Santiago Moronta Martínez
Raúl Romero Díaz
Miguel Ángel Sánchez Arroyo
J. Ángel Velázquez Iturbide

Revisión del Esquema de Programación de los Algoritmos Voraces

Número 2009-03

Serie de Informes Técnicos DLSI1-URJC
ISSN 1988-8074
Departamento de Lenguajes y Sistemas Informáticos I
Universidad Rey Juan Carlos

Índice

1 Introducción ... 1
2 Esquemas Algorítmicos de los Algoritmos Voraces ... 1
2.1 Propuesta de Programación de un Esquema Voraz en Java ... 2
3 Aplicación de los Esquemas de Java a Algoritmos Voraces ... 4
3.1 Desglose de Monedas ... 4
3.1.1 Implementación según el Esquema sin Ordenación Previa ... 5
3.1.2 Implementación según el Esquema sin Ordenación Previa ... 8
3.2 Problema de la Mochila ... 10
3.2.1 Implementación según el Esquema sin Ordenación Previa ... 11
3.2.2 Implementación según el Esquema con Ordenación Previa ... 13
3.3 Planificación de Actividades ... 15
3.3.1 Implementación según el Esquema sin Ordenación Previa ... 16
3.3.2 Implementación según el Esquema con Ordenación Previa ... 19
3.4 Árbol de Recubrimiento de Coste Mínimo ... 20
3.4.1 Algoritmo de Prim ... 20
3.4.2 Algoritmo de Kruskal ... 25
3.5 Caminos más Cortos desde un Solo Origen ... 28
4 Conclusiones ... 32
Agradecimientos ... 32
Referencias ... 32

Revisión del Esquema de Programación de los Algoritmos Voraces

Santiago Moronta Martínez, Raúl Romero Díaz, Miguel Ángel Sánchez Arroyo, J. Ángel Velázquez Iturbide
Departamento de Lenguajes y Sistemas Informáticos I, Universidad Rey Juan Carlos, C/ Tulipán s/n, 28933, Móstoles, Madrid
{s.moronta, r.romero, ma.sancheza}@alumnos.urjc.es, angel.velazquez@urjc.es

Resumen. El presente trabajo revisa el esquema de programación utilizado para representar genéricamente los algoritmos voraces. El trabajo considera dos esquemas alternativos: uno sin ordenación previa de los datos de entrada, y otro con ordenación previa de la entrada y reordenación posterior de la salida. El estudio permite revisar el esquema sin ordenación previa y decidir cuándo es más conveniente utilizar uno u otro esquema.

1 Introducción

Existen multitud de problemas cuyo objetivo es obtener un subconjunto de elementos que satisfaga ciertas condiciones o restricciones, y que a poder ser sean soluciones optimizadas. Para cada uno de estos problemas pueden existir *n* soluciones, y todas ellas sean totalmente válidas y óptimas.

Todos estos problemas no se resuelven de golpe, si no que llevan consigo una ardua tarea de estudio de las distintas posibilidades existentes, es decir, resolviendo el subconjunto de soluciones paso a paso, analizando en cada uno de ellos cual es el elemento más adecuado para formar parte de la solución final.

Los algoritmos voraces siguen en cierta medida lo mencionado anteriormente, pero con la certeza de que aquel elemento que es añadido a la solución es el más óptimo del conjunto de elementos existentes a procesar.

Se debe mencionar que el proceso de obtención de una solución óptima para problemas de tipo voraz, es un proceso lento y costoso, ya que es repetitivo en el sentido de analizar en cada paso cual es el elemento más adecuado para formar parte de la solución.

2 Esquemas Algorítmicos de los Algoritmos Voraces

Para poder abarcar con mayor detalle un esquema común de resolución a los problemas voraces, se han analizado las distintas propuestas facilitadas por autores de varios libros [1, 3, 4]. Sus propuestas se muestran en las figuras siguientes:

```
Función voraz (C:conjunto) : conjunto
   {C es el conjunto de candidatos}
   S← • {construimos la solución en el conjunto S}
   Mientras C • • y no solución(S) hacer
      X ← seleccionar(C)
      C ← C \ {x}
      Si factible (S U {x}) entonces S ← S U {x}
   Si solución (S) entonces devolver S
                   sino devolver <<no hay soluciones>>
```

Fig. 3. Esquema voraz del libro de Brassard y Bratley [1]

```
PROCEDURE Voraz (IN candidatos: {1..N}, OUT solucion: {1..N})
   VAR
      siguiente: 1..N
      solucion := {};
   WHILE (candidatos • {}) ^ ¬EsSolucion(solucion) DO
      siguiente := Seleccionar(candidatos);
      candidatos := candidatos - {siguiente};
      IF EsSolucion(solucion  U {siguiente}) THEN
         solucion := solucion U {siguiente};
```

Fig. 1. Esquema voraz del libro de Galve et al. [3]

```
fun voraz(datos:conjunto) dev S:conjunto
   var
      candidatos:conjunto
      S:= {}    {en S se va construyendo la solución}
      candidatos:= datos

   mientras candidatos • {} ^ ¬solución(S) hacer
      x:= seleccionar(candidatos)
      candidatos:= candidatos - {x}
      si factible(S U {x}) entonces S:= S U {x} fsi
   fmientras
ffun
```

Fig. 2. Esquema voraz del libro de Martí, Ortega y Verdejo [4]

2.1 Propuesta de Programación de un Esquema Voraz en Java

La siguiente propuesta es una variación tomada del esquema genérico presente en el libro Brassard y P. Bratley [1] (véase Fig. 3). El objetivo, ha sido implementar distintos tipos de problemas voraces aplicados al algoritmo genérico, y poder comprobar que su aplicación es válida indistintamente del tipo de problema voraz a resolver. A continuación se detalla el esquema propuesto en este trabajo, tanto para un problema sin ordenación previa de la entrada a tratar, como para un problema con ordenación de la entrada y reordenación de la salida.

```
public  List<Candidato>  algoritmoVorazSinOrdenacion(List<Candidato>
candidatos) {
   inicializarSolucion(solucion);

   while  (! esSolucion(solucion)   &&  (quedanCandidatos(candidatos))
   {
      candidato = seleccionarCandidato(candidatos);
      eliminarCandidato(candidato,candidatos);

      if (esCandidatoFactible(candidato,solucion)) {
         anyadirCandidato(candidato,solucion);
         actualizarCandidatos(candidatos);
      }
   }

   if (esSolucion(solucion)) {
      return solucion;
   } else {
      return null;
   }
}
```

Fig. 5. Implementación de un esquema voraz sin ordenación previa

El esquema propuesto tiene como variación respecto al esquema tomado como base, la actualización de los candidatos cuando es añadido a la solución un nuevo elemento. Esta variación se debe a que en algunos problemas de algoritmos voraces no se dispone inicialmente de todo el conjunto de candidatos, sino que este conjunto se actualiza a medida que se avanza en la resolución del problema.

```
public  List<Candidato>  algoritmoVorazConOrdenacion(List<Candidato>
candidatos) {
   inicializarSolucion(solucion);
   ordenarCandidatos(candidatos);

   while  (! esSolucion(solucion)   &&  (quedanCandidatos(candidatos))
   {
      candidato = seleccionarCandidato(candidatos);
      eliminarCandidato(candidato,candidatos);

      if (esCandidatoFactible(candidato,solucion)) {
         anyadirCandidato(candidato,solucion);
         actualizarCandidatos(candidatos);
      }
   }

   if (esSolucion(solucion)) {
      reordenarCandidatos(solucion);
      return solucion;
   } else {
      return null;
   }
}
```

Fig. 4. Implementación de un esquema voraz con ordenación previa

Para la variante con ordenación de la entrada y reordenación de la salida se han tenido que añadir dos nuevas rutinas que realicen estas tareas ya que en el esquema base no se contempla la resolución de problemas con ordenación.

3 Aplicación de los Esquemas de Java a Algoritmos Voraces

Los problemas que a continuación se mencionan, se han reprogramado siguiendo el esquema algorítmico como propuesta común de solución genérica. Para el siguiente estudio, se ha trabajado con los problemas del cambio de monedas, de la mochila, caminos mínimos desde un origen especificado (algoritmo de Dijkstra), de selección de actividades, y árboles de recubrimiento de coste mínimo (algoritmos de Prim y Kruskal)

Cada una de las soluciones que se listan, son implementaciones específicas por diferentes autores, además de las adaptaciones de los diferentes problemas haciendo uso de los esquemas propuestos.

3.1 Desglose de Monedas

El objetivo del problema de las monedas se centra en desglosar una cantidad en un conjunto de monedas tratando de cumplir alguna condición; en este caso, utilizar el menor número de monedas. Para ello, se parte de un conjunto de tipos de monedas válidas, de las que se supone que hay cantidad (menor) de monedas de los tipos considerados, tales que sumados sus valores equivalgan al importe.

```
TYPE
  MONEDAS  - M500 | M100 | M50 | M25 | M5 | M1,
  VALORES – INTEGER M500..M1,
  CANTIDADES - INTEGER M500..M1,

PROCEDURE Desglose (IN importe : INTEGER, IN valAsig : VALORES, OUT
cambios : CANTIDADES)
  VAR
    modena : MONEDAS

  FOR moneda IN M500..M1 DO
    cambio moneda := 0;
    FOR moneda IN M500..M1 DO
      WHILE valAsig moneda <= importe DO
           cambio moneda := cambio moneda +1;
           importe := importe – valAsig moneda;
```

Fig. 6. Algoritmo voraz para el problema del desglose de monedas [3]

```
Fun monedas (M [0..n] de nat, C:nat) dev sol[0..n] de nat
  sol[0..n]:=[0]
  falta:=C
  i:=n
  mientras falta • 0 ^ i•0 hacer
    sol[i]:=falta div M[i]
    falta:= falta mod M[i]
    i:=i-1
  fmientras
ffun
```

Fig. 7. Algoritmo voraz para el problema del desglose de monedas [4]

3.1.1 Implementación según el Esquema sin Ordenación Previa

El problema del desglose de monedas se ha recodificado según el esquema general propuesto. A continuación se detallan las principales decisiones tomadas, y las posibles variaciones, ya que cada problema tiene características únicas.

```
public List<Moneda> algoritmoVorazSinOrdenacion(List<Moneda>
candidatos, int total) {
  // Crear una copia de los candidatos
  List<Moneda> copiaCandidatos = crearCopiaCandidatos(candidatos);
  // Inicializar solución
  List<Moneda> solucion = new ArrayList<Moneda>();
  // Candidato
  Moneda candidato = null;

  while ((! esSolucion(solucion,total)) &&
        (quedanCandidatos(copiaCandidatos)) ) {
    // Seleccionar candidato
    candidato = seleccionarCandidato(copiaCandidatos);
    // Eliminar candidato
    copiaCandidatos.remove(candidato);

    if (esFactibleCandidato(candidato,solucion,total)) {
      // Añadir candidato a la solución
      anyadirCandidato(candidato,solucion,total);
    }
  }

  if (esSolucion(solucion,total)) {
    return solucion;
  }

  return null;
}
```

Fig. 8. Implementación del algoritmo voraz sin ordenación previa para el problema del desglose de monedas

En la Fig. 8 se puede observar el método principal que representa el algoritmo voraz sin ordenación para el problema del desglose de monedas. La nueva

codificación sigue la estructura presentada como propuesta, donde existe una inicialización previa de cada uno de los componentes; como puede ser el conjunto de candidatos y conjunto solución; y donde posteriormente se inicia el bucle de repetición para la búsqueda de los candidatos válidos.

A continuación se detallan cada uno de los métodos que entran en juego en la resolución del problema.

Método 'esSolución'
Objetivo: Determinar si se ha llegado a la solución final del problema. A partir de la solución parcial, se comprueba si la cantidad existente en la solución parcial es igual a la cantidad total a devolver.
Parámetros: Solución parcial y cantidad total a devolver.

Método 'quedanCandidatos'
Objetivo: Determinar la existencia de candidatos aún sin procesar.
Parámetros: Conjunto de candidatos sin procesar.

```
private Boolean quedanCandidatos(List<Moneda> monedas) {
   return (! monedas.isEmpty());
}
```

Fig. 10. Método 'quedanCandidatos' para el problema del desglose de monedas

Método 'seleccionarCandidato'
Objetivo: Facilitar el candidato más adecuado de todo el conjunto de candidatos posibles. La condición, la moneda de mayor valor respecto al resto.
Parámetros: Conjunto de candidatos sin procesar.

```
private Moneda seleccionarCandidato(List<Moneda> monedas) {
   Moneda mayor = null;
   for (Moneda moneda : monedas) {
        if ((mayor == null) || (mayor.getValor() <
              moneda.getValor())) {
            mayor = moneda;
        }
   }
   return mayor;
}
```

Fig. 11. Método 'seleccionarCandidato' para el problema del desglose de monedas

```
private Boolean esSolucion(List<Moneda> monedas, int total) {
   int cantidadTotal = 0;
   for (Moneda moneda : monedas) {
        cantidadTotal += moneda.getCantidad() * moneda.getValor();
   }
   return (total == cantidadTotal);
}
```

Fig. 9. Método 'esSolución' para el problema del desglose de monedas

Método 'esFactibleCandidato'
Objetivo: Determinar si un candidato es factible. Para ello basta con comprobar que el valor de candidato no supere en ningún caso la diferencia entre la cantidad total a devolver y la cantidad total de la solución parcial.
Parámetros: Candidato a evaluar, conjunto de valores solución y cantidad total a devolver.

```
private Boolean esFactibleCandidato(Moneda candidato, List<Moneda>
solucion, int total) {
   int cantidadTotal = 0;

   for (Moneda moneda : solucion) {
      cantidadTotal += moneda.getCantidad() * moneda.getValor();
   }

   int resto = (total - cantidadTotal);

   return (resto >= candidato.getValor());
}
```

Fig. 12. Método 'esFactibleCandidato' para el problema del desglose de monedas

Método 'añadirCandidato'
Objetivo: Se encarga de agregar el candidato que es factible al conjunto de objetos solución. Además de actualizar la cantidad de monedas de ese tipo que son necesarias devolver.
Parámetros: Candidato a agregar a la solución, conjunto de objetos solución y la cantidad total a devolver.

```
private List<Moneda> anyadirCandidato(Moneda candidato,
List<Moneda> solucion, int total) {
   int cantidadTotal = 0;

   for (Moneda moneda : solucion) {
      cantidadTotal += moneda.getCantidad() * moneda.getValor();
   }

   int resto = (total - cantidadTotal);

   // Calcular el número de monedas
   candidato.setCantidad(resto / candidato.getValor());
   // Añadir el candidato a la solución
   solucion.add(candidato);

   return solucion;
}
```

Fig. 13. Método 'añadirCandidato' para el problema del desglose de monedas

3.1.2 Implementación según el Esquema sin Ordenación Previa

En el apartado anterior se citaba la implementación que resolvía el problema del desglose de monedas siguiendo un esquema voraz genérico sin ordenación. La resolución del mismo problema, pero siguiendo un esquema con ordenación, es similar al anterior, con la diferencia de que al principio se hace una ordenación de los datos de entrada, y una vez que se ha encontrado la solución, reordenarlos a su posición inicial de entrada.

```
public List<Moneda> algoritmoVorazConOrdenacion(List<Moneda>
candidatos, int total) {
  // Inicializar solución
  List<Moneda> solucion = new ArrayList<Moneda>();
  // Ordenar candidatos
  List<Moneda> copiaCandidatos = ordenarCandidatos(candidatos);
  // Candidato
  Moneda candidato = null;

  while ((! esSolucion(solucion,total)) &&
       (quedanCandidatos(copiaCandidatos)) ) {
    // Seleccionar el primer candidato
    candidato = copiaCandidatos.get(0);
    // Eliminar candidato
    copiaCandidatos.remove(candidato);

    if (esFactibleCandidato(candidato,solucion,total)) {
       // Añadir candidato a la solución
       anyadirCandidato(candidato,solucion,total);
    }
  }

  if (esSolucion(solucion,total)) {
    return ordenarSolucion(solucion,candidatos);
  }

  return null;
}
```

Fig. 14. Implementación del algoritmo voraz con ordenación previa para el problema del desglose de monedas

En la Fig. 14, si se compara con el esquema sin ordenación visto anteriormente, las únicas diferencias son la ordenación de los candidatos de entrada en la parte de inicialización, y su reordenación al final del mismo, una vez que se ha dado con la solución final.

La ordenación de los candidatos de entrada en el problema del desglose en monedas, se ha centrado en ordenar cada elemento de mayor a menor valor, ya que el objetivo final del problema es dar como solución el menor número de monedas. La ventaja principal de realizar una ordenación previa del conjunto de elementos de entrada, es que en la búsqueda del candidato válido no es necesaria la búsqueda de aquel elemento con mayor valor, ya que estos se encuentran ordenados.

```java
private List<Moneda> ordenarCandidatos(List<Moneda> candidatos) {
   // Crear una copia de los candidatos ordenados
   List<Moneda> auxiliarCandidatos =
   crearCopiaCandidatos(candidatos);
   List<Moneda> salidaCandidatosOrdenados = new ArrayList<Moneda>();
   Moneda elemento = null;
   int it = 0;
   while (! auxiliarCandidatos.isEmpty()) {
      if ((elemento == null) || (elemento.getValor() <
          auxiliarCandidatos.get(it).getValor())) {
         elemento = auxiliarCandidatos.get(it);
      }
      it++;
      if (it == auxiliarCandidatos.size()) {
         salidaCandidatosOrdenados.add(elemento);
         auxiliarCandidatos.remove(elemento);
         elemento = null;
         it = 0;
      }
   }
   return salidaCandidatosOrdenados;
}
```

Fig. 15. Método para ordenar los datos de entrada del problema del desglose de monedas

Y en la reordenación de la salida, se facilitará la solución en función de la ordenación que traía consigo el conjunto de candidatos de entrada. Para ello, se ha trabajado a lo largo de toda la implementación, con una copia de la entrada, para posteriormente facilitar la reordenación de la solución con el conjunto original de candidatos.

```java
private List<Moneda> ordenarSolucion(List<Moneda> solucion,
List<Moneda> candidatos) {
   // Crear una copia de los candidatos ordenados
   List<Moneda> auxiliarCandidatos =
   crearCopiaCandidatos(candidatos);

   // Recorrer la lista de candidatos en busca del valor de la lista
   // de soluciones
   for (Moneda monedaCandidato : auxiliarCandidatos) {
      for (Moneda monedaSolucion : solucion) {
         if (monedaCandidato.getValor() == monedaSolucion.getValor())
         {
           monedaCandidato.setCantidad(monedaSolucion.getCantidad());
         }
      }
   }
   return auxiliarCandidatos;
}
```

Fig. 16. Método para reordenar los resultados del problema del desglose de monedas

3.2 Problema de la Mochila

Para tener una idea de lo que pretende plantear el problema de la mochila, se pasa a enunciar el objetivo a alcanzar: se tienen n objetos, cada uno con un peso p_i, $1 \leq i \leq n$, y una mochila con una capacidad C. Si se mete una fracción x_i, $0 \leq x_i \leq 1$ del objeto i en la mochila, entonces se consigue un beneficio $b_i\ x_i$. El objetivo es llenar la mochila de manera que se maximice el beneficio total.

La suma de todos los pesos iniciales debe ser mayor que C, porque en caso contrario $x_i = 1$, $1 \leq i \leq n$ es una solución claramente óptima. Además, la suma de los pesos de una solución debe ser igual a al capacidad C de la mochila, porque si fuera menor siempre se podría aumentar dicha suma con cierta cantidad de algún objeto (sumando su consiguiente beneficio) hasta alcanzar una peso total igual a C.

```
Function mochila (w[1..n],v[1..n], W): matriz [1..n]
   {Inicializacion}
   Para i= 1 hasta n hacer x[i] ← 0
   Peso ← 0
   {bucle voraz}
   Mientras peso < W hacer
      i ← el mejor objeto restante { ver más abajo}
      si peso + w[i] • W entonces      x[i] ← 1
         peso ← peso + w[i]
      sino x[i] ← (W-peso)/ w[i]
         peso <-W
   devolver x
```

Fig. 17. Algoritmo voraz para el problema de la mochila [1]

```
PROCEDURE Mochila (IN beneficios : REAL i..N, IN pesos : REAL i..N, IN
capacidad : REAL, OUT objectos : REAL i..N)
   VAR
      resto : REAL,
      i : INTEGER
   FOR i IN 1..N DO
      objetos i := 0.0;
      resto := capacidad;
      i := N;
      LOOP
         IF i = N+1 THEN EXIT;
            IF pesos i <= resto THEN
              objectos i := 1;
              resto := resto – pesos i;
            ELSE
               objetos i := resto / pesos i;
               EXIT;
        i := i+1;
```

Fig. 18. Algoritmo voraz para el problema de la mochila [3]

```
fun mochila (P[1..n], V[1..n] de real, M:real) dev sol[1..n] de real
   sol[1..n] :=[0]
   peso:=0
   mientras peso<M hacer
      i:= el mejor objeto restante
      si peso + P[i] • M entonces
         sol[i]:=1
         peso:= peso + P[i]
      si no
         sol[i]:= (M – peso)/P[i]
         peso:=M
      fsi
   fmientras
ffun
```

Fig. 19. Algoritmo voraz para el problema de la mochila [4]

3.2.1 Implementación según el Esquema sin Ordenación Previa

El algoritmo genérico para el problema de la mochila, consta de un parte de inicialización (candidatos y solución), y otra parte destinada a la búsqueda de los candidatos apropiados para formar parte de la solución final.

En la Fig. 20 se puede apreciar que sigue la estructura general, con una sección de inicialización, y otra de búsqueda de soluciones parciales. La implementación de los métodos internos utilizadas son:

```
public List<Objeto> algoritmoVorazSinOrdenacion(List<Objeto>
candidatos, float capacidad) {
   // Crear una copia de los candidatos
   List<Objeto> copiaCandidatos =
   this.crearCopiaCandidatos(candidatos);
   // Inicializar solución
   List<Objeto> solucion = new ArrayList<Objeto>();
   // Candidato
   Objeto candidato = null;

   while ((! esSolucion(solucion,capacidad)) &&
         (quedanCandidatos(copiaCandidatos)) ) {
      // Seleccionar candidato
      candidato = seleccionarCandidato(copiaCandidatos);
      // Eliminar candidato
      copiaCandidatos.remove(candidato);

      if (esFactibleCandidato(candidato,solucion,capacidad)) {
         // Añadir candidato a la solución
         anyadirCandidato(candidato,solucion,capacidad);
      }
   }

   if (esSolucion(solucion,capacidad)) {
      return solucion;
   }

   return null;
}
```

Fig. 20. Implementación del algoritmo voraz sin ordenación previa para el problema de la mochila

Método ‘esSolución’
Objetivo: Determinar si se ha llegado a la solución final del problema. En el momento que el peso de la solución parcial sea igual al peso máximo de la mochila se habrá llegado a la solución final.
Parámetros: Solución parcial y cantidad total a transportar.

```
private Boolean esSolucion(List<Objeto> solucion, float capacidad) {
   float capacidadTotal = new Float(0.0);
   for (Objeto objeto : solucion) {
      capacidadTotal += objeto.getCantidad() * objeto.getPeso();
   }
   return (capacidad == capacidadTotal);
}
```

Fig. 21. Método 'esSolución' para el problema de la mochila

Método ‘quedanCandidatos’
Objetivo: Determinar la existencia de candidatos aún sin procesar.
Parámetros: Conjunto de candidatos sin procesar.

```
private Boolean quedanCandidatos(List<Objeto> candidatos) {
   return (! candidatos.isEmpty());
}
```

Fig. 22. Método 'quedanCandidatos' para el problema de la mochila

Método’seleccionarCandidato’
Objetivo: Facilitar el candidato más adecuado de todo el conjunto de candidatos posibles. La condición, objeto con mayor relación entre el beneficio y peso.
Parámetros: Conjunto de candidatos sin procesar.

```
private Objeto seleccionarCandidato(List<Objeto> candidatos) {
   Objeto mayor = null;
   for (Objeto objeto : candidatos) {
      if  ((mayor == null) || ((mayor.getBeneficio() /
            mayor.getPeso()) <  (objeto.getBeneficio() /
            objeto.getPeso()))) {
         mayor = objeto;
      }
   }
   return mayor;
}
```

Fig. 23. Método 'seleccionarCandidato' para el problema de la mochila

Método ‘candidatoFactible’
Objetivo: Determinar si un candidato es factible. En el problema de la mochila no es necesaria hacer la comprobación de factibilidad de los candidatos, ya que el tratamiento, en este caso, no se lleva a cabo eliminando de la solución al candidato no válido, si no que es insertado con valor cero, de ahí la no de su implementación. Únicamente se crea un método que devuelve siempre candidato válido.
Parámetros: Candidato a evaluar, conjunto de valores solución y cantidad total a transportar en la mochila.

Método ‘añadirCandidato’
Objetivo: Se encarga de agregar el candidato que es factible al conjunto de objetos solución. En el insertado al conjunto de solución, se actualizan los valores o las cantidades de dicho objeto a ser transportadas en la mochila.
Parámetros: Candidato a agregar a la solución, conjunto de objetos solución y la cantidad total a transportar en la mochila.

```
private List<Objeto> anyadirCandidato(Objeto candidato, List<Objeto>
solucion, float capacidad) {
   float capacidadTotal = new Float(0.0);
   for (Objeto objeto : solucion) {
      capacidadTotal += objeto.getCantidad() * objeto.getPeso();
   }
   float resto = (capacidad - capacidadTotal);
   if (resto >= candidato.getPeso()) {
      candidato.setCantidad(1);
   } else {
      candidato.setCantidad(resto / candidato.getPeso());
   }
   solucion.add(candidato);
   return solucion;
}
```

Fig. 24. Método 'añadirCandidato' para el problema de la mochila

3.2.2 Implementación según el Esquema con Ordenación Previa

El esquema genérico con ordenación para el *Problema de la Mochila* tiene un aspecto similar al del *Desglose de Monedas*. Dispone tanto de un método que permite la ordenación de la entrada de datos, y un método de reordenación de la salida.

```java
public List<Objeto> algoritmoVorazConOrdenacion(List<Objeto>
candidatos, float capacidad) {
   // Inicializar solución
   List<Objeto> solucion = new ArrayList<Objeto>();
   // Ordenar candidatos
   List<Objeto> copiaCandidatos = ordenarCandidatos(candidatos);
   // Candidato
   Objeto candidato = null;
   while ((! esSolucion(solucion,capacidad)) &&
          (quedanCandidatos(copiaCandidatos)) ) {
      // Seleccionar el primer candidato
      candidato = copiaCandidatos.get(0);
      // Eliminar candidato
      copiaCandidatos.remove(candidato);

      if (esFactibleCandidato(candidato,solucion,capacidad)) {
         // Añadir candidato a la solución
         anyadirCandidato(candidato,solucion,capacidad);
      }
   }
   if (esSolucion(solucion,capacidad)) {
      return ordenarSolucion(solucion,candidatos);
   }
   return null;
}
```

Fig. 25. Implementación del algoritmo voraz con ordenación previa para el problema de la mochila

Los elementos de entrada se han ordenado en relación con el beneficio y el peso, obteniendo como conjunto final un listado de elementos ordenados de mayor a menor aporte de valor.

```java
private List<Objeto> ordenarCandidatos(List<Objeto> candidatos) {
   // Crear una copia de los candidatos ordenados
   List<Objeto> auxiliarCandidatos =
   crearCopiaCandidatos(candidatos);
   List<Objeto> salidaCandidatosOrdenados = new ArrayList<Objeto>();
   Objeto elemento = null;
   int it = 0;
   while (! auxiliarCandidatos.isEmpty()) {
      if ((elemento == null) || ((elemento.getBeneficio() /
           elemento.getPeso()) <
           auxiliarCandidatos.get(it).getBeneficio() /
           auxiliarCandidatos.get(it).getPeso()))) {
          elemento = auxiliarCandidatos.get(it);
      }
      it++;
      if (it == auxiliarCandidatos.size()) {
          salidaCandidatosOrdenados.add(elemento);
          auxiliarCandidatos.remove(elemento);
          elemento = null;
          it = 0;
      }
   }
   return salidaCandidatosOrdenados;
}
```

Fig. 26. Método para ordenar los datos de entrada del problema de la mochila

La reordenación de la salida, se facilita en función de la ordenación que traía consigo el conjunto de candidatos de entrada. Para ello, se ha trabajado a lo largo de toda la implementación, con una copia de la entrada, para posteriormente facilitar la reordenación de la solución con el conjunto original de candidatos.

```
private List<Objeto> ordenarSolucion(List<Objeto> solucion,
List<Objeto> candidatos) {
   // Crear una copia de los candidatos ordenados
   List<Objeto> auxiliarCandidatos =
    crearCopiaCandidatos(candidatos);

   // Recorrer la lista de candidatos en busca del valor
   for (Objeto objetoCandidato : auxiliarCandidatos) {
      for (Objeto objetoSolucion : solucion) {
         if ((objetoCandidato.getBeneficio() ==
              objetoSolucion.getBeneficio()) &&
             (objetoCandidato.getPeso() ==
              objetoSolucion.getPeso())) {
            objetoCandidato.setCantidad(objetoSolucion.getCantidad());
         }
      }
   }

   return auxiliarCandidatos;
}
```

Fig. 27. Método para ordenar los resultados del problema de la mochila

3.3 Planificación de Actividades

El problema se centra en planificar la realización del máximo número de actividades posibles. Las premisas a seguir son:

- n actividades a_1, …, a_n que utilizan un recurso compartido durante unos periodos prefijados de antemano.
  - s_i, instante de comienzo de a_i.
  - f_i, instante de finalización de a_i.
- El recurso no puede ser utilizado de forma concurrente.

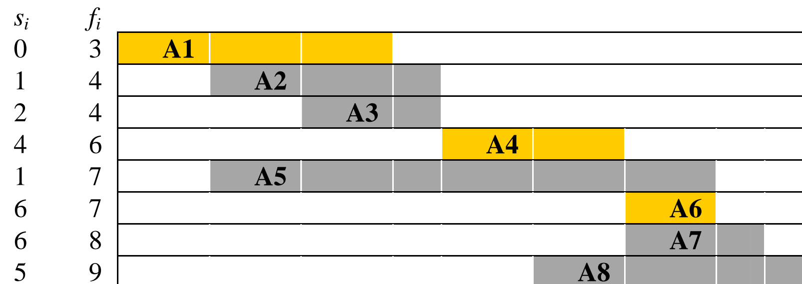

Fig. 28. Ejemplo del problema de planificación de tareas

```
GREEDY-ACTIVITY-SELECTOR (s, f)
  n ← length[s]
  A ← {a1}
  i ← 1
  for m ← 2 to n
    do if sm • fi
      then A← A U {am}
             i ← m
  return A
```

Fig. 29. Algoritmo voraz para el problema de la planificación de tareas [2]

3.3.1 Implementación según el Esquema sin Ordenación Previa

En la Fig. 30 se puede observar la implementación para la planificación de tareas, el cual sigue la misma estructura que en ejemplos anteriores.

```
public List<Actividad> algoritmoVorazSinOrdenacion(List<Actividad>
candidatos) {
  // Crear una copia de los candidatos
  List<Actividad> copiaCandidatos =
  crearCopiaCandidatos(candidatos);
  // Inicializar solución
  List<Actividad> solucion = new ArrayList<Actividad>();
  // Candidato
  Actividad candidato = null;

  while (quedanCandidatos(copiaCandidatos)) {
    // Seleccionar candidato
    candidato = seleccionarCandidato(copiaCandidatos);
    // Eliminar candidato
    copiaCandidatos.remove(candidato);

    if (esFactibleCandidato(candidato,solucion)) {
      // Añadir candidato a la solución
      anyadirCandidato(candidato,solucion);
    }
  }

  if (esSolucion(solucion)) {
    return solucion;
  }

  return null;
}
```

Fig. 30. Implementación del algoritmo voraz sin ordenación previa para el problema de la selección de actividades

Algunos de los métodos carecen de sentido para la planificación de tareas, como por ejemplo, el método que determina si es solución, ya que no existirá ninguna solución antes de terminar de procesar todos y cada uno de los candidatos de entrada. Por tanto, la forma de finalizar la búsqueda, será cuando no existan más candidatos.

Método' quedanCandidatos'
Objetivo: Determinar la existencia de candidatos aún sin procesar.
Parámetros: Conjunto de candidatos sin procesar.

```
private boolean quedanCandidatos(List<Actividad> candidatos) {
  return (! candidatos.isEmpty());
}
```

Fig. 31. Método 'quedanCandidatos' para el problema de la selección de actividades

Método 'seleccionarCandidato'

Objetivo: Facilitar el candidato más adecuado de todo el conjunto de candidatos posibles. La condición, que el tiempo de finalización de la tarea sea menor que la del resto de candidatos.
Parámetros: Conjunto de candidatos sin procesar.

```
private Actividad seleccionarCandidato(List<Actividad> candidatos) {
   // El mejor candidato será aquel cuya tiempo de finalización sea
   // el menor que el resto
   Actividad mejor = null;

   for (Actividad actividad : candidatos) {
      if ((mejor == null) || (mejor.getFin() > actividad.getFin())) {
         mejor = actividad;
      }
   }
   return mejor;
}
```

Fig. 32. Método 'seleccionarCandidato' para el problema de la selección de actividades

Método 'esFactibleCandidato'
Objetivo: Determinar si un candidato es factible. En este caso, un candidato será válido cuando su tiempo de inicio sea mayor o igual que el de resto de elementos contenidos en la solución parcial.
Parámetros: Candidato a evaluar y conjunto de valores de la solución parcial.

```
private boolean esFactibleCandidato(Actividad candidato,
List<Actividad> solucion) {
   for (Actividad actividad : solucion) {
      if (candidato.getInicio() < actividad.getFin()) {
         return false;
      }
   }

   return true;
}
```

Fig. 33. Método 'esFactibleCandidato' para el problema de la selección de actividades

Método 'añadirCandidato'
Objetivo: Se encarga de agregar el candidato que es factible al conjunto de objetos solución.
Parámetros: Candidato a agregar a la solución y conjunto de objetos solución.

```
private void anyadirCandidato(Actividad candidato, List<Actividad>
solucion){
  solucion.add(candidato);
}
```

Fig. 34. Método 'añadirCandidato' para el problema de la selección de actividades

3.3.2 Implementación según el Esquema con Ordenación Previa

La versión con ordenación para la *Planificación de Tareas* no requiere de la necesidad de ordenar la salida, ya que el conjunto que se devuelve como solución sólo incluye las actividades que pueden ser realizadas sin que se solapen las unas con las otras. El resto de actividades son desechadas, por tanto, no se encuentran presentes en la solución.

```
public List<Actividad> algoritmoVorazConOrdenacion(List<Actividad>
candidatos) {
  // Inicializar solución
  List<Actividad> solucion = new ArrayList<Actividad>();
  // Ordenar candidatos
  List<Actividad> copiaCandidatos = ordenarCandidatos(candidatos);
  // Candidato
  Actividad candidato = null;

  while (quedanCandidatos(copiaCandidatos)) {
    // Seleccionar candidato
    candidato = seleccionarCandidato(copiaCandidatos);
    // Eliminar candidato
    copiaCandidatos.remove(candidato);

    if (esFactibleCandidato(candidato,solucion)) {
      // Añadir candidato a la solución
      anyadirCandidato(candidato,solucion);
    }
  }

  if (esSolucion(solucion)) {
    // No es necesaria la ordenación de la solución, ya que se
    // encuentra ordenada
    return solucion;
  }

  return null;
}
```

Fig. 35. Implementación del algoritmo voraz con ordenación previa para el problema de la planificación de tareas

Los elementos de entrada se han ordenado en función de la finalización de la actividad.

```
private List<Actividad> ordenarCandidatos(List<Actividad>
candidatos) {
   // Crear una copia de los candidatos ordenados
   List<Actividad> auxiliarCandidatos =
   crearCopiaCandidatos(candidatos);
   List<Actividad> salidaCandidatosOrdenados =
   new ArrayList<Actividad>();
   Actividad elemento = null;
   int it = 0;

   while (! auxiliarCandidatos.isEmpty()) {
      if ((elemento == null) || (elemento.getFin() >
         auxiliarCandidatos.get(it).getFin())) {
        elemento = auxiliarCandidatos.get(it);
      }
      it++;
      if (it == auxiliarCandidatos.size()) {
         salidaCandidatosOrdenados.add(elemento);
         auxiliarCandidatos.remove(elemento);
         elemento = null;
         it = 0;
      }
   }

   return salidaCandidatosOrdenados;
}
```

Fig. 36. Método para ordenar los datos de entrada del problema de la selección de actividades

3.4 Árbol de Recubrimiento de Coste Mínimo

Los árboles de recubrimiento de coste mínimo tienen como objetivo resolver la conexión de todos los vértices de un grafo dado, donde la suma de los costes de cada uno de éstos sea mínima.

3.4.1 Algoritmo de Prim

La característica principal que acoge el algoritmo de Prim para la resolución de problemas con árboles de recubrimiento de coste mínimo se fundamentan en seleccionar aquellos arcos de menor coste definidos dentro del conjunto de candidatos, y luego elegir el menor desde el nodo en el que se encuentre situado como punto de partida.

Para hacerse una idea del funcionamiento de dicho algoritmo, a continuación se definen los pasos a seguir para la resolución de un algoritmo voraz:

1. Elegir un nodo inicial (cualquiera).
2. Inicializar la solución. Para ello, se debe crear un grafo vacío de coste mínimo e insertar el nodo inicial seleccionado en el paso anterior.
3. Inicializar el conjunto de candidatos. El conjunto de candidatos se encontrará formado por aquellos arcos incidentes al nodo en el que nos encontremos situados.
4. Mientras no se llegue a la solución se irá iterando. En este caso se llegará a la solución cuando el número de nodos del árbol de recubrimiento mínimo (solución) sea mayor o igual al número de nodos del grafo inicial.
   4.1. Extraer el primer candidato y eliminarlo.
   4.2. Comprobar si el candidato es factible. Para conocer si un candidato es factible se procederá a comprobar que éste no pertenece al árbol de recubrimiento mínimo.
      4.2.1. Si no pertenece, actualizar la solución insertando el arco al árbol y poniéndolo en el conjunto de nodos recorridos.
      4.2.2. Actualizar el conjunto de candidatos con los arcos incidentes al nuevo nodo.

```
Función Prim (G= <N,A>:grafo,longitud : A → R) : conjunto de
aristas
  {Iniciacion}
  T ← •
  B ← { un miembro arbitrario de N}
  Mientras B • N hacer
       Buscar e = {u,v} de longitud minima tal que
               u • B y v • N\B
       T ← T U {e}
       B ← B U {v}
  Devolver T
```

Fig. 37. Algoritmo de Prim [2]

El algoritmo de Prim recodificado siguiendo el esquema general es bastante extenso. La Fig. 38 muestra la implementación utilizando el esquema sin ordenación, ya que carece de sentido ordenar el árbol.

```
public Grafo algoritmoVoraz(Grafo grafo) {
   // Crear una copia del grafo
   Grafo copiaGrafo = grafo.getCopiaGrafo();
   // Elegir un nodo inicial cualquiera
   Nodo nodo = copiaGrafo.getNodos().get(0);
   // Definir el número de nodos del grafo
   Integer numeroNodosGrafo = copiaGrafo.getNumeroNodos();
   // Inicializar la solución con el nodo seleccionado (ARM)
   Grafo solucion = new Grafo();
   solucion.addNodo(nodo);
   // Definir el número de nodos del grafo solución
   Integer numeroNodosSolucion = solucion.getNumeroNodos();
   // Lista de nodos procesados
   List<Nodo> nodosSolucion = solucion.getNodos();
   // Inicializar los candidatos, lista de arcos
   List<Arco> candidatos = this.actualizarCandidatos(copiaGrafo,
   nodo, nodosSolucion);

   // Iterar mientras el num de nodos de la solución sea menor que
   // el num de nodos del grafo
   while (! this.esSolucion(numeroNodosSolucion, numeroNodosGrafo)){
      // Extraer el mejor candidado de la lista de candidatos
      Arco candidato = this.seleccionarCandidato(candidatos);

      // Eliminar el arco de la lista de candidatos
      candidatos.remove(candidato);

      // Comprobar si el candidato seleccionado es factible. Si
      //devuelve nulo significará que el arco no es factible
      nodo = this.esFactibleCandidato(candidato, nodosSolucion);

      if (nodo != null) {
         // Actualizar la solución
         solucion.addArco(candidato);
         nodosSolucion.add(nodo);
         numeroNodosSolucion++;

         // Actualizar la lista de candidatos
         this.actualizarCandidatos(copiaGrafo, nodo, nodosSolucion);
      }
   }
   return solucion;
}
```

Fig. 38. Implementación del algoritmo de Prim (sin ordenación previa)

Para poder llevar a cabo la implementación anterior, se ha tenido que crear un tipo de datos, denominado *Grafo*, que este a su vez se encuentra compuesto por *Nodos* y *Arcos*.

```
public class Grafo {
   private List<Arco> grafo;

   public Grafo() {grafo = new ArrayList<Arco>();}

   /**
   * @return the grafo
   */
   public List<Arco> getGrafo() {
      return grafo;
   }

   /**
    * @param grafo the grafo to set
    */
   public void setGrafo(List<Arco> grafo) {
      this.grafo = grafo;
   }
   ......................
```

Fig. 39. Clase 'Grafo'

La clase Grafo contiene todos los métodos que permiten definir el comportamiento de la propia clase. Únicamente se han implementado aquellos métodos necesarios para implementar los ejemplos presentes en el documento. En dicha clase se pueden encontrar métodos destinados a:

- Obtener el grafo.
- Definir el grafo.
- Añadir un arco al grafo.
- Eliminar un arco del grafo.
- Añadir un nodo al grafo.
- Eliminar un nodo del grafo.
- Obtener el número del arcos contenidos en el grafo.
- Obtener el número de nodos contenidos en el grafo.
- Obtener una copia del grafo.
- Obtener el conjunto de arcos del grafo.
- Obtener el conjunto de nodos del grafo.
- Comprobar la existencia de nodos dentro del grafo.
- Comprobar si el grafo es conexo.

Además de la clase *Grafo*, también se han creado los tipos de datos *Arco* y *Nodo*. Tanto la clase *Arco*, como la clase *Nodo* contienen los métodos necesarios, para en este caso, poder trabajar con los algoritmos de Prim y Kruskal (árboles de recubrimiento de coste mínimo), además del algoritmo para la resolución de caminos más cortos desde un solo origen.

```
public class Nodo {

   private String identificador;

   public Nodo() {}

   public Nodo(String identificador) {
      this.identificador = identificador;
   }

   /**
    * @return the identificador
   */
   public String getIdentificador() {
      return identificador;
   }
   ........................
   ........................
}
```

Fig. 41. Clase 'Nodo'

```
public class Arco {
   private Nodo nodoInicial;
   private Nodo nodoFinal;
   private Integer valor;

   public Arco() {}

   public Arco(Nodo nodoInicial, Nodo nodoFinal, Integer valor) {
      this.nodoInicial = nodoInicial;
      this.nodoFinal = nodoFinal;
      this.valor = valor;
   }

   /**
    * @return the nodoInicial
   */
   public Nodo getNodoInicial() {
      return nodoInicial;
   }

   /**
   * @param nodoInicial the nodoInicial to set
   */
   public void setNodoInicial(Nodo nodoInicial) {
      this.nodoInicial = nodoInicial;
   }
}
```

Fig. 40. Clase 'Arco'

Los métodos utilizados en la implementación de la Fig. 38 no han sido implementados para la realización del presente trabajo. A continuación se describen algunos de los métodos más complejos del esquema.

Método 'actualizarCandidatos'
El método de actualización de los candidatos se encarga de devolver la lista de arcos incidentes al nodo que se está procesando. Los candidatos válidos son aquellos cuyos extremos no pertenecen a los nodos existentes en la solución parcial.

Método esFactibleCandidato'
Comprueba que un candidato es factible. Si ambos extremos del candidato pertenecen a la solución parcial, entonces el candidato no es factible.

Método 'seleccionarCandidato'
Obtiene el candidato de menor coste del conjunto de candidatos posibles.

3.4.2 Algoritmo de Kruskal

El algoritmo de Kruskal dispone de la misma característica que el algoritmo de Prim. La única diferencia entre ambos es la forma de resolución, que a continuación se detalla.

1. Inicializar la solución con un grafo vacío.
2. Se obtiene el conjunto de arcos de grafo original.
3. Mientras el número de arcos del árbol de recubrimiento mínimo (solución) sea igual al número de nodos del grafo original menos uno, iterar.
   - 3.1. Se saca el arco con menor coste del conjunto obtenido en el segundo paso (se elimina el arco).
   - 3.2. Si el arco no genera ningún ciclo se añade al árbol.

```
Funcion Kruskal (G= <N,A>:grafo,longitud : A → R) : conjunto de
aristas
   {Inicializacion}
   Ordenar A por longitudes crecientes
   n ← el numero de nodos que hay en N
   T ← • {contendrá las aristas del árbol de recubrimiento minimo}
   Iniciar n conjuntos, cada uno de los cuales contiene un elemento
   disitinto de N
   {bucle voraz}
   Repetir
      e ← {u,v}
      compu ← buscar (u)
      compv ← buscar (v)
      si compu • compv entonces
         fusionar (compu, compv)
         T ← T U {e}
   Hasta que T contenga n-1 aristas
   Devolver T
```

Fig. 42. Algoritmo de Kruskal [1]

```
TYPE
  NODO -> 1..N,
  ARCO -> (NODO, NODO),
  FUNCION_COSTE -> FUNC (ARCO): REAL,
  CONJ_NODOS -> {NODO}
FUNCTION ArbolRecubridorMin ( nodos: LISTA(NODO), arcos:
LISTA(ARCO), fCoste: FUNCION_COSTE: LISTA(ARCO) ->
  nArcos = 0 => [ ]
    arcos == (primerArco, restoArcos) =>
    primerArco == (o, d) =>
      VALUE
        conjO -> ExtraerConj (o, conjsNodos),
        conjD -> ExtraerConj (d, conjsNodos)
      IN
        conjO = ConjD =>
        Kruskal (conjsNodos, restoArcos, nArcos)
        Cons (primerArco, Kruskal(Cons(conjO • ConjD,
                                      Suprimir(conjO,
                                      Suprimir(ConjD,
                                       conjsNodos))),
                               restoArcos, nArcos – 1))
```

Fig. 43. Algoritmo de Kruskal [3]

Al igual que para el algoritmo de Prim, el algoritmo de Kruskal es costoso en extensión de código, pero sencilla en legibilidad y facilidad de escritura. En la Fig. 44 se puede apreciar la codificación del algoritmo según el esquema general propuesto.

```
public Grafo algoritmoVoraz(Grafo grafo) {
  // Comprobar si el grafo a procesar es conexo
  if (grafo.isConexo()) {
    Grafo copiaGrafo = grafo.getCopiaGrafo();
    Integer numeroNodosGrafo = copiaGrafo.getNumeroNodos();
    Integer numeroCandidatos = new Integer(0);
    List<Arco> candidatos = copiaGrafo.getArcos();
    // Inicializar la solución (Árbol de Recubrimiento Mínimo)
    Grafo solucion = new Grafo();
    // Mientras el núm de arcos procesados sea < que el núm de
    // nodos del grafo - 1
    while (! this.esSolucion(numeroCandidatos, numeroNodosGrafo)) {
      // Extraer el mejor candidado de la lista de candidatos
      Arco candidato = this.seleccionarCandidato(candidatos);
      // Eliminar el arco de la lista de candidatos
      candidatos.remove(candidato);
      // Comprobar si el candidato seleccionado es factible.
      solucion.addArco(candidato);
      if (this.esFactibleCandidato(solucion)) {
        numeroCandidatos++;
      } else {
        solucion.removeArco(candidato);
      }
    }
    return solucion;
  } else {
    return null;
  }
}
```

Fig. 44. Implementación del algoritmo de Kruskal (sin ordenación previa)

Para la implementación del algoritmo de Kruskal se han hecho uso de los datos de tipo *Grafo*, *Arco* y *Nodo*. Los métodos utilizados en la implementación de la Fig. 38 no han sido implementados para la realización del presente trabajo. A continuación se describen algunos de los métodos más complejos del esquema.

Método ‘actualizarCandidatos’
En el esquema de Kruskal no tiene cabida un método de actualización de candidatos, ya que inicialmente el conjunto de candidatos ya dispone de todos los elementos del grafo.

Método ‘esFactibleCandidato’
Comprueba que un candidato es factible. Si ambos extremos del candidato pertenecen a la solución parcial, entonces el candidato no es factible, concretando, si al insertar el arco candidato en la solución se produce un ciclo, este es descartado como elemento válido para formar parte de la solución.

Método ‘seleccionarCandidato’
Obtiene el candidato de menor coste del conjunto de candidatos posibles.

3.5 Caminos más Cortos desde un Solo Origen

Sea un grafo dirigido *G*=(*N*, *A*), donde *N* es el conjunto de nodos y *A* es el conjunto de arcos de *G*. Cada arco tiene asociada una longitud (o coste o distancia) no negativa; la longitud de un camino es la suma de las longitudes de los arcos que forman el camino. Cada camino comienza en un nodo, llamado nodo origen, y termina en otro, llamado nodo destino. Si tomamos uno de los nodos del grafo como nodo origen, el problema es determinar la longitud del camino más corto desde el nodo origen a cada uno de los restantes nodos del grafo.

```
Función Dijkstra (L[1..n, 1..n]): matriz [2..n]
  Matriz D[2..n]
  {Iniciacion}
  C← 2, 3,..,n }{S=N\C solo existe implícitamente}
  Para i ← 2 hasta n hacer D[i] ← L[1,i]
  {bucle voraz}
  Repetir n -2 veces
    v ← algún elemento de C que minimiza D[v]
    C ← C \ {v} {e implícitamente S ← S U {v}}
    Para cada w • C hacer
       D[w] ← min(D[w],D[v]+L[v,w])
  Devolver D
```

Fig. 46. Algoritmo de Dijkstra [1]

```
PROCEDURE CaminosMasCortor (IN grafo: REAL1..N,.1..N, IN origen: 1..N,
OUT distancias: REAL1..N) ->
  VAR
    candidatos := {1..N},
    i, menor, nodo: 1..N

  candidatos := {1..N} – {origen};
  distancias_orden := 0.0;
  FOR i IN 1..N DO
    IF i • origen THEN
      distanciasi := grafo_orden;
  FOR i IN 1..N-1 DO
    menor := Extraer(candidatos);
    FOR nodo IN 1..N DO
      IF distancias_nodo < distancias_menor THEN
        menor := nodo;
    candidatos := candidatos – {menor};
    FOR nodo IN 1..N DO
      IF nodo IN candidatos THEN
        distanciasnodo := Min(distancias_nodo, distancias_menor +
                              grafo_menor_nodo)
```

Fig. 45. Algoritmo de Dijkstra [3]

La codificación obtenida para la resolución del problema de los caminos más cortos desde un solo origen sigue, como en los dos ejemplos anteriores, haciendo uso de los tipos de datos *Grafo, Arco* y *Nodo*. Mencionar, que algunos de los métodos utilizados en la resolución tienen un cierto nivel de complejidad, además de contener un número elevado de líneas de código.

En la *Figura 3.5.3* se puede apreciar la implementación del problema siguiendo el esquema general, con su parte de inicialización de componentes, ya sea aplicada al conjunto de candidatos como a la solución final, y la sección de búsqueda de soluciones parciales.

```
public Grafo algoritmoVoraz(Grafo grafo) {

   // Inicialización (referencia al esquema de resolución genérico)
   // Crear una copia del grafo
   Grafo copiaGrafo = grafo.getCopiaGrafo();
   // Elegir un nodo inicial cualquiera
   Nodo nodo = copiaGrafo.getNodos().get(0);
   // Definir el número de nodos del grafo
   Integer numeroNodosGrafo = copiaGrafo.getNumeroNodos();
   // Inicializar la solución a vacío (ARM)
   Grafo solucion = new Grafo();
   // Definir el número de nodos del grafo solución
   Integer numeroNodosSolucion = solucion.getNumeroNodos();
   // Inicializar los candidatos, lista de arcos
   List<Arco> candidatos = this.actualizarCandidatos(copiaGrafo,
   solucion, nodo, nodo, new ArrayList<Arco>());
   // Fin Inicialización

   // Iterar mientras el num de nodos de la solución sea < que el
   // num de nodos del grafo a resolver
   while (! this.esSolucion(numeroNodosSolucion, numeroNodosGrafo)){
      // Extraer el mejor candidado de la lista de candidatos
      Arco candidato = this.seleccionarCandidato(candidatos);
      // Eliminar el arco de la lista de candidatos
      candidatos.remove(candidato);
      // Comprobar si el candidato seleccionado es factible
      if (this.esFactibleCandidato(candidato, solucion)) {
      // Actualizar la solución
         anyadirCandidato(copiaGrafo, solucion, candidato);
         numeroNodosSolucion++;
         // Actualizar la lista de candidatos
         candidatos = this.actualizarCandidatos(copiaGrafo, solucion,
         candidato.getNodoFinal(), nodo, candidatos);
      }
   }
   return solucion;
}
```

Fig. 47. Implementación del algoritmo de Kruskal (sin ordenación previa)

Los métodos más destacables son aquellos que se encuentran relacionados con la actualización de los candidatos en cada iteración y añadir un candidato al conjunto solución. A continuación se listan cada uno de los métodos utilizados, con el fin de

obtener la resolución final al problema. Mencionar que algunos de los métodos presentes en el esquema general no tiene cavidad dentro de la implementación actual, como por ejemplo, el método que se encarga de comprobar al existen de candidatos a procesar.

Método 'esSolución'
Objetivo: Determinar si se ha llegado a la solución final del problema. A partir del número de nodos procesados como parte de la solución, se va comprobando en cada iteración que no se supere el número de nodos del grafo de entrada.
Parámetros: Solución parcial y cantidad total a transportar.

```
private boolean esSolucion(Integer numeroNodosSolucion, Integer
                           numeroNodosGrafo) {
   return (numeroNodosSolucion >= numeroNodosGrafo - 1);
}
```

Fig. 48. Método 'esSolución' para el algoritmo de Kruskal

Método 'actualizarCandidatos'
Objetivo: Actualizar el conjunto de candidatos disponibles en función del arco en el que nos encontremos situados. Se trata del método más complicado del problema. Para cumplir con el objetivo, el método se encarga de obtener todos los arcos con las distancias actualizadas desde el nodo origen haciendo *backtracking*.
Parámetros: Grafo inicial, solución parcial, nodo actual, nodo inicial y conjunto de candidatos.

Por razones de espacio, el código relativo a la actualización de los candidatos no se ha incluido. Éste puede ser consultado en el código fuente que se adjunta dentro de la clase que implementa el algoritmo de Dijkstra.

Método 'seleccionarCandidato'
Objetivo: Facilitar el candidato más adecuado de todo el conjunto de candidatos posibles. La condición de selección, que el tiempo de finalización de la tarea sea menor que la del resto de candidatos.
Parámetros: Conjunto de candidatos sin procesar.

```java
private Actividad seleccionarCandidato(List<Actividad> candidatos) {
   Actividad mejor = null;

   for (Actividad actividad : candidatos) {
      if ((mejor == null) || (mejor.getFin() > actividad.getFin())) {
         mejor = actividad;
      }
   }
   return mejor;
}
```

Fig. 49. Método 'seleccionarCandidato' para el algoritmo de Kruskal

Método 'esFactibleCandidato'
Objetivo: Determinar si un candidato es factible. En este caso, un candidato será válido cuando su tiempo de inicio sea mayor o igual que el de resto de elementos contenidos en la solución parcial.
Parámetros: Candidato a evaluar y conjunto de valores de la solución parcial.

```java
private boolean esFactibleCandidato(Actividad candidato,
                                    List<Actividad> solucion) {
   for (Actividad actividad : solucion) {
      if (candidato.getInicio() < actividad.getFin()) {
         return false;
      }
   }
   return true;
}
```

Fig. 50. Método 'esFactibleCandidato' para el algoritmo de Kruskal

Método' añadirCandidato'
Objetivo: Se encarga de agregar el candidato que es factible al conjunto de objetos solución.
Parámetros: Candidato a agregar a la solución y conjunto de objetos solución.

```java
private void anyadirCandidato(Actividad candidato,
                              List<Actividad> solucion){
   solucion.add(candidato);
}
```

Fig. 51. Método 'añadirCandidato' para el algoritmo de Kruskal

4 Conclusiones

Podemos destacar varias conclusiones del trabajo realizado. En primer lugar, con el esquema sin ordenación propuesto se han podido resolver los seis problemas tratados. Nuestro esquema difiere del propuesto en libros de algoritmos en que el conjunto de candidatos no es fijo, sino que se actualiza en cada iteración del algoritmo.

En segundo lugar, hemos estudiado cuándo puede usarse o conviene usar los esquemas sin o con ordenación previa. Nuestra conclusión es que la ordenación previa puede hacerse cuando los candidatos son fijos y conocidos a priori. Es el caso de los problemas de cambio de moneda, mochila y planificación de actividades. Sin embargo, debe usarse el esquema sin ordenación previa cuando el conjunto de candidatos varía dinámicamente. Es el caso de los problemas del árbol de recubrimiento de coste mínimo y los caminos mínimos desde un solo origen.

Los esquemas generales propuestos se han implementado en Java. El estilo de programación utilizado conduce a algoritmos muy largos. Por tanto, deben entenderse como esquemas de comportamiento (es decir, semánticos), más que de código (es decir, sintácticos).

Agradecimientos

Este trabajo se ha realizado dentro del “Seminario de Software Avanzado para Informática Educativa”, correspondiente al Máster Oficial en Tecnologías de la Información y Sistemas Informáticos. El trabajo se ha financiado parcialmente por el proyecto TIN2008-04301 del Ministerio de Innovación y Ciencia.

Referencias

1. G. Brassard y P. Bratley, Fundamentos de algoritmia, Prentice-Hall, 1997.
2. T. H. Cormen, C. E. Leiserson y R. L. Rivest, Introduction to Algorithms, The MIT Press, 2ª ed., 2001.
3. J. Galve, J.C. González, Á. Sánchez y Á. Velázquez, Algorítmica: diseño y análisis de algoritmos funcionales e imperativos, Ra-Ma, 1993.
4. N. Martí Oliet, Y. Ortega Mallén y J. A. Verdejo López, Estructuras de datos y métodos algorítmicos – Ejercicios resueltos, Pearson, 2004.