Algoritmos de Búsqueda Tabú
Aplicación en un problema de rutas

Anexo

Abel Naya Forcano

Trabajo de fin de grado en Matemáticas
Universidad de Zaragoza

Directores del trabajo:
Herminia I. Calvete
Ángel R. Francés

Julio de 2016

Una implementación del algoritmo Taburoute

Este documento es un anexo al trabajo fin de grado 'Algoritmos de Búsqueda Tabú. Aplicación en un problema de rutas'. Contiene el código fuente, en el lenguaje de programación Java, de una implementación propia del algoritmo presentado en el articulo de Gendreau et al. (A tabu search heuristic for the vehicle routing problem. Management Science, 40(10):1276–1290, 1994). Las 23 clases que componen el programa son las siguientes:

1. EstructuraTabu.java 1
2. Genius.java 3
3. Grafo.java 4
4. RP_insert.java 8
5. RP_insert_1o2.java 9
6. RP_insert_I.java 10
7. RP_insert_II.java 11
8. RP_insert_IIr.java 12
9. RP_insert_Ir.java 13
10. RP_remove.java 14
11. RP_remove_0.java 15
12. RP_remove_1.java 16
13. RP_remove_I.java 17
14. RP_remove_II.java 18
15. RP_remove_IIr.java 19
16. RP_remove_Ir.java 20
17. Ruta.java 21
18. RutaAbstracta.java 38
19. RutaProxi.java 42
20. Solucion.java 45
21. SolucionAbstracta.java 50
22. SolucionProxi.java 51
23. TabuRoute.java 55

Clase 1: EstructuraTabu.java

```
1  import java.util.HashMap;
2  import java.util.Map;
3
4  /**
5   * Especifica cuando la inserción de un nodo en una ruta en una
        iteración es Tabu
6   */
7  public class EstructuraTabu {
8
9      /**
10      * Representa un par nodo-ruta
11      */
12     private static class _Par {
13
14         private final int nodo, ruta;
15
16         @Override
17         public int hashCode() {
18             int hash = 7;
19             hash = 23 * hash + this.nodo;
20             hash = 23 * hash + this.ruta;
21             return hash;
22         }
23
24         @Override
25         public boolean equals(Object obj) {
26             if (obj == null) {
27                 return false;
28             }
29             if (getClass() != obj.getClass()) {
30                 return false;
31             }
32             final _Par other = (_Par) obj;
33             if (this.nodo != other.nodo) {
34                 return false;
35             }
36             if (this.ruta != other.ruta) {
37                 return false;
38             }
39             return true;
40         }
41
42         public _Par(int nodo, int ruta) {
43             this.nodo = nodo;
44             this.ruta = ruta;
45         }
46
47     }
48
49     /**
50      * La estructura tabú
51      */
52     private final Map<_Par, Integer> tabu;
53
54     /**
55      * Inicializa una estructura tabu donde ningún movimiento es tabú
56      */
```

```
57      public EstructuraTabu() {
58          tabu = new HashMap<>();
59      }
60
61      /**
62       * Comprueba si insertar el nodo en la ruta en una iteración es
             tabu o no
63       */
64      boolean isTabu(int nodo, int ruta, int iteracion) {
65          Integer value = tabu.get(new _Par(nodo, ruta));
66
67          if (value == null) {
68              return false;
69          }
70
71          return iteracion <= value;
72      }
73
74      /**
75       * Marca como tabu la inserción del nodo en la ruta hasta la
             iteración dada
76       */
77      void setTabu(int nodo, int ruta, int iteracion) {
78          tabu.put(new _Par(nodo, ruta), iteracion);
79      }
80
81  }
```

Clase 2: Genius.java

```
1  /**
2   * Representa el algoritmo GENIUS
3   */
4  public class Genius {
5
6      /**
7       * Genera un nuevo array con los nodos reorganizados insertando los
8       * elementos de la secuencia mediante GENI y realizando US
            posteriormente
9       */
10     public static int[] ejecutar(int[] sequence) {
11         int p = 5;
12         Ruta res = new Ruta();
13         for (int v : sequence) {
14             res.stringing(v, p)
15                     .modificarRuta();
16         }
17
18         res.us(p);
19
20         return res.toArray();
21     }
22
23 }
```

Clase 3: Grafo.java

```
1  import java.io.BufferedReader;
2  import java.util.Arrays;
3  import java.util.Comparator;
4  import java.util.Scanner;
5
6  /**
7   * Almacena los parametros del problema
8   */
9  public class Grafo {
10
11     /**
12      * Un valor considerado infinito
13      */
14     private static double infinity;
15
16     /**
17      * Numero de nodos
18      */
19     private static int n;
20
21     /**
22      * numero maximo de rutas
23      */
24     private static int mbar;
25
26     /**
27      * Matriz de adyacencia
28      */
29     private static double[][] cij;
30
31     /**
32      * Distancia maxima de las rutas
33      */
34     private static double l;
35
36     /**
37      * Capacidad maxima de las rutas
38      */
39     private static double q;
40
41     /**
42      * Capacidad de cada nodo
43      */
44     private static double[] qi;
45
46     /**
47      * Tiempo de servicio de cada nodo
48      */
49     private static double[] di;
50
51     /**
52      * Una lista de arrays de los vecinos mas cercanos
53      */
54     private static int[][] nearest;
55
56     /**
57      * Lee los parametros de un fichero
```

```
58       */
59      public static void inicializar(BufferedReader file) {
60          Scanner sc = new Scanner(file);
61          n = sc.nextInt();
62          mbar = n;
63          q = sc.nextInt();
64          l = sc.nextInt();
65          int dropTime = sc.nextInt();
66          nearest = new int[n + 1][n];
67          qi = new double[n + 1];
68          di = new double[n + 1];
69          cij = new double[n + 1][n + 1];
70          Integer[] nodos = new Integer[n + 1];
71          int x[] = new int[n + 1];
72          int y[] = new int[n + 1];
73          x[0] = sc.nextInt();
74          y[0] = sc.nextInt();
75          qi[0] = 0;
76          di[0] = 0;
77          nodos[0] = 0;
78          infinity = 0;
79          for (int c = 1; c <= n; ++c) {
80              x[c] = sc.nextInt();
81              y[c] = sc.nextInt();
82              qi[c] = sc.nextInt();
83              di[c] = dropTime;
84              cij[c][c] = 0;
85              nodos[c] = c;
86              for (int d = 0; d < c; ++d) {
87                  double dist = Math.sqrt(Math.pow(x[d] - x[c], 2) +
                        Math.pow(y[d] - y[c], 2));
88                  dist = Math.round(dist * 10000d) / 10000d;
89                  cij[c][d] = dist;
90                  cij[d][c] = dist;
91                  infinity += dist;
92              }
93          }
94          ComparadorNoDirigido comparador = new ComparadorNoDirigido();
95          for (int c = 1; c <= n; ++c) {
96              comparador.setOrigen(c);
97              Arrays.sort(nodos, comparador);
98              int pos = 0;
99              for (int d : nodos) {
100                 if (d == c) {
101                     continue;
102                 }
103                 nearest[c][pos] = d;
104                 pos++;
105             }
106         }
107     }
108
109     public static double getInfinity() {
110         return infinity;
111     }
112
113     public static int getN() {
114         return n;
```

```
115     }
116
117     public static int getMbar() {
118         return mbar;
119     }
120
121     public static double getCij(int from, int to) {
122         return cij[from][to];
123     }
124
125     public static double getQi(int id) {
126         return qi[id];
127     }
128
129     public static double getDi(int id) {
130         return di[id];
131     }
132
133     public static double getQ() {
134         return q;
135     }
136
137     public static double getL() {
138         return l;
139     }
140
141     public static int[] getNearestNodos(int v) {
142         return nearest[v];
143     }
144
145     /**
146      * Muestra la información por pantalla
147      */
148     public static void showData() {
149         System.out.println("***************************");
150         System.out.println("********** Data ***********");
151         System.out.println("***************************");
152         System.out.println("INFINITY: " + infinity);
153         System.out.println("n: " + n);
154         System.out.println("mbar: " + mbar);
155         System.out.println("L: " + l);
156         System.out.println("Q: " + q);
157         System.out.println();
158         System.out.println("Qi:");
159         System.out.println(Arrays.toString(qi));
160         System.out.println("Di:");
161         System.out.println(Arrays.toString(di));
162         System.out.println();
163         System.out.println("Cij:");
164         for (double[] row : cij) {
165             for (double column : row) {
166                 System.out.print(String.format("%6.2f ", column));
167             }
168             System.out.print("\n");
169         }
170         System.out.println();
171         System.out.println("nearest:");
172         for (int i = 1; i < nearest.length; ++i) {
```

```
173             System.out.println(i + ": " + Arrays.toString(nearest[i]));
174         }
175     }
176 
177     /**
178      * Un comparador que ordena nodos basados en la menor distancia:
             min{cij, cji}
179      */
180     private static class ComparadorNoDirigido implements
           Comparator<Integer> {
181 
182         private int origen;
183 
184         public void setOrigen(int origen) {
185             this.origen = origen;
186         }
187 
188         @Override
189         public int compare(Integer o1, Integer o2) {
190             double dif = Math.min(getCij(origen, o1), getCij(o1,
                   origen)) - Math.min(getCij(origen, o2), getCij(o2,
                   origen));
191             return dif > 0 ? 1 : dif < 0 ? -1 : 0;
192         }
193     }
194 }
```

Clase 4: RP_insert.java

```
1  /**
2   * Una modificacion en la que se inserta el nodo en la ruta
3   */
4  public abstract class RP_insert extends RutaProxi {
5
6      public RP_insert(Ruta original, Nodo v, double deltaCost) {
7          super(original, v, deltaCost);
8      }
9
10     @Override
11     public double getSumQi() {
12         super.getSumQi();
13
14         return getRuta().getSumQi() + Grafo.getQi(getV().getId());
15     }
16
17     @Override
18     public double getSumDi() {
19         super.getSumDi();
20
21         return getRuta().getSumDi() + Grafo.getDi(getV().getId());
22     }
23
24 }
```

Clase 5: RP_insert_1o2.java

```java
/**
 * La modificación de una ruta que pasará a tener los nodos dados en orden
 */
public class RP_insert_1o2 extends RP_insert {

    private final Nodo[] orden;

    public RP_insert_1o2(Nodo[] orden, Ruta original, Nodo v, double deltaCost) {
        super(original, v, deltaCost);
        this.orden = orden;
    }

    @Override
    public void internal_modificar() {

        Nodo o = getRuta().getOrigen();

        join(o, orden[0]);

        for (int i = 0; i < orden.length - 1; i++) {
            join(orden[i], orden[i + 1]);
        }

        join(orden[orden.length - 1], o);

        getRuta().actualizarInsertado(getV());
    }

}
```

Clase 6: RP_insert_I.java

```
/**
 * La modificacion de una ruta insertando el nodo mediante GENI tipo I
    en el sentido original
 */
public class RP_insert_I extends RP_insert {

    private final Nodo vi;
    private final Nodo vj;
    private final Nodo vk;

    public RP_insert_I(Ruta original, Nodo v, double deltaCost, Nodo
       vi, Nodo vj, Nodo vk) {
        super(original, v, deltaCost);
        this.vi = vi;
        this.vj = vj;
        this.vk = vk;

    }

    @Override
    public void internal_modificar() {
        Ruta ruta = getRuta();

        Nodo v = getV();
        Nodo viNext = vi.getNext();
        Nodo vjNext = vj.getNext();
        Nodo vkNext = vk.getNext();

        reverseNodos(viNext, vj);
        reverseNodos(vjNext, vk);

        join(vi, v);

        join(v, vj);

        join(viNext, vk);

        join(vjNext, vkNext);

        ruta.actualizarInsertado(getV());

    }

}
```

Clase 7: RP_insert_II.java

```
/**
 * La modificacion de una ruta insertando el nodo mediante GENI tipo
    II en el sentido original
 */
public class RP_insert_II extends RP_insert {

    private final Nodo vi;
    private final Nodo vj;
    private final Nodo vk;
    private final Nodo vl;

    public RP_insert_II(Ruta original, Nodo v, double deltaCost, Nodo
       vi, Nodo vj, Nodo vk, Nodo vl) {
        super(original, v, deltaCost);
        this.vi = vi;
        this.vj = vj;
        this.vk = vk;
        this.vl = vl;
    }

    @Override
    public void internal_modificar() {
        Ruta ruta = getRuta();

        Nodo v = getV();
        Nodo viNext = vi.getNext();
        Nodo vjNext = vj.getNext();
        Nodo vkPrev = vk.getPrev();
        Nodo vlPrev = vl.getPrev();

        reverseNodos(viNext, vlPrev);
        reverseNodos(vl, vj);

        join(vi, v);

        join(v, vj);

        join(vl, vjNext);

        join(vkPrev, vlPrev);

        join(viNext, vk);

        ruta.actualizarInsertado(getV());

    }
}
```

Clase 8: RP_insert_IIr.java

```
1  /**
2   * La modificacion de una ruta insertando el nodo mediante GENI tipo
       II en el sentido contrario
3   */
4  public class RP_insert_IIr extends RP_insert {
5
6      private final Nodo vi;
7      private final Nodo vj;
8      private final Nodo vk;
9      private final Nodo vl;
10
11     public RP_insert_IIr(Ruta original, Nodo v, double deltaCost, Nodo
           vi, Nodo vj, Nodo vk, Nodo vl) {
12         super(original, v, deltaCost);
13         this.vi = vi;
14         this.vj = vj;
15         this.vk = vk;
16         this.vl = vl;
17     }
18
19     @Override
20     public void internal_modificar() {
21         Ruta ruta = getRuta();
22
23         Nodo v = getV();
24         Nodo viNext = vi.getPrev();
25         Nodo vjNext = vj.getPrev();
26         Nodo vkPrev = vk.getNext();
27         Nodo vlPrev = vl.getNext();
28
29         reverseNodos(vi, vk);
30         reverseNodos(vkPrev, vjNext);
31
32         join(vi, v);
33
34         join(v, vj);
35
36         join(vl, vjNext);
37
38         join(vkPrev, vlPrev);
39
40         join(viNext, vk);
41
42         ruta.actualizarInsertado(getV());
43
44     }
45 }
```

Clase 9: RP_insert_Ir.java

```
1  /**
2   * La modificacion de una ruta insertando el nodo mediante GENI tipo I
        en el sentido contrario
3   */
4  public class RP_insert_Ir extends RP_insert {
5
6      private final Nodo vi;
7      private final Nodo vj;
8      private final Nodo vk;
9
10     public RP_insert_Ir(Ruta original, Nodo v, double deltaCost, Nodo
           vi, Nodo vj, Nodo vk) {
11         super(original, v, deltaCost);
12         this.vi = vi;
13         this.vj = vj;
14         this.vk = vk;
15
16     }
17
18     @Override
19     public void internal_modificar() {
20         Ruta ruta = getRuta();
21
22         Nodo v = getV();
23         Nodo viNext = vi.getPrev();
24         Nodo vjNext = vj.getPrev();
25         Nodo vkNext = vk.getPrev();
26
27         reverseNodos(vi, vkNext);
28
29         join(vi, v);
30
31         join(v, vj);
32
33         join(viNext, vk);
34
35         join(vjNext, vkNext);
36
37         ruta.actualizarInsertado(getV());
38
39     }
40 }
```

Clase 10: RP_remove.java

```
1  /**
2   * Una modificación en la que se elimina el nodo de la ruta
3   */
4  public abstract class RP_remove extends RutaProxi {
5
6      public RP_remove(Ruta original, Nodo v, double deltaCost) {
7          super(original, v, deltaCost);
8      }
9
10     @Override
11     public double getSumQi() {
12         super.getSumQi();
13
14         return getRuta().getSumQi() - Grafo.getQi(getV().getId());
15     }
16
17     @Override
18     public double getSumDi() {
19         super.getSumDi();
20
21         return getRuta().getSumDi() - Grafo.getDi(getV().getId());
22     }
23 }
```

Clase 11: RP_remove_0.java

```
/**
 * La modificacion de una ruta que pasará a tener únicamente el origen
 */
public class RP_remove_0 extends RP_remove {

    public RP_remove_0(Ruta original, Nodo v, double deltaCost) {
        super(original, v, deltaCost);
    }

    @Override
    public void internal_modificar() {
        Nodo origen = getRuta().getOrigen();

        join(origen, origen);

        getRuta().actualizarRemovido(getV());
    }

}
```

Clase 12: RP_remove_1.java

```
1  /**
2   * La modificación de una ruta que pasará a tener un solo vértice
3   */
4  class RP_remove_1 extends RP_remove {
5
6      private final Nodo a;
7
8      public RP_remove_1(Nodo a, Ruta route, Nodo v, double deltaCost) {
9          super(route, v, deltaCost);
10         this.a = a;
11     }
12
13     @Override
14     public void internal_modificar() {
15         Ruta ruta = getRuta();
16
17         Nodo origen = ruta.getOrigen();
18
19         join(origen, a);
20         join(a, origen);
21
22         ruta.actualizarRemovido(getV());
23     }
24
25 }
```

Clase 13: RP_remove_I.java

```java
/**
 * La modificacion de una ruta eliminando el nodo mediante Unstringing
    tipo I en el sentido original
 */
public class RP_remove_I extends RP_remove {

    private final Nodo vj;
    private final Nodo vk;

    public RP_remove_I(Ruta original, Nodo v, double deltaCost, Nodo
       vj, Nodo vk) {
        super(original, v, deltaCost);
        this.vj = vj;
        this.vk = vk;
    }

    @Override
    public void internal_modificar() {
        Ruta ruta = getRuta();

        Nodo vi = getV();
        Nodo viNext = vi.getNext();
        Nodo viPrev = vi.getPrev();
        Nodo vjNext = vj.getNext();
        Nodo vkNext = vk.getNext();

        reverseNodos(viNext, vk);
        reverseNodos(vkNext, vj);

        join(viPrev, vk);

        join(viNext, vj);

        join(vkNext, vjNext);

        ruta.actualizarRemovido(getV());

    }

}
```

Clase 14: RP_remove_II.java

```
1  /**
2   * La modificacion de una ruta eliminando el nodo mediante Unstringing
       tipo II en el sentido original
3   */
4  public class RP_remove_II extends RP_remove {
5
6      private final Nodo vj;
7      private final Nodo vk;
8      private final Nodo vl;
9
10     public RP_remove_II(Ruta original, Nodo v, double deltaCost, Nodo
           vj, Nodo vk, Nodo vl) {
11         super(original, v, deltaCost);
12         this.vj = vj;
13         this.vk = vk;
14         this.vl = vl;
15     }
16
17     @Override
18     public void internal_modificar() {
19         Ruta ruta = getRuta();
20
21         Nodo vi = getV();
22         Nodo viNext = vi.getNext();
23         Nodo viPrev = vi.getPrev();
24         Nodo vjPrev = vj.getPrev();
25         Nodo vkNext = vk.getNext();
26         Nodo vlNext = vl.getNext();
27
28         reverseNodos(viNext, vjPrev);
29         reverseNodos(vlNext, vk);
30
31         join(viPrev, vk);
32
33         join(vlNext, vjPrev);
34
35         join(viNext, vj);
36
37         join(vl, vkNext);
38
39         ruta.actualizarRemovido(getV());
40
41     }
42 }
```

Clase 15: RP_remove_IIr.java

```
1  /**
2   * La modificacion de una ruta eliminando el nodo mediante Unstringing
        tipo II en el sentido contrario
3   */
4  public class RP_remove_IIr extends RP_remove {
5
6      private final Nodo vj;
7      private final Nodo vk;
8      private final Nodo vl;
9
10     public RP_remove_IIr(Ruta original, Nodo v, double deltaCost, Nodo
           vj, Nodo vk, Nodo vl) {
11         super(original, v, deltaCost);
12         this.vj = vj;
13         this.vk = vk;
14         this.vl = vl;
15     }
16
17     @Override
18     public void internal_modificar() {
19         Ruta ruta = getRuta();
20
21         Nodo vi = getV();
22         Nodo viNext = vi.getPrev();
23         Nodo viPrev = vi.getNext();
24         Nodo vjPrev = vj.getNext();
25         Nodo vkNext = vk.getPrev();
26         Nodo vlNext = vl.getPrev();
27
28         reverseNodos(vl, vj);
29         reverseNodos(viPrev, vkNext);
30
31         join(viPrev, vk);
32
33         join(vlNext, vjPrev);
34
35         join(viNext, vj);
36
37         join(vl, vkNext);
38
39         ruta.actualizarRemovido(getV());
40
41     }
42
43 }
```

Clase 16: RP_remove_Ir.java

```
1  /**
2   * La modificacion de una ruta eliminando el nodo mediante Unstringing
       tipo I en el sentido contrario
3   */
4  public class RP_remove_Ir extends RP_remove {
5
6      private final Nodo vj;
7      private final Nodo vk;
8
9      public RP_remove_Ir(Ruta original, Nodo v, double deltaCost, Nodo
           vj, Nodo vk) {
10         super(original, v, deltaCost);
11         this.vj = vj;
12         this.vk = vk;
13     }
14
15     @Override
16     public void internal_modificar() {
17         Ruta ruta = getRuta();
18
19         Nodo vi = getV();
20         Nodo viNext = vi.getPrev();
21         Nodo viPrev = vi.getNext();
22         Nodo vjNext = vj.getPrev();
23         Nodo vkNext = vk.getPrev();
24
25         reverseNodos(viPrev, vjNext);
26
27         join(viPrev, vk);
28
29         join(viNext, vj);
30
31         join(vkNext, vjNext);
32
33         ruta.actualizarRemovido(getV());
34
35     }
36 }
```

Clase 17: Ruta.java

```
1  import java.util.Iterator;
2  import java.util.TreeSet;
3
4  /**
5   * Representa una ruta real
6   */
7  public class Ruta extends RutaAbstracta {
8
9      /**
10      * El proximo id para la proxima ruta creada
11      */
12     static private int id_ruta_Next = 1;
13
14     /**
15      * el id de esta ruta
16      */
17     private final int id_ruta;
18
19     /**
20      * numero de clientes (origen no incluido)
21      */
22     private int size;
23
24     /**
25      * La lista de vecinos mas cercanos de esta ruta desde todos los
            demas
26      */
27     private final TreeSet<Nodo>[] vecinosSalida;
28
29     /**
30      * La lista de vecinos mas cercanos de esta ruta hacia todos los
            demas
31      */
32     private final TreeSet<Nodo>[] vecinosLlegada;
33
34     /**
35      * Cache de los valores
36      */
37     private boolean cache;
38
39     private double cacheQi;
40     private double cacheCij;
41     private double cacheRCij;
42     private double cacheDi;
43
44     /**
45      * La lista de nodos dado su id
46      */
47     private Nodo[] nodos;
48
49     /**
50      * Construye una ruta vacía. crea un nodo origen
51      */
52     public Ruta() {
53         this(new int[0]);
54     }
55
```

```
56      /**
57       * El constructor a partir del array de nodos
58       */
59      public Ruta(int[] array) {
60          id_ruta = id_ruta_Next++;
61
62          vecinosSalida = new TreeSet[Grafo.getN() + 1];
63          vecinosLlegada = new TreeSet[Grafo.getN() + 1];
64          nodos = new Nodo[Grafo.getN() + 1];
65
66          cache = false;
67
68          initializeFromArray(array);
69      }
70
71      /**
72       * Realiza la construccion de las estructuras auxiliares pasando
             el array.
73       * Se pierde la información anterior
74       */
75      private void initializeFromArray(int[] array) {
76
77          size = array.length;
78
79          vecinosSalida[0] = new TreeSet<>(getComparator(0, true));
80          vecinosLlegada[0] = new TreeSet<>(getComparator(0, false));
81
82          Nodo origen = new Nodo(0);
83          nodos[0] = origen;
84
85          for (int i = 0; i < vecinosSalida.length; ++i) {
86              vecinosSalida[i] = new TreeSet<>(getComparator(i, true));
87              vecinosLlegada[i] = new TreeSet<>(getComparator(i, false));
88
89              vecinosSalida[i].add(origen);
90              vecinosLlegada[i].add(origen);
91          }
92
93          Nodo pre = origen;
94
95          for (int v = 0; v < size; v++) {
96              int id = array[v];
97              Nodo actual = new Nodo(id);
98              nodos[id] = actual;
99
100             pre.setNext(actual);
101             actual.setPrev(pre);
102
103             for (int i = 0; i < id; ++i) {
104                 vecinosSalida[i].add(actual);
105                 vecinosLlegada[i].add(actual);
106             }
107             for (int i = id + 1; i < vecinosSalida.length; ++i) {
108                 vecinosSalida[i].add(actual);
109                 vecinosLlegada[i].add(actual);
110             }
111
112             pre = actual;
```

```
113         }
114
115         pre.setNext(origen);
116         origen.setPrev(pre);
117
118         cache = false;
119     }
120
121
122     public Nodo getOrigen() {
123         return nodos[0];
124     }
125
126     /**
127      * Añade el nodo al final de la ruta (entre el último cliente y el
             origen)
128      */
129     public void añadirAlFinal(int insertar) {
130         Nodo nuevo = new Nodo(insertar);
131         nodos[insertar] = nuevo;
132
133         Nodo prev = nodos[0].getPrev();
134
135         prev.setNext(nuevo);
136         nuevo.setPrev(prev);
137
138         nuevo.setNext(nodos[0]);
139         nodos[0].setPrev(nuevo);
140
141         actualizarInsertado(nuevo);
142     }
143
144     /**
145      * actualiza las variables auxiliares si el nodo ahora pertenece a
             la ruta
146      */
147     public void actualizarInsertado(Nodo insertado) {
148
149         nodos[insertado.getId()] = insertado;
150
151         size++;
152
153         for (int i = 0; i < vecinosSalida.length; ++i) {
154             if (i == insertado.getId()) {
155                 continue;
156             }
157             vecinosSalida[i].add(insertado);
158             vecinosLlegada[i].add(insertado);
159         }
160
161         cache = false;
162     }
163
164     /**
165      * actualiza las variables auxiliares si el nodo ya no pertenece a
             la ruta
166      */
167     public void actualizarRemovido(Nodo removido) {
```

```
168
169         nodos[removido.getId()] = null;
170
171         size--;
172
173         for (int i = 0; i < vecinosSalida.length; ++i) {
174             if (i == removido.getId()) {
175                 continue;
176             }
177             vecinosSalida[i].remove(removido);
178             vecinosLlegada[i].remove(removido);
179         }
180
181         cache = false;
182     }
183
184     /**
185      * Devuelve la suma de los Qi.
186      * Valor guardado en cache
187      */
188     @Override
189     public double getSumQi() {
190         if (!cache) {
191             calculateCache();
192         }
193         return cacheQi;
194     }
195
196     /**
197      * Devuelve la suma de los Cij.
198      * Valor guardado en cache
199      */
200     @Override
201     public double getSumCij() {
202         if (!cache) {
203             calculateCache();
204         }
205         return cacheCij;
206     }
207
208     /**
209      * Devuelve la suma de los Cij si se recorre la ruta al reves.
210      * Valor guardado en cache
211      */
212     public double getReverseSumCij() {
213         if (!cache) {
214             calculateCache();
215         }
216         return cacheRCij;
217     }
218
219     /**
220      * Devuelve la suma de los Di.
221      * Valor guardado en cache
222      */
223     @Override
224     public double getSumDi() {
225         if (!cache) {
```

```
226             calculateCache();
227         }
228         return cacheDi;
229     }
230 
231     /**
232      * El numero de nodos en la ruta sin incluir el origen
233      */
234     public int getSize() {
235         return size;
236     }
237 
238     /**
239      * Devuelve una lista con todos los vecinos de esta ruta ordenados
             por distancia Cij desde el nodo dado
240      */
241     private TreeSet<Nodo> getVecinosSalida(int nodo) {
242         return vecinosSalida[nodo];
243     }
244 
245     /**
246      * Devuelve una lista con todos los vecinos de esta ruta ordenados
             por distancia Cij hacia el nodo dado
247      */
248     private TreeSet<Nodo> getVecinosLlegada(int nodo) {
249         return vecinosLlegada[nodo];
250     }
251 
252     /**
253      * Inicializa la variable pos de los nodos de esta ruta
254      */
255     public void updateAllPos() {
256 
257         Nodo it = getOrigen();
258 
259         for (int i = 0; i <= getSize(); ++i) {
260             it.setPos(i);
261 
262             it = it.getNext();
263 
264         }
265     }
266 
267     /**
268      * Devuelve un array con punteros a los nodos de la ruta en el
             orden actual
269      */
270     private Nodo[] getArrayOfNodos() {
271         Nodo[] array = new Nodo[size + 1];
272 
273         Nodo it = getOrigen();
274 
275         for (int i = 0; i <= getSize(); ++i) {
276             array[i] = it;
277             it = it.getNext();
278         }
279 
280         return array;
```

```
281     }
282
283     /**
284      * Calcula y almacena el cache
285      */
286     private void calculateCache() {
287
288         cacheCij = 0;
289         cacheRCij = 0;
290         cacheQi = 0;
291         cacheDi = 0;
292
293         Nodo it = nodos[0];
294         Nodo itNext = it.getNext();
295
296         for (int i = 0; i <= getSize(); ++i) {
297             cacheCij += Grafo.getCij(it.getId(), itNext.getId());
298             cacheRCij += Grafo.getCij(itNext.getId(), it.getId());
299             cacheQi += Grafo.getQi(it.getId());
300             cacheDi += Grafo.getDi(it.getId());
301
302             it = itNext;
303             itNext = itNext.getNext();
304         }
305
306         cache = true;
307     }
308
309     /**
310      * Una representación de la ruta en forma de String
311      */
312     @Override
313     public String toString() {
314         StringBuilder string = new StringBuilder();
315         string.append('[').append(size).append(']').append('\n');
316
317         Nodo it = nodos[0];
318         string.append("--> 0");
319
320         for (int i = 0; i <= getSize(); ++i) {
321             it = it.getNext();
322             string.append(",").append(it.getId());
323         }
324         if (it.getId() != 0) {
325             return "ERROR:\n" + string.toString();
326         }
327
328         string.append("\n<-- 0");
329         for (int i = 0; i <= getSize(); ++i) {
330             it = it.getPrev();
331             string.append(",").append(it.getId());
332         }
333         if (it.getId() != 0) {
334             return "ERROR:\n" + string.toString();
335         }
336
337         return string.toString();
338     }
```

```
339
340     /**
341      * Devuelve el id único de esta ruta
342      */
343     public int getId() {
344         return id_ruta;
345     }
346
347     /**
348      * Devuelve un array con los clientes de la ruta (nodos salvo el
             origen) empezando por el posterior al origen
349      * Si la ruta está vacía devuelve un array vacío
350      */
351     public int[] toArray() {
352         int[] array = new int[size];
353
354         Nodo it = nodos[0].getNext();
355         for (int i = 0; i < size; ++i) {
356             array[i] = it.getId();
357             it = it.getNext();
358         }
359
360         return array;
361     }
362
363     private Nodo getNodoFromId(int id) {
364         Nodo v = nodos[id];
365         return v;
366     }
367
368     ////////////////////////////////
369     /////// ALGORITMOS //////////////
370     ////////////////////////////////
371
372     /**
373      * El algoritmo Unstringing+Stringing. Modifica la ruta
374      */
375     public void us(int p) {
376
377         if (size < 2) {
378             return;//si el tamaño es 1 este algoritmo no merece la pena
379         }
380
381         Nodo[] taustar = getArrayOfNodos();
382         double zstar = getSumCij();
383
384         Nodo[] ordenFijo = getArrayOfNodos();
385         //guardamos el array antes, pues en cada iteracion el orden se
               modificará
386
387         int t = 1;//modificacion, el origen no lo tocamos
388
389         while (t < ordenFijo.length) {
390
391             Nodo vt = ordenFijo[t];//El nodo t
392
393             unstringing(vt.getId(), p).modificarRuta();
394             stringing_nodo(vt, p).modificarRuta();
```

```
395
396             if (getSumCij() < zstar) {
397                 taustar = getArrayOfNodos();
398                 zstar = getSumCij();
399                 t = 1;
400             } else {
401                 t++;
402             }
403
404         }
405
406         //reordenamos con la mejor ordenacion
407         for (int i = 1; i < taustar.length; i++) {
408             taustar[i - 1].setNext(taustar[i]);
409             taustar[i].setPrev(taustar[i - 1]);
410         }
411
412         taustar[taustar.length - 1].setNext(taustar[0]);
413         taustar[0].setPrev(taustar[taustar.length - 1]);
414
415     }
416
417     /**
418      * El algoritmo Stringing.
419      * Devuelve una ruta_delta resultado de añadir el nodo a la ruta
420      */
421     public RutaProxi stringing(int id, int p) {
422         return stringing_nodo(new Nodo(id), p);
423     }
424
425     private RutaProxi stringing_nodo(Nodo v, int p) {
426
427         //necesitamos al menos 3 elementos. El origen y dos mas
428         if (getSize() < 2) {
429             return stringing_minimal(v);
430         }
431
432         //Step 1
433         updateAllPos();
434
435         BestMove best = new BestMove();
436
437         //Step 2: find best move
438         stringing_direccional(v, p, false, best);
439
440         stringing_direccional(v, p, true, best);
441
442         //Step 3: return
443         if (best.vl == null) {
444             if (!best.reversed) {
445                 return new RP_insert_I(this, v, best.deltaCost,
                        best.vi, best.vj, best.vk);
446             } else {
447                 return new RP_insert_Ir(this, v, best.deltaCost,
                        best.vi, best.vj, best.vk);
448             }
449         } else {
450             if (!best.reversed) {
```

```
451                 return new RP_insert_II(this, v, best.deltaCost,
                        best.vi, best.vj, best.vk, best.vl);
452             } else {
453                 return new RP_insert_IIr(this, v, best.deltaCost,
                        best.vi, best.vj, best.vk, best.vl);
454             }
455         }
456     }
457
458     /**
459      * Cuando hay uno o dos nodos únicamente, calculamos la mejor
            manera de insertar otro de forma exacta
460      */
461     private RutaProxi stringing_minimal(Nodo v) {
462         switch (getSize()) {
463             case 0:
464                 //solo esta el origen, insertar tal cual
465                 return new RP_insert_1o2(new Nodo[]{v}, this, v,
                        Grafo.getCij(0, v.getId()) +
                        Grafo.getCij(v.getId(), 0));
466             case 1:
467
468                 //hay un nodo, insertar antes o despues
469                 Nodo vi = getOrigen().getNext();
470                 double costeInsertarAntes = -Grafo.getCij(0,
                        vi.getId()) + Grafo.getCij(0, v.getId()) +
                        Grafo.getCij(v.getId(), vi.getId());
471                 double costeInsertarDespues =
                        -Grafo.getCij(vi.getId(), 0) +
                        Grafo.getCij(v.getId(), vi.getId()) +
                        Grafo.getCij(vi.getId(), 0);
472
473                 if (costeInsertarAntes < costeInsertarDespues) {
474                     //insertarlo antes
475                     return new RP_insert_1o2(new Nodo[]{v, vi}, this,
                            v, costeInsertarAntes);
476                 } else {
477                     //insertarlo despues
478                     return new RP_insert_1o2(new Nodo[]{vi, v}, this,
                            v, costeInsertarDespues);
479                 }
480
481             default:
482                 throw new Error("no se reconoce el numero de nodos: "
                        + getSize());
483         }
484     }
485
486     /**
487      * Metodo de ayuda.
488      * Es el algoritmo como tal y calcula la mejor solucion para una
            direccion concreta
489      */
490     private void stringing_direccional(Nodo v, int p, boolean reverse,
            BestMove best) {
491
492         double reverseCost = 0;
493         if (reverse) {
```

```
494             reverseCost = -getSumCij() + getReverseSumCij();
495         }
496
497         TreeSet<Nodo> viPossible = getVecinosSalida(v.getId());
498         TreeSet<Nodo> vjPossible = getVecinosLlegada(v.getId());
499
500         Iterator<Nodo> viIterator = viPossible.iterator();
501         for (int i = 0; i < p && viIterator.hasNext(); ++i) {
502             Nodo vi = viIterator.next();
503
504             Iterator<Nodo> vjIterator = vjPossible.iterator();
505             for (int j = 0; j < p && vjIterator.hasNext(); ++j) {
506                 Nodo vj = vjIterator.next();
507
508                 if (vi.getId() == vj.getId()) {
509                     continue;
510                 }
511
512                 Nodo viNext = getNext(vi, reverse);
513                 Nodo vjNext = getNext(vj, reverse);
514
515                 TreeSet<Nodo> vkPossible =
                        getVecinosSalida(viNext.getId());
516                 TreeSet<Nodo> vlPossible =
                        getVecinosLlegada(vjNext.getId());
517
518                 Iterator<Nodo> vkIterator = vkPossible.iterator();
519                 for (int k = 0; k < p && vkIterator.hasNext(); ++k) {
520                     Nodo vk = vkIterator.next();
521                     if (!isBetween(vj, vk, vi, reverse)) {
522                         continue;
523                     }
524                     Nodo vkNext = getNext(vk, reverse);
525
526                     //Type I insertion
527                     if (vk.getId() != vi.getId() && vk.getId() !=
                            vj.getId()) {
528                         Double deltaCost = reverseCost;
529                         deltaCost -= Grafo.getCij(vi.getId(),
                                viNext.getId());
530                         deltaCost -= Grafo.getCij(vj.getId(),
                                vjNext.getId());
531                         deltaCost -= Grafo.getCij(vk.getId(),
                                vkNext.getId());
532
533                         deltaCost += Grafo.getCij(vi.getId(),
                                v.getId());
534                         deltaCost += Grafo.getCij(v.getId(),
                                vj.getId());
535                         deltaCost += Grafo.getCij(viNext.getId(),
                                vk.getId());
536                         deltaCost += Grafo.getCij(vjNext.getId(),
                                vkNext.getId());
537
538                         deltaCost += costOfReverseing(viNext, vj,
                                reverse);
539                         deltaCost += costOfReverseing(vjNext, vk,
                                reverse);
```

```
540
541                         if (!best.filled || deltaCost <
                                best.deltaCost) {
542                             best.fill(deltaCost, vi, vj, vk, null,
                                    reverse);
543                         }
544                     }
545
546                     Iterator<Nodo> vlIterator = vlPossible.iterator();
547                     for (int l = 0; l < p && vlIterator.hasNext();
                            ++l) {
548                         Nodo vl = vlIterator.next();
549                         if (!isBetween(vi, vl, vj, reverse)) {
550                             continue;
551                         }
552
553                         //Type II insertion
554                         if (vk.getId() != vj.getId()
555                                 && vk.getId() != vjNext.getId()
556                                 && vl.getId() != vi.getId()
557                                 && vl.getId() != viNext.getId()) {
558
559                             Double deltaCost = reverseCost;
560
561                             deltaCost -= Grafo.getCij(vi.getId(),
                                    viNext.getId());
562                             deltaCost -= Grafo.getCij(getPrev(vl,
                                    reverse).getId(), vl.getId());
563                             deltaCost -= Grafo.getCij(vj.getId(),
                                    vjNext.getId());
564                             deltaCost -= Grafo.getCij(getPrev(vk,
                                    reverse).getId(), vk.getId());
565
566                             deltaCost += Grafo.getCij(vi.getId(),
                                    v.getId());
567                             deltaCost += Grafo.getCij(v.getId(),
                                    vj.getId());
568                             deltaCost += Grafo.getCij(vl.getId(),
                                    vjNext.getId());
569                             deltaCost += Grafo.getCij(getPrev(vk,
                                    reverse).getId(), getPrev(vl,
                                    reverse).getId());
570                             deltaCost += Grafo.getCij(viNext.getId(),
                                    vk.getId());
571
572                             deltaCost += costOfReverseing(viNext,
                                    getPrev(vl, reverse), reverse);
573                             deltaCost += costOfReverseing(vl, vj,
                                    reverse);
574
575                             if (!best.filled || deltaCost <
                                    best.deltaCost) {
576                                 best.fill(deltaCost, vi, vj, vk, vl,
                                        reverse);
577                             }
578
579                         }
580
```

```
581                     }
582
583                 }
584
585             }
586
587         }
588     }
589
590     /**
591      * Realiza el procedimiento unstringing.
592      * Devuelve la ruta_delta que se crea al quitar el nodo de la ruta
593      */
594     public RutaProxi unstringing(int id, int p) {
595
596         Nodo v = getNodoFromId(id);
597
598         //necesitamos al menos tres elementos: el origen, el que
                quitamos y otro
599         if (getSize() < 3) {
600             return unstringing_minimal(v);
601         }
602
603         updateAllPos();
604
605         BestMove best = new BestMove();
606
607         unstringing_direccional(v, p, false, best);
608
609         unstringing_direccional(v, p, true, best);
610
611         if (best.vl == null) {
612             if (!best.reversed) {
613                 return new RP_remove_I(this, v, best.deltaCost,
                        best.vj, best.vk);
614             } else {
615                 return new RP_remove_Ir(this, v, best.deltaCost,
                        best.vj, best.vk);
616             }
617         } else {
618             if (!best.reversed) {
619                 return new RP_remove_II(this, v, best.deltaCost,
                        best.vj, best.vk, best.vl);
620             } else {
621                 return new RP_remove_IIr(this, v, best.deltaCost,
                        best.vj, best.vk, best.vl);
622             }
623         }
624     }
625
626     /**
627      * cuando hay dos o tres nodos, lo quitamos de la mejor manera
            posible
628      */
629     private RutaProxi unstringing_minimal(Nodo v) {
630         switch (getSize()) {
631             case 1:
632
```

```
633                 Double deltaCost = -Grafo.getCij(0, v.getId()) -
                        Grafo.getCij(v.getId(), 0);
634
635                 //eliminarlo
636                 return new RP_remove_0(this, v, deltaCost);
637
638             case 2:
639                 //hay un nodo extra, lo juntamos con el cero
640                 Nodo a = getOrigen().getNext();
641                 if (a.getId() == v.getId()) {
642                     a = a.getNext();
643                 }
644
645                 deltaCost = -getSumCij() + Grafo.getCij(0, a.getId())
                        + Grafo.getCij(a.getId(), 0);
646
647                 return new RP_remove_1(a, this, v, deltaCost);
648
649             default:
650                 throw new Error("no se reconoce el numero de nodos: "
                        + getSize());
651         }
652     }
653
654     /**
655      * Metodo de ayuda.
656      * Es el algoritmo como tal y calcula la mejor solucion para una
            direccion concreta
657      */
658     private void unstringing_direccional(Nodo vi, int p, boolean
            reverse, BestMove best) {
659
660         double reverseCost = 0;
661         if (reverse) {
662             reverseCost = -getSumCij() + getReverseSumCij();
663         }
664
665         Nodo viNext = getNext(vi, reverse);
666         Nodo viPrev = getPrev(vi, reverse);
667
668         TreeSet<Nodo> vjPossible = getVecinosSalida(viNext.getId());
669         Iterator<Nodo> vjIterator = vjPossible.iterator();
670         for (int i = 0; i < p && vjIterator.hasNext(); ++i) {
671             Nodo vj = vjIterator.next();
672             if (vj.getId() == vi.getId() || vj.getId() ==
                    viNext.getId() || vj.getId() == viPrev.getId()) {
673                 continue;
674             }
675
676             Nodo vjNext = getNext(vj, reverse);
677             Nodo vjPrev = getPrev(vj, reverse);
678
679             TreeSet<Nodo> vkPossible =
                    getVecinosSalida(viPrev.getId());
680             Iterator<Nodo> vkIterator = vkPossible.iterator();
681             for (int k = 0; k < p && vkIterator.hasNext(); ++k) {
682                 Nodo vk = vkIterator.next();
683                 if (vk.getId() == vi.getId()) {
```

```
684                     continue;
685                 }
686
687                 Nodo vkNext = getNext(vk, reverse);
688
689                 if (isBetween(viNext, vk, vjPrev, reverse)) {
690
691                     //TYPE I
692                     Double deltaCost = reverseCost;
693                     deltaCost -= Grafo.getCij(viPrev.getId(),
                            vi.getId());
694                     deltaCost -= Grafo.getCij(vi.getId(),
                            viNext.getId());
695                     deltaCost -= Grafo.getCij(vk.getId(),
                            vkNext.getId());
696                     deltaCost -= Grafo.getCij(vj.getId(),
                            vjNext.getId());
697
698                     deltaCost += Grafo.getCij(viPrev.getId(),
                            vk.getId());
699                     deltaCost += Grafo.getCij(viNext.getId(),
                            vj.getId());
700                     deltaCost += Grafo.getCij(vkNext.getId(),
                            vjNext.getId());
701
702                     deltaCost += costOfReverseing(viNext, vk, reverse);
703                     deltaCost += costOfReverseing(vkNext, vj, reverse);
704
705                     if (!best.filled || deltaCost < best.deltaCost) {
706                         best.fill(deltaCost, null, vj, vk, null,
                                reverse);
707                     }
708
709                 }
710
711                 if (!isBetween(vjNext, vk, getPrev(viPrev, reverse),
                        reverse) || vjNext.getId() == viPrev.getId()) {
712                     continue;
713                 }
714
715                 TreeSet<Nodo> vlPossible =
                        getVecinosLlegada(vkNext.getId());
716                 Iterator<Nodo> vlIterator = vlPossible.iterator();
717                 for (int l = 0; l < p && vlIterator.hasNext(); ++l) {
718                     Nodo vl = vlIterator.next();
719                     if (vl.getId() == vi.getId()) {
720                         continue;
721                     }
722
723                     Nodo vlNext = getNext(vl, reverse);
724
725                     if (!isBetween(vj, vl, getPrev(vk, reverse),
                            reverse)) {
726                         continue;
727                     }
728
729                     //TYPE II
730                     Double deltaCost = reverseCost;
```

```
731                     deltaCost -= Grafo.getCij(viPrev.getId(),
                            vi.getId());
732                     deltaCost -= Grafo.getCij(vi.getId(),
                            viNext.getId());
733                     deltaCost -= Grafo.getCij(vjPrev.getId(),
                            vj.getId());
734                     deltaCost -= Grafo.getCij(vl.getId(),
                            vlNext.getId());
735                     deltaCost -= Grafo.getCij(vk.getId(),
                            vkNext.getId());
736
737                     deltaCost += Grafo.getCij(viPrev.getId(),
                            vk.getId());
738                     deltaCost += Grafo.getCij(vlNext.getId(),
                            vjPrev.getId());
739                     deltaCost += Grafo.getCij(viNext.getId(),
                            vj.getId());
740                     deltaCost += Grafo.getCij(vl.getId(),
                            vkNext.getId());
741
742                     deltaCost += costOfReverseing(viNext, vjPrev,
                            reverse);
743                     deltaCost += costOfReverseing(vlNext, vk, reverse);
744
745                     if (!best.filled || deltaCost < best.deltaCost) {
746                         best.fill(deltaCost, null, vj, vk, vl,
                                reverse);
747                     }
748
749                 }
750
751             }
752         }
753     }
754
755     ////////////////////////////////
756     ///// Funciones auxiliares //////
757     ////////////////////////////////
758
759     /**
760      * Calcula la diferencia entre recorrer la subruta start-end en
            dirección contraria menos recorrerla en dirección normal
761      */
762     private double costOfReverseing(Nodo start, Nodo end, boolean
            reverse) {
763         Nodo it = start;
764         Nodo itNext = getNext(start, reverse);
765         double cost = 0;
766         while (it.getId() != end.getId()) {
767             cost -= Grafo.getCij(it.getId(), itNext.getId());
768             cost += Grafo.getCij(itNext.getId(), it.getId());
769
770             it = itNext;
771             itNext = getNext(itNext, reverse);
772         }
773
774         return cost;
775     }
```

```
776
777     /**
778      * funcion de ayuda: devuelve el nodo siguiente de la ruta
             considerando el sentido:
779      * el nodo siguiente si recorremos normal, el anterior si
             recorremos al reves
780      */
781     private Nodo getNext(Nodo v, boolean reverse) {
782         return reverse ? v.getPrev() : v.getNext();
783     }
784
785     /**
786      * funcion de ayuda: devuelve el nodo anterior de la ruta
             considerando el sentido:
787      * el nodo anterior si recorremos normal, el siguiente si
             recorremos al reves
788      */
789     private Nodo getPrev(Nodo v, boolean reverse) {
790         return reverse ? v.getNext() : v.getPrev();
791     }
792
793     /**
794      * Comprueba si los tres nodos están en orden -abc- considerando
             el sentido?
795      */
796     private boolean isBetween(Nodo a, Nodo b, Nodo c, boolean reverse)
        {
797
798         int s = reverse ? -1 : 1;
799
800         int aP = a.getPos() * s;
801         int bP = b.getPos() * s;
802         int cP = c.getPos() * s;
803
804         //sentido normal
805         if (aP <= cP) {
806             // a c
807             return (aP <= bP && bP <= cP);// a b c = bien
808         } else {
809             //c a
810             return !(cP < bP && bP < aP);// c b a = mal
811         }
812
813     }
814
815     ////////////////////////////////
816     /////// Clases auxiliares ///////
817     ////////////////////////////////
818
819     /**
820      * Un movimiento indicando el nuevo coste,
821      * vi,vj,vk, vl (si existe es el tipo II si no es el tipo I)
822      * y si se ha considerado el cambiar el sentido
823      */
824     private static class BestMove {
825
826         private double deltaCost;
827         private Nodo vi;
```

```
828         private Nodo vj;
829         private Nodo vk;
830         private Nodo vl;
831         private boolean reversed;
832 
833         private boolean filled = false;
834 
835         private void fill(double reduction, Nodo vi, Nodo vj, Nodo vk,
                Nodo vl, boolean reversed) {
836             this.deltaCost = reduction;
837             this.vi = vi;
838             this.vj = vj;
839             this.vk = vk;
840             this.vl = vl;
841             this.reversed = reversed;
842 
843             filled = true;
844         }
845 
846     }
847 }
```

Clase 18: RutaAbstracta.java

```
1  import java.util.Comparator;
2  import java.util.HashMap;
3
4  /**
5   * Representa una ruta genérica.
6   * Contiene la subclase Nodo
7   */
8  public abstract class RutaAbstracta {
9
10     abstract double getSumQi();
11
12     abstract double getSumDi();
13
14     abstract double getSumCij();
15
16     public boolean isFactible_capacidad() {
17         return getSumQi() <= Grafo.getQ();
18     }
19
20     public boolean isFactible_longitud() {
21         return getSumCij() + getSumDi() <= Grafo.getL();
22     }
23
24     public boolean isFactible() {
25         return isFactible_capacidad() && isFactible_longitud();
26     }
27
28     public double getF1() {
29         return getSumCij();
30     }
31
32     public double getF2() {
33         double res = getF1();
34         double temp;
35
36         temp = getSumQi() - Grafo.getQ();
37         if (temp > 0) {
38             res += TabuRoute.getAlpha() * temp;
39         }
40
41         temp = getSumCij() + getSumDi() - Grafo.getL();
42         if (temp > 0) {
43             res += TabuRoute.getBeta() * temp;
44         }
45
46         return res;
47     }
48
49     ////////////////////////
50     /// clases auxiliares ///
51     ////////////////////////
52
53     /**
54      * Representa un nodo de una ruta.
55      * Implementación de una lista doblemente enlazada.
56      */
57     static class Nodo {
```

```
58
59          /**
60           * El nodo que representa (0 es el origen)
61           */
62          private final int id;
63
64          /**
65           * El nodo anterior
66           */
67          private Nodo prev = null;
68
69          /**
70           * El nodo posterior
71           */
72          private Nodo next = null;
73
74          /**
75           * La posicion del nodo en la ruta.
76           * Valor modificado externamente
77           */
78          private int pos = -1;
79
80
81          public Nodo(int id) {
82              this.id = id;
83          }
84
85          public int getId() {
86              return id;
87          }
88
89          public Nodo getNext() {
90              return next;
91          }
92
93          public Nodo getPrev() {
94              return prev;
95          }
96
97          public void setPrev(Nodo prev) {
98              this.prev = prev;
99          }
100
101         public void setNext(Nodo next) {
102             this.next = next;
103         }
104
105         /**
106          * intercambia el nodo anterior con el posterior
107          */
108         public void reverse() {
109             Nodo temp = next;
110             next = prev;
111             prev = temp;
112         }
113
114         public int getPos() {
115             return pos;
```

```
116         }
117
118         public void setPos(int pos) {
119             this.pos = pos;
120         }
121
122         @Override
123         public String toString() {
124             return "(" + (prev == null ? "null" : prev.getId()) +
                    ")->" + id + "->(" + (next == null ? "null" :
                    next.getId()) + ") [" + pos + "]";
125         }
126
127     }
128
129     /**
130      * Los contenedores de los comparadores
131      */
132     private static final HashMap<Integer, NodoComparador>
            comparatorsSalida = new HashMap<>();
133     private static final HashMap<Integer, NodoComparador>
            comparatorsLlegada = new HashMap<>();
134
135     /**
136      * Devuelve el comparador dado el de partida y la direccion.
137      * Sucesivas llamadas con los mismos parametros de entrada
             devuelven el mismo objeto (cache)
138      */
139     static public NodoComparador getComparator(int partida, boolean
            salida) {
140         HashMap<Integer, NodoComparador> holder = salida ?
                comparatorsSalida : comparatorsLlegada;
141
142         if (holder.containsKey(partida)) {
143             return holder.get(partida);
144         } else {
145             NodoComparador comp = new NodoComparador(partida, salida);
146             holder.put(partida, comp);
147             return comp;
148         }
149     }
150
151     /**
152      * Un comparador de nodos dado uno de partida y la dirección
153      */
154     static public class NodoComparador implements Comparator<Nodo> {
155
156         /**
157          * El nodo con el que se compararan
158          */
159         private final int partida;
160         /**
161          * si considerar partida-otro
162          */
163         private final boolean salida;
164
165
166         private NodoComparador(int origin, boolean salida) {
```

```
167             this.partida = origin;
168             this.salida = salida;
169         }
170 
171         @Override
172         public int compare(Nodo v1, Nodo v2) {
173             double dif = salida
174                     ? Grafo.getCij(partida, v1.getId()) -
                           Grafo.getCij(partida, v2.getId())
175                     : Grafo.getCij(v1.getId(), partida) -
                           Grafo.getCij(v2.getId(), partida);
176 
177             //si 0 discriminar por id
178             if (dif == 0) {
179                 dif = v1.getId() - v2.getId();
180             }
181 
182             return dif > 0 ? //si es positivo
183                     1 //devolvemos 1
184                     : dif < 0 ? //si es negativo
185                             -1 //devolvemos -1
186                             : 0 //si no, iguales, 0
187                     ;
188         }
189 
190     }
191 }
```

Clase 19: RutaProxi.java

```
1  import java.util.ConcurrentModificationException;
2
3  /**
4   * La interfaz de modificacion de una ruta.
5   * Representa una ruta resultado de realizar una modificación a una
        ruta existente, sin realizar la modificación.
6   */
7  public abstract class RutaProxi extends RutaAbstracta {
8
9      /**
10      * El nodo que se moverá
11      */
12     private final Nodo v;
13
14     /**
15      * La ruta original. Será modificada
16      */
17     private Ruta original;
18
19     /**
20      * La diferencia de los costes: nueva-antigua
21      */
22     private final double deltaCost;
23
24
25     public RutaProxi(Ruta original, Nodo v, double deltaCost) {
26         this.original = original;
27         this.deltaCost = deltaCost;
28         this.v = v;
29     }
30
31
32     public Ruta getRuta() {
33         checkModification();
34
35         return original;
36     }
37
38     public double getDeltaCost() {
39         checkModification();
40
41         return deltaCost;
42     }
43
44     public Nodo getV() {
45         checkModification();
46
47         return v;
48     }
49
50     /**
51      * La suma de los Cij de la ruta modificada
52      */
53     @Override
54     public double getSumCij() {
55         checkModification();
56
```

```
57          return getRuta().getSumCij() + getDeltaCost();
58      }
59
60      /**
61       * La suma de los Qi de la ruta modificada
62       */
63      @Override
64      double getSumQi() {
65          checkModification();
66
67          return 0;
68      }
69
70      /**
71       * La suma de los Di de la ruta modificacion
72       */
73      @Override
74      double getSumDi() {
75          checkModification();
76
77          return 0;
78      }
79
80      /**
81       * Modifica la solucion original.
82       * Una vez llamado esta funcion, todas las funciones de este
             objeto lanzarán una ConcurrentModificationException
83       */
84      public void modificarRuta() {
85          checkModification();
86
87          internal_modificar();
88
89          original = null;
90      }
91
92      abstract void internal_modificar();
93
94      /**
95       * Lanza una excepción ConcurrentModificationException si la ruta
             ya ha sido modificada (no hay ruta original)
96       */
97      private void checkModification() {
98          if (original == null) {
99              throw new ConcurrentModificationException("Se ha intentado
                    acceder a una RutaProxi tras haber efectuado la
                    modificación");
100         }
101     }
102
103     //////////////
104     //Utilidades//
105     //////////////
106
107     /**
108      * Cambia next<->prev en cada uno de los nodos desde from hasta to
             incluidos
109      */
```

```
110     void reverseNodos(Nodo from, Nodo to) {
111         Nodo it = from;
112 
113         while (it.getId() != to.getId()) {
114             it.reverse();
115             it = it.getPrev();//el anterior, que originalmente era el
                    siguiente
116         }
117         it.reverse();
118     }
119 
120     /**
121      * junta los nodos haciendo: from.next=to to.prev=from
122      */
123     void join(Nodo from, Nodo to) {
124         from.setNext(to);
125         to.setPrev(from);
126     }
127 
128 }
```

Clase 20: Solucion.java

```
1  import java.util.Arrays;
2  import java.util.Collection;
3  import java.util.HashMap;
4
5  /**
6   * Representa una solución: un conjunto de rutas
7   */
8  public class Solucion extends SolucionAbstracta {
9
10     /**
11      * El set de rutas.
12      */
13     private final HashMap<Integer, Ruta> arrayRutas;
14
15     /**
16      * Una ruta vacia para realizar comprobaciones
17      */
18     private Ruta rutaVacia;
19
20     //auxiliares
21
22     /**
23      * A qué ruta pertenece qué nodo
24      */
25     private final int[] pertenencia;
26
27     /**
28      * Constructor base, genera una solución vacía, sin nodos
29      */
30     public Solucion() {
31         arrayRutas = new HashMap<>();
32
33         rutaVacia = new Ruta();
34
35         pertenencia = new int[Grafo.getN() + 1];
36         Arrays.fill(pertenencia, -1);
37
38     }
39
40     /**
41      * Constructor fromArray()
42      */
43     public Solucion(int[][] Sstar) {
44         this();
45
46         for (int[] ruta : Sstar) {
47             añadirRuta(new Ruta(ruta));
48         }
49     }
50
51     /**
52      * Crea una solucion factible (o intenta serlo) dado el array de
            nodos en orden
53      */
54     public Solucion(int[] array) {
55         this();
56
```

```
57          Ruta rutaActual = new Ruta();
58
59          int preInsertar = array[0];
60          rutaActual.añadirAlFinal(preInsertar);
61
62          int pos = 1;
63
64          while (pos < array.length) {
65              int insertar = array[pos];
66              if (getm() >= Grafo.getMbar()) {
67                  break;
68              }
69              if (rutaActual.getSumQi() + Grafo.getQi(insertar) <=
                    Grafo.getQ() && rutaActual.getSumCij() -
                    Grafo.getCij(preInsertar, 0) +
                    Grafo.getCij(preInsertar, insertar) +
                    Grafo.getCij(insertar, 0) + rutaActual.getSumDi() +
                    Grafo.getDi(insertar) <= Grafo.getL()) {
70                  rutaActual.añadirAlFinal(insertar);
71                  preInsertar = insertar;
72              } else {
73                  añadirRuta(rutaActual);
74                  rutaActual = new Ruta();
75                  rutaActual.añadirAlFinal(insertar);
76                  preInsertar = insertar;
77              }
78              pos++;
79          }
80
81          while (pos < array.length) {
82              int insertar = array[pos];
83              rutaActual.añadirAlFinal(insertar);
84              pos++;
85          }
86
87          añadirRuta(rutaActual);
88      }
89
90
91      @Override
92      public double getF1() {
93          double res = 0;
94          for (Ruta r : arrayRutas.values()) {
95              res += r.getF1();
96          }
97          return res;
98      }
99
100
101     @Override
102     public double getF2() {
103         double res = 0;
104         for (Ruta r : arrayRutas.values()) {
105             res += r.getF2();
106         }
107         return res;
108     }
109
```

```
110     /**
111      * Añade la ruta a la lista de rutas.
112      * Se queda con el objeto
113      */
114     public void añadirRuta(Ruta nuevaRuta) {
115         int idRuta = nuevaRuta.getId();
116         arrayRutas.put(idRuta, nuevaRuta);
117 
118         updateRouteWithId(idRuta);
119 
120     }
121 
122     /**
123      * El numero de rutas con al menos algun cliente
124      */
125     public int getm() {
126         return arrayRutas.size();
127     }
128 
129     /**
130      * Si la solución cumple la restricción de capacidad
131      */
132     public boolean isFactible_capacidad() {
133         for (Ruta r : arrayRutas.values()) {
134             if (!r.isFactible_capacidad()) {
135                 return false;
136             }
137         }
138         return true;
139     }
140 
141     /**
142      * Si la solución cumple la restricción de longitud
143      */
144     public boolean isFactible_longitud() {
145         for (Ruta r : arrayRutas.values()) {
146             if (!r.isFactible_longitud()) {
147                 return false;
148             }
149         }
150         return true;
151     }
152 
153     /**
154      * Si la solución es factible
155      */
156     @Override
157     public boolean isFactible() {
158         return isFactible_capacidad() && isFactible_longitud();
159     }
160 
161     /**
162      * Devuelve la ruta a la que pertenece el nodo dado
163      */
164     public int getIdOfRutaContainingNodo(int v) {
165         return pertenencia[v];
166     }
167 
```

```
168     /**
169      * Devuelve el id de la ruta vacía, usado como ruta de comprobación
170      */
171     public int getIdOfEmptyRoute() {
172         return rutaVacia.getId();
173     }
174 
175     /**
176      * Devuelve la ruta de id dado
177      */
178     public Ruta getRouteOfId(int id) {
179         if (id == rutaVacia.getId()) {
180             return rutaVacia;
181         }
182 
183         return arrayRutas.get(id);
184     }
185 
186     /**
187      * Notifica de que la ruta ha cambiado
188      */
189     public void updateRouteWithId(int id) {
190 
191         if (id == rutaVacia.getId()) {
192             if (rutaVacia.getSize() != 0) {
193                 Ruta nueva = rutaVacia;
194                 rutaVacia = new Ruta();
195                 añadirRuta(nueva);
196                 //se realiza una llamada a este metodo otra vez.
197                 //Si la rutaVacia no se modifica hay un bucle infinito
198             }
199         }
200 
201         Ruta ruta = getRouteOfId(id);
202 
203         if (ruta.getSize() == 0) {
204             arrayRutas.remove(ruta.getId());
205             return;
206         }
207 
208         int[] nodos = ruta.toArray();
209         for (int nodo : nodos) {
210             pertenencia[nodo] = id;
211         }
212     }
213 
214     /**
215      * Le aplica US a esta solucion
216      */
217     public void applyUS(int p) {
218         for (Ruta r : arrayRutas.values()) {
219             r.us(p);
220         }
221 
222     }
223 
224     /**
225      * Devuelve una colección con todas las rutas no vacías de esta
```

```
         solución
226       */
227      public Collection<Ruta> getAllRoutes() {
228          return arrayRutas.values();
229      }
230
231      /**
232       * Devuelve un array representando la información de esta solución:
233       * Cada elemento es un array que representa una ruta
234       */
235      public int[][] toArray() {
236          int[][] array = new int[arrayRutas.size()][];
237
238          int i = 0;
239          for (Ruta ruta : arrayRutas.values()) {
240              array[i] = ruta.toArray();
241              i++;
242          }
243
244          return array;
245      }
246
247  }
```

Clase 21: SolucionAbstracta.java

```
1  public abstract class SolucionAbstracta {
2
3      /**
4       * devuelve el valor de F1 de esta solucion
5       */
6      public abstract double getF1();
7
8      /**
9       * Devuelve el valor de F2 de esta solucion
10      */
11     public abstract double getF2();
12
13     /**
14      * Si la solución es factible
15      */
16     public abstract boolean isFactible();
17
18 }
```

Clase 22: SolucionProxi.java

```
1  import java.util.ConcurrentModificationException;
2
3  /**
4   * La clase que representa una modificación de una solución.
5   * Esta modificación consiste en pasar un nodo de una ruta a otra
6   */
7  public class SolucionProxi extends SolucionAbstracta {
8
9      /**
10      * La solucion original
11      */
12     private Solucion base;
13
14     /**
15      * El nodo que se moverá
16      */
17     private int v;
18
19     /**
20      * La ruta de donde se quita el nodo
21      */
22     private RutaProxi rr;
23
24     /**
25      * El id de la ruta Rr
26      */
27     private int idRr;
28
29     /**
30      * La ruta en donde se añadirá el nodo
31      */
32     private RutaProxi rs;
33
34     /**
35      * El id de la ruta Rs
36      */
37     private int idRs;
38
39     /**
40      * Un valor, calculado externamente, que 'valora' este movimiento
41      */
42     private double f;
43
44
45     public SolucionProxi(Solucion base) {
46         this.base = base;
47     }
48
49     /**
50      * Constructor copia
51      */
52     public SolucionProxi(SolucionProxi original) {
53         base = original.base;
54         v = original.v;
55         rr = original.rr;
56         idRr = original.idRr;
57         rs = original.rs;
```

```
58          idRs = original.idRs;
59          f = original.f;
60      }
61
62      /**
63       * Indica el nodo que se modificará.
64       */
65      public void setNodo(int v) {
66          checkModification();
67
68          this.v = v;
69      }
70
71
72      public int getNodo() {
73          checkModification();
74
75          return v;
76      }
77
78
79      public int getRr() {
80          checkModification();
81
82          return idRr;
83      }
84
85      /**
86       * Indica la ruta de la que se quitará el nodo.
87       * Genera y almacena una ruta_delta con la información
88       */
89      public void setRr(int idRuta, int p) {
90          checkModification();
91
92          idRr = idRuta;
93          rr = base.getRouteOfId(idRuta).unstringing(v, p);
94      }
95
96      /**
97       * Indica la ruta a la que se añadirá el nodo.
98       * Genera y almacena una ruta_delta con la información
99       */
100     public void setRs(int route, int p) {
101         checkModification();
102
103         idRs = route;
104         rs = base.getRouteOfId(route).stringing(v, p);
105     }
106
107     /**
108      * Si la nueva solución es factible o no
109      */
110     @Override
111     public boolean isFactible() {
112         checkModification();
113
114         for (Ruta r : base.getAllRoutes()) {
115             int id = r.getId();
```

```
116             if (id == idRr || id == idRs) {
117                 continue;
118             }
119             if (!r.isFactible()) {
120                 return false;
121             }
122         }
123 
124         if (!rr.isFactible()) {
125             return false;
126         }
127         if (!rs.isFactible()) {
128             return false;
129         }
130         return true;
131     }
132 
133     /**
134      * El valor de F1 de la nueva solución
135      */
136     @Override
137     public double getF1() {
138         checkModification();
139 
140         double res = 0;
141         for (Ruta r : base.getAllRoutes()) {
142             int id = r.getId();
143             if (id == idRr || id == idRs) {
144                 continue;
145             }
146             res += r.getF1();
147         }
148 
149         res += rr.getF1();
150         res += rs.getF1();
151         return res;
152     }
153 
154     /**
155      * El valor de F2 de la nueva solución
156      */
157     @Override
158     public double getF2() {
159         checkModification();
160 
161         double res = 0;
162         for (Ruta r : base.getAllRoutes()) {
163             int id = r.getId();
164             if (id == idRr || id == idRs) {
165                 continue;
166             }
167             res += r.getF2();
168         }
169 
170         res += rr.getF2();
171         res += rs.getF2();
172         return res;
173     }
```

```
174
175
176     public void setF(double f) {
177         checkModification();
178
179         this.f = f;
180     }
181
182
183     public double getF() {
184         checkModification();
185
186         return f;
187     }
188
189     /**
190      * Realiza la modificación.
191      * Una vez llamado esta funcion, todas las funciones de este
             objeto lanzarán una ConcurrentModificationException
192      */
193     public void modificarSolucion() {
194         checkModification();
195
196         rr.modificarRuta();
197         rs.modificarRuta();
198         base.updateRouteWithId(idRs);
199         base.updateRouteWithId(idRr);
200
201         base = null;
202     }
203
204     /**
205      * Lanza una excepción ConcurrentModificationException si ya se ha
             modificado este objeto (no hay solucion base)
206      */
207     private void checkModification() {
208         if (base == null) {
209             throw new ConcurrentModificationException("Se ha intentado
                    acceder a una SolucionProxi tras haber efectuado la
                    modificación");
210         }
211     }
212 }
```

Clase 23: TabuRoute.java

```
1  import java.util.ArrayList;
2  import java.util.Arrays;
3  import java.util.Collections;
4  import java.util.HashSet;
5  import java.util.Random;
6  import java.util.concurrent.ThreadLocalRandom;
7
8  /**
9   * Esta clase contiene los algoritmos principales del problema:
        TabuRoute y Search
10  */
11 public class TabuRoute {
12
13     /**
14      * La solucion actual del problema, con la que los algoritmos
           trabajan
15      */
16     private static Solucion s;
17
18     /**
19      * El mayor valor de F1 encontrado
20      */
21     private static double f1star;
22
23     /**
24      * El mayor valor de F2 encontrado
25      */
26     private static double f2star;
27
28     /**
29      * La mejor solucion factible encontrada
30      */
31     private static int[][] sstar;
32
33     /**
34      * La mejor solución encontrada
35      */
36     private static int[][] stildestar;
37
38     /**
39      * El numero de veces que cada nodo ha sido movido. fv[0] no se usa
40      */
41     private static int[] fv;
42
43     //Variables de algoritmo
44
45     /**
46      * El valor alpha
47      */
48     private static double alpha;
49
50     /**
51      * El valor beta
52      */
53     private static double beta;
54
55
```

```
56      public static double getAlpha() {
57          return alpha;
58      }
59
60
61      public static void setAlpha(double alpha) {
62          TabuRoute.alpha = alpha;
63      }
64
65
66      public static double getBeta() {
67          return beta;
68      }
69
70
71      public static void setBeta(double beta) {
72          TabuRoute.beta = beta;
73      }
74
75      /**
76       * Inicia el algoritmo de resolucion y devuelve la solución.
77       * Esta es la función que se debe llamar tras haber inicializado
             Grafo
78       */
79      public static int[][] ejecutar() {
80          return tabuRoute((int) Math.floor(Math.sqrt(Grafo.getN()) /
                2));
81      }
82
83      /**
84       * ************************
85       *  El algoritmo taburoute
86       * ************************
87       * @param lambda el numero de soluciones iniciales distintas
88       */
89      private static int[][] tabuRoute(int lambda) {
90
91          //Step -1
92          stildestar = null;
93          sstar = null;
94          f2star = Grafo.getInfinity();
95          f1star = Grafo.getInfinity();
96          int n = Grafo.getN();
97          fv = new int[n + 1];
98
99          //Step 0
100         alpha = 1;
101         beta = 1;
102         f1star = Grafo.getInfinity();
103
104         Random random = new Random();
105
106         //Step 1
107         for (int step = 0; step < lambda; ++step) {
108             System.out.println("Landa " + step);
109             //Step 1 (a)
110             int i = random.nextInt(n) + 1;
111
```

```
112             //Step 1 (b)
113             int[] sequence = new int[n];
114             for (int j = i; j <= n; ++j) {
115                 sequence[j - i] = j;
116             }
117             for (int j = 1; j < i; ++j) {
118                 sequence[j + n - i] = j;
119             }
120 
121             //Step 1 (b) y (c)
122             int[] tour = Genius.ejecutar(sequence);
123 
124             //Step 1 (c)
125             s = new Solucion(tour);
126 
127             if (s.isFactible() && s.getF1() < f1star) {
128                 f1star = s.getF1();
129                 sstar = s.toArray();
130                 //System.out.println("Nueva
                        solución:"+Sstar.toString());
131             }
132             if (s.getF2() < f2star) {
133                 f2star = s.getF2();
134                 stildestar = s.toArray();
135             }
136 
137             //Step 1 (d)
138             int[] todos = new int[n];
139             for (int ii = 0; ii < n; ++ii) {
140                 todos[ii] = ii + 1;
141             }
142             search(todos, 5 * s.getm(), 0, 5, 5, 10, 0.01, 10, n);
143 
144         }
145 
146         //Step 1 (e)
147         //Error pdf: esto va fuera del bucle
148         if (f1star < Grafo.getInfinity()) {
149             s = new Solucion(sstar);
150         } else {
151             s = new Solucion(stildestar);
152         }
153 
154         //Step 2
155         System.out.println("Diversificación");
156         int[] todos = new int[n];
157         for (int ii = 0; ii < n; ++ii) {
158             todos[ii] = ii + 1;
159         }
160         search(todos, 5 * s.getm(), 0, 5, 5, 10, 0.01, 10, 50 * n);
161         if (f1star < Grafo.getInfinity()) {
162             s = new Solucion(sstar);
163         } else {
164             s = new Solucion(stildestar);
165         }
166 
167         //Step 3
168         int[] handler = Arrays.copyOf(fv, n);
```

```
169         Arrays.sort(handler);
170         int half = handler[n / 2];
171 
172         int mostUsed = 0;
173         for (int ii = 0; ii < fv.length; ++ii) {
174             if (fv[ii] >= half) {
175                 handler[mostUsed++] = ii;
176             }
177         }
178 
179         System.out.println("Intensificación");
180         search(Arrays.copyOf(handler, mostUsed), mostUsed, 0, 5, 5,
                10, 0.01, 10, n);
181         if (f1star < Grafo.getInfinity()) {
182             return sstar;
183         } else {
184             return null;
185         }
186     }
187 
188     /**
189      * *********************
190      *  El algoritmo SEARCH
191      * *********************
192      *
193      * @param W El conjunto de nodos que se pueden mover
194      * @param q El numero de nodos que se moveran
195      * @param p1 =max{p2,k}
196      * @param p2 parametro de Stringing
197      * @param phiMin minimo valor de phi
198      * @param phiMax maximo valor de phi
199      * @param g parametro de escala
200      * @param h iteraciones tras las cuales se refrescara el valor de
             alpha y beta
201      * @param nMax numero maximo de iteraciones a ejecutar tras la
             ultima mejora
202      */
203     private static void search(int W[], int q, int p1, int p2, int
            phiMin, int phiMax, double g, int h, int nMax) {
204 
205         //step -1
206         boolean wasUSused = false;
207         double deltaMax = 0;
208         double prevF2;
209 
210         boolean[] prevCapacityFeasible = new boolean[h];
211         prevCapacityFeasible[0] = s.isFactible_capacidad();
212 
213         boolean[] prevLengthFeasible = new boolean[h];
214         prevLengthFeasible[0] = s.isFactible_longitud();
215         int recentChange = 0;
216 
217         //step 0
218         int t = 1;
219         EstructuraTabu tabuStructure = new EstructuraTabu();
220 
221         while (true) {
222             //System.out.println(t);
```

```
223
224             //step 1
225             ArrayList<Integer> randomW = new ArrayList<>(W.length);
226             for (int e : W) {
227                 randomW.add(e);
228             }
229             Collections.shuffle(randomW);
230             while (randomW.size() > q) {
231                 randomW.remove(randomW.size() - 1);
232             }
233
234             SolucionProxi Sbar = null;
235
236             //step 2
237             for (int v : randomW) {
238
239                 int Rr = s.getIdOfRutaContainingNodo(v);
240                 p1 = Math.max(p2, s.getRouteOfId(Rr).getSize());
241
242                 int[] nearestV = Grafo.getNearestNodos(v);
243
244                 HashSet<Integer> nearestRs = new
                       HashSet<>(nearestV.length);
245                 for (int i = 0; i < nearestV.length && i < p1; i++) {
246                     if (nearestV[i] == 0) {
247                         continue;
248                     }
249                     nearestRs.add(
250                             s.getIdOfRutaContainingNodo(nearestV[i]) );
251                 }
252
253                 if (s.getm() < Grafo.getMbar()) {
254                     nearestRs.add(s.getIdOfEmptyRoute());
255                 }
256
257                 SolucionProxi Sprime = new SolucionProxi(s);
258                 Sprime.setNodo(v);
259                 Sprime.setRr(Rr, p1);
260
261                 for (int Rs : nearestRs) {
262                     if (Rs == Rr) {
263                         continue;
264                     }
265
266                     //Step 2 (a)
267                     Sprime.setRs(Rs, p2);
268
269                     //Step 2 (b)
270                     if (tabuStructure.isTabu(v, Rs, t) &&
                           !(Sprime.isFactible() ? Sprime.getF1() < f1star
                           : Sprime.getF2() < f2star)) {
271                         continue;
272                     }
273
274                     //Step 2 (c)
275                     if (Sprime.getF2() < s.getF2()) {
276                         Sprime.setF(Sprime.getF2());
277                     } else {
```

```
278                     Sprime.setF(Sprime.getF2() + deltaMax *
                            Math.sqrt(s.getm()) * g * fv[v] / t);
279                 }
280
281                 if (Sbar == null || Sprime.getF() < Sbar.getF()) {
282                     Sbar = new SolucionProxi(Sprime);
283                 }
284
285             }
286
287         }
288
289         //Step 3
290         prevF2 = s.getF2();
291
292         //Step 4
293         if (Sbar.getF2() > s.getF2() && s.isFactible() &&
                !wasUSused) {
294
295             s.applyUS(p2);
296
297             wasUSused = true;
298         } else {
299             //Step 5
300             tabuStructure.setTabu(Sbar.getNodo(), Sbar.getRr(), t
                    + ThreadLocalRandom.current().nextInt(phiMin,
                    phiMax + 1));
301             fv[Sbar.getNodo()]++;
302
303             Sbar.modificarSolucion();
304             wasUSused = false;
305         }
306
307         if (s.isFactible() && s.getF1() < f1star) {
308             f1star = s.getF1();
309             sstar = s.toArray();
310             //System.out.println("Nueva
                    solución:"+Sstar.toString());
311             recentChange = t;
312         }
313         if (s.getF2() < f2star) {
314             f2star = s.getF2();
315             stildestar = s.toArray();
316             recentChange = t;
317         }
318
319         if (Math.abs(prevF2 - s.getF2()) > deltaMax) {
320             deltaMax = Math.abs(prevF2 - s.getF2());
321         }
322
323         //Step 6
324         prevCapacityFeasible[t % h] = s.isFactible_capacidad();
325         prevLengthFeasible[t % h] = s.isFactible_longitud();
326
327         if ((t + 1) % h == 0) {
328             int i;
329
330             for (i = prevCapacityFeasible.length - 1; i > 0; --i) {
```

```
331                     if (prevCapacityFeasible[i] !=
                            prevCapacityFeasible[i - 1]) {
332                         break;
333                     }
334                 }
335                 if (i <= 0) {
336                     if (prevCapacityFeasible[0]) {
337                         alpha /= 2;
338                     } else {
339                         alpha *= 2;
340                     }
341                 }
342 
343                 for (i = prevLengthFeasible.length - 1; i > 0; --i) {
344                     if (prevLengthFeasible[i] != prevLengthFeasible[i
                            - 1]) {
345                         break;
346                     }
347                 }
348                 if (i <= 0) {
349                     if (prevLengthFeasible[0]) {
350                         beta /= 2;
351                     } else {
352                         beta *= 2;
353                     }
354                 }
355 
356             }
357 
358             //Step 7
359             if (t - recentChange >= nMax) {
360                 break;
361             }
362 
363             t++;
364         }
365 
366     }
367 
368 }
```