
  Conexion[conexionId=13, distanciaKm=1.0]  
  Conexion[conexionId=14, distanciaKm=3.0]  
  Conexion[conexionId=15, distanciaKm=1.0]  
  Conexion[conexionId=16, distanciaKm=1.0]  
  Conexion[conexionId=17, distanciaKm=3.0]  
  Conexion[conexionId=18, distanciaKm=1.0]  
  Conexion[conexionId=19, distanciaKm=1.0]  
  Conexion[conexionId=20, distanciaKm=1.0]

=====

Solucion A\*:  
Camino desde 0 hasta 0: [0, 2, 5, 3, 7, 4, 6, 1, 0];  
Kms: 9,000000;  
Beneficio: 1463,000000;  
Path de la solucion = [2, 5, 3, 7, 4, 6, 1, 0]

Solucion PDR:  
Camino desde 0 hasta 0: [0, 2, 5, 3, 7, 4, 6, 1, 0];  
Kms: 9,000000;  
Beneficio: 1463,000000;  
Path de la solucion = [2, 5, 3, 7, 4, 6, 1, 0]

Solucion BT:  
Camino desde 0 hasta 0: [0, 2, 5, 3, 7, 4, 6, 1, 0];  
Kms: 9,000000;  
Beneficio: 1463,000000;  
Path de la solucion = [2, 5, 3, 7, 4, 6, 1, 0]

## 4.5. Ejercicio 4 – Manual (BT)

### 4.5.1. RepartidorProblem

```
package ejercicio4.manual;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.stream.IntStream;

import _datos.DatosRepartidor;
import us.lsi.common.List2;
import us.lsi.common.Set2;

public record RepartidorProblem(Integer cliente, Set<Integer> pendientes, List<Integer> visitados,
    Double kms) {

    // Metodo de factoria
    public static RepartidorProblem of(Integer cliente, Set<Integer> pendientes, List<Integer> visitados,
        Double kms) {
        return new RepartidorProblem(cliente, pendientes, visitados, kms);
    }

    // Vertice inicial: el cliente es el 0, los pendientes son todos los vertices que hay en el grafo,
    // los visitados es una lista vacia y los km recorridos son 0
    public static RepartidorProblem initial() {
        List<Integer> visitados = new ArrayList<>();

        Set<Integer> pendientes = new HashSet<>();
        for(int i=0; i<DatosRepartidor.getNumVertices(); i++) {
            pendientes.add(i);
        }

        return of(0, pendientes, visitados, 0.);
    }

    // Alternativas a las que podemos ir, es decir, a los vertices (del grafo) a los que podemos ir desde
    // el ultimo que hemos visitado
    public List<Integer> actions() {
        List<Integer> alternativas = new ArrayList<>();
        List<Integer> pendientesVisitar = pendientes().stream().toList();

        // si ya hemos recorrido todos los vertices no habra alternativas
        if (cliente() < DatosRepartidor.getNumVertices()) {
            // si estamos en el cliente 0, tenemos que ver aquellos que esten conectados al vertice
            // origen, es decir, el 0, quitando obviamente el 0
            if (cliente() == 0) {
                for (Integer vertice : pendientesVisitar) {
                    if (vertice != 0 && DatosRepartidor.existeArista(0, vertice)) {
                        alternativas.add(vertice);
                    }
                }
            } // si estamos en el ultimo cliente, tenemos que ver que el ultimo visitado pueda conectarse
            // con el destino que es el 0
            else if (cliente() == DatosRepartidor.getNumVertices() - 1) {
                if (pendientesVisitar.contains(0)
                    && DatosRepartidor.existeArista(visitados().get(visitados().size() - 1), 0)) {
                    alternativas.add(0);
                }
            } // si estamos en el cualquier otro vertice, tenemos que mirar aquellos que queden que puedan
            // estar conectados con el ultimo vertice visitado quitando el 0 que sera el destino final
            else {
                for (Integer vertice : pendientesVisitar) {
                    if (vertice != 0
                        && DatosRepartidor.existeArista(visitados().get(visitados().size() - 1), vertice)) {
                        alternativas.add(vertice);
                    }
                }
            }
        }
        return alternativas;
    }
}
```

```

// Vecino al que vamos, es decir, incrementamos el indice, quitamos la accion (vertice al que vamos)
// del conjunto de pendientes, añadimos la accion al conjunto de vertices visitados y aumentamos los
// kms con el peso de la arista entre vertices
public RepartidorProblem neighbor(Integer a) {
    // eliminamos de los pendientes
    Set<Integer> pendientesVecino = Set2.copy(pendientes());
    pendientesVecino.remove(a);

    // añadimos a los visitados
    List<Integer> visitadosVecino = List2.copy(visitados());
    visitadosVecino.add(a);

    // añadimos kms recorridos
    Double kmsVecino = kms();
    if (cliente() == 0) {
        kmsVecino += DatosRepartidor.getPesoArista(0, a);
    } else {
        kmsVecino += DatosRepartidor.getPesoArista(visitados().get(visitados().size() - 1), a);
    }

    return of(cliente() + 1, pendientesVecino, visitadosVecino, kmsVecino);
}

// Heuristica: lo mejor que puede pasar a partir de un vertice es que se consigan todos los beneficios
// de los clientes (vertices) que quedan por ir y restandole los kms suponiendo que la distancia minima
// es 1
public Double heuristic() {
    Double beneficio = 0.;

    List<Integer> pendientes = pendientes().stream().toList(); // para poder recorrerlo
    for(Integer vertice: pendientes) {
        beneficio += DatosRepartidor.getBeneficioCliente(vertice);
    }
    Double kms = IntStream.range(1, pendientes().size()).mapToDouble(x -> kms() + x).sum();

    return beneficio - kms - 1;
}

```

#### 4.5.2. RepartidorState

```

package ejercicio4.manual;

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

import _datos.DatosRepartidor;
import _soluciones.SolucionRepartidor;

public class RepartidorState {

    // Propiedades del estado
    RepartidorProblem actual;
    Double acumulado;
    List<Integer> acciones;
    List<RepartidorProblem> anteriores;

    // Constructor
    private RepartidorState(RepartidorProblem prob, Double ac, List<Integer> ls1,
                           List<RepartidorProblem> ls2) {
        actual = prob;
        acumulado = ac;
        acciones = ls1;
        anteriores = ls2;
    }

    // Metodo de factoria
    public static RepartidorState of(RepartidorProblem prob, Double ac, List<Integer> ls1,
                                     List<RepartidorProblem> ls2) {
        return new RepartidorState(prob, ac, ls1, ls2);
    }
}

```

```

// Estado inicial: lo obtenemos a partir del problema inicial
public static RepartidorState initial() {
    RepartidorProblem probInicial = RepartidorProblem.initial();
    List<Integer> acciones = new ArrayList<>();
    List<RepartidorProblem> anteriores = new ArrayList<>();
    return of(probInicial, 0., acciones, anteriores);
}

// forward: como cambia el estado al avanzar, es decir, al acumulado sumarle el beneficio de ir
// a ese vertice menos la distancia recorrida acumulada, añadir a la lista de acciones la accion
// que tomamos, añadir a anteriores el problema actual y el actual pasa a ser el problema vecino
public void forward(Integer a) {
    Double beneficio = DatosRepartidor.getBeneficioCliente(a);
    Double distanciaKms = actual.kms();
    if(actual.cliente() == 0) {
        distanciaKms += DatosRepartidor.getPesoArista(0, a);
    } else {
        distanciaKms += DatosRepartidor.getPesoArista(acciones.size()-1, a);
    }

    acumulado += beneficio - distanciaKms;
    acciones.add(a);
    anteriores.add(actual);
    actual = actual.neighbor(a);
}

// back: como cambia el estado al volver atras, es decir, al acumulado hay que restarle el beneficio
// de haber llegado al ultimo menos los kms que ha recorrido, quitar la ultima accion de la lista de
// acciones, quitar de la lista de anteriores el ultimo visto y el actual pasa a ser este ultimo
// que hemos quitado
public void back() {
    int last = acciones.size()-1;
    RepartidorProblem prob_ant = anteriores.get(last);

    Double beneficio = DatosRepartidor.getBeneficioCliente(acciones.get(last));
    Double distanciaKms = actual.kms();

    acumulado -= (beneficio - distanciaKms);
    acciones.remove(last);
    anteriores.remove(last);
    actual = prob_ant;
}

// Alternativas: alternativas del estado inicial, es decir, las alternativas del vertice actual
public List<Integer> alternativas() {
    return actual.actions();
}

// Cota: sirve para saber que es lo mejor que puede pasar desde un vertice tomando la alternativa a
public Double cota(Integer a) {
    Double weight = DatosRepartidor.getBeneficioCliente(a) - actual.kms();
    if(actual.cliente() == 0) {
        weight -= DatosRepartidor.getPesoArista(0, a);
    } else {
        weight -= DatosRepartidor.getPesoArista(acciones.size()-1, a);
    }

    return acumulado + weight + actual.neighbor(a).heuristic();
}

// esSolucion: que no queden vertices por visitar, es decir, que los pendientes sea un conjunto
// vacio
public Boolean esSolucion() {
    return actual.pendientes().size() == 0;
}

// esTerminal: recorrer todos los vertices del grafo, es decir, que el cliente sea el numero de
// vertices que hay en el grafo
public Boolean esTerminal() {
    return actual.cliente() == DatosRepartidor.getNumVertices();
}

// getSolucion: devuelve la solucion del estado en el que estamos
public SolucionRepartidor getSolucion() {
    return SolucionRepartidor.of(acciones);
}
}

```

#### 4.5.3. RepartidorBT

```

package ejercicio4.manual;

import java.util.HashSet;

public class RepartidorBT {

    // Propiedades
    private static Double mejorValor;
    private static RepartidorState estado;
    private static Set<SolucionRepartidor> soluciones;

    // Funcion que da los valores iniciales y llama al algoritmo en si
    public static void search() {
        soluciones = new HashSet<>();
        mejorValor = 0.; // estamos maximizando
        estado = RepartidorState.initial();
        bt_search();
    }

    // Funcion que es basicamente el algoritmo
    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();
                }
            }
        }
    }

    // Funcion que devuelve las soluciones del problema
    public static Set<SolucionRepartidor> getSoluciones() {
        return soluciones;
    }
}

```

#### 4.5.4. TestsEjercicio4BT

```

package ejercicios.tests.manual;

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

import _datos.DatosRepartidor;
import _soluciones.SolucionRepartidor;
import _utils.TestsPIIS;
import ejercicio4.manual.RepartidorBT;

public class TestsEjercicio4BT {

    public static void main(String[] args) {
        System.out.println("* TESTS EJERCICIO 4 - BT MANUAL *\n");
        TestsPIIS.line("*");

        List.of(1,2).forEach(num_test -> {
            DatosRepartidor.iniDatos("ficheros/Ejercicio4DatosEntrada"+num_test+".txt");
            RepartidorBT.search();
            SolucionRepartidor sol = RepartidorBT.getSoluciones().stream().max(Comparator.comparing(SolucionRepartidor::getBeneficio)).orElse(null);
            String res = "\n- Solucion obtenida: " + sol + "\n";
            System.out.println(res);
            TestsPIIS.line("*");
        });
    }
}

```

#### **4.5.5. Volcado de pantalla**

```
* TESTS EJERCICIO 4 - BT MANUAL *

*****
- Numero de clientes leidos: 5
- Clientes leidos:
  Cliente[clienteId=0, beneficio=0.0]
  Cliente[clienteId=1, beneficio=400.0]
  Cliente[clienteId=2, beneficio=300.0]
  Cliente[clienteId=3, beneficio=200.0]
  Cliente[clienteId=4, beneficio=100.0]

- Numero de conexiones leidas: 8
- Conexiones leidas:
  Conexion[conexionId=0, distanciaKm=1.0]
  Conexion[conexionId=1, distanciaKm=100.0]
  Conexion[conexionId=2, distanciaKm=1.0]
  Conexion[conexionId=3, distanciaKm=100.0]
  Conexion[conexionId=4, distanciaKm=1.0]
  Conexion[conexionId=5, distanciaKm=1.0]
  Conexion[conexionId=6, distanciaKm=100.0]
  Conexion[conexionId=7, distanciaKm=5.0]

- Solucion obtenida:
Camino desde 0 hasta 0: [0, 1, 2, 3, 4, 0];
Kms: 9,000000;
Beneficio: 981,000000;

*****
- Numero de clientes leidos: 8
- Clientes leidos:
  Cliente[clienteId=0, beneficio=0.0]
  Cliente[clienteId=1, beneficio=100.0]
  Cliente[clienteId=2, beneficio=200.0]
  Cliente[clienteId=3, beneficio=300.0]
  Cliente[clienteId=4, beneficio=200.0]
  Cliente[clienteId=5, beneficio=300.0]
  Cliente[clienteId=6, beneficio=200.0]
  Cliente[clienteId=7, beneficio=200.0]

- Numero de conexiones leidas: 13
- Conexiones leidas:
  Conexion[conexionId=8, distanciaKm=2.0]
  Conexion[conexionId=9, distanciaKm=1.0]
  Conexion[conexionId=10, distanciaKm=1.0]
  Conexion[conexionId=11, distanciaKm=3.0]
  Conexion[conexionId=12, distanciaKm=1.0]
  Conexion[conexionId=13, distanciaKm=1.0]
  Conexion[conexionId=14, distanciaKm=3.0]
  Conexion[conexionId=15, distanciaKm=1.0]
  Conexion[conexionId=16, distanciaKm=1.0]
  Conexion[conexionId=17, distanciaKm=3.0]
  Conexion[conexionId=18, distanciaKm=1.0]
  Conexion[conexionId=19, distanciaKm=1.0]
  Conexion[conexionId=20, distanciaKm=1.0]

- Solucion obtenida:
Camino desde 0 hasta 0: [0, 2, 5, 3, 7, 4, 6, 1, 0];
Kms: 9,000000;
Beneficio: 1463,000000;

*****
```

## 5. UTILS

### 5.1. GraphsPI5

```
package _utils;

import java.util.function.Predicate;

import ejercicio1.CafeEdge;
import ejercicio1.CafeHeuristic;
import ejercicio1.CafeVertex;
import ejercicio2.CursosEdge;
import ejercicio2.CursosHeuristic;
import ejercicio2.CursosVertex;
import ejercicio3.InvestigadoresEdge;
import ejercicio3.InvestigadoresHeuristic;
import ejercicio3.InvestigadoresVertex;
import ejercicio4.RepartidorEdge;
import ejercicio4.RepartidorHeuristic;
import ejercicio4.RepartidorVertex;
import us.lsi.graphs.virtual.EGraph;
import us.lsi.graphs.virtual.EGraph.Type;
import us.lsi.graphs.virtual.EGraphBuilder;
import us.lsi.path.EGraphPath.PathType;

// Clase factoria para crear los constructores de los grafos de los ejercicios
public class GraphsPI5 {

    // Ejercicio1: Builder
    public static EGraphBuilder<CafeVertex, CafeEdge> cafeBuilder(CafeVertex v_inicial,
        Predicate<CafeVertex> es_terminal) {
        return EGraph.virtual(v_inicial, es_terminal, PathType.Sum, Type.Max)
            .goalHasSolution(CafeVertex.goalHasSolution())
            .heuristic(CafeHeuristic::heuristic);
    }

    // Ejercicio2: Builder
    public static EGraphBuilder<CursosVertex, CursosEdge> cursosBuilder(CursosVertex v_inicial,
        Predicate<CursosVertex> es_terminal) {
        return EGraph.virtual(v_inicial, es_terminal, PathType.Sum, Type.Min)
            .goalHasSolution(CursosVertex.goalHasSolution())
            .heuristic(CursosHeuristic::heuristic);
    }

    // Ejercicio3: Builder
    public static EGraphBuilder<InvestigadoresVertex, InvestigadoresEdge> investigadoresBuilder(
        InvestigadoresVertex v_inicial, Predicate<InvestigadoresVertex> es_terminal) {
        return EGraph.virtual(v_inicial, es_terminal, PathType.Sum, Type.Max)
            .goalHasSolution(InvestigadoresVertex.goalHasSolution())
            .heuristic(InvestigadoresHeuristic::heuristic);
    }

    // Ejercicio4: Builder
    public static EGraphBuilder<RepartidorVertex, RepartidorEdge> repartidorBuilder(
        RepartidorVertex v_inicial, Predicate<RepartidorVertex> es_terminal) {
        return EGraph.virtual(v_inicial, es_terminal, PathType.Sum, Type.Max)
            .goalHasSolution(RepartidorVertex.goalHasSolution())
            .heuristic(RepartidorHeuristic::heuristic);
    }
}
```

## 5.2. TestsPI5

```
package _utils;

import java.util.function.BiFunction;
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.virtual.EGraph;
import us.lsi.graphs.virtual.EGraphBuilder;

// Clase que reune y simplifica todos los tests
public class TestsPI5 {

    public static<V,E> GraphPath<V, E> testAStar(EGraph<V, E> graph) {
        var alg_star = AStar.of(graph);
        var res = alg_star.search().orElse(null);
        var outGraph = alg_star.outGraph();
        if(outGraph != null) {
            toDot("AStar", outGraph, res);
        }
        return res;
    }

    public static<V,E> GraphPath<V, E> testPDR(EGraph<V, E> graph) {
        var alg_pdr = DPR.of(graph, null, null, null, true);
        var res = alg_pdr.search().orElse(null);
        var outGraph = alg_pdr.outGraph();
        if(outGraph != null) {
            toDot("PDR", outGraph, res);
        }
        return res;
    }

    public static<V,E,S> GraphPath<V, E> testBT(EGraph<V, E> graph) {
        var alg_bt = BT.of(graph, null, null, null, true);
        var res = alg_bt.search().orElse(null);
        var outGraph = alg_bt.outGraph();
        if(outGraph != null) {
            toDot("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);
        }
        line("_");
    }

    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)));
}

public static<V,E> void tests(V vInicial, Predicate<V> goal,
BiFunction<V, Predicate<V>, EGraphBuilder<V,E>> builder, Function<GraphPath<V, E>, ?> solutionOf) {
    var pathA = testAStar(builder.apply(vInicial, goal).build());
    toConsole("A*", pathA, solutionOf);

    var pathPDR = testPDR(builder.apply(vInicial, goal).build());
    toConsole("PDR", pathPDR, solutionOf);

    var pathBT = testBT(builder.apply(vInicial, goal).build());
    toConsole("BT", pathBT, solutionOf);

    line("*");
}

```