Universidad de la Laguna
ETS Ingeniería Informatica

Non-guillotine 2D bin packing problem

EDUARDO PEREZ MEDEROS

MIGUEL MONTERREY VARELA

JAIME GONZALEZ VALDES

ÓSCAR MATEOS LÓPEZ

HEURISTICAS 3º

INFORME DEL PROYECTO

CURSO 2010/2011

13 DE ENERO DE 2011

1. Introducción

Nuestro grupo eligió el problema del empaquetado en cajas en 2 dimensiones sin guillotina y sin rotación, *non-guillotine two-dimensional bin packing problem*. Durante el informe dedicaremos varios apartados a describir el problema (descripción del problema, función objetivo,…), realizaremos una revisión bibliográfica, describiremos los procedimientos implementados y la experiencia computacional obtenida y por último enumeraremos las conclusiones obtenidas.

2. Definición del problema

Dado un conjunto de n *ítems bidimensionales* (rectángulos), {i1, i2, ..., in}, de dimensiones respectivas $(h(i_1), w(i_1))$, $(h(i_2), w(i_2))$, ..., $(h(i_n), w(i_n))$, determinar el menor número de bins (rectángulos) de dimensiones (H, W) necesarios para empaquetar sin solapamientos todos los ítems. Se supondrá que $h(i_j) \leqslant H$, $w(i_j) \leqslant W$, $\forall j = 1, ..., n$ y que los ítems no pueden rotarse. El empaquetado resultante no necesariamente debe poder obtenerse con cortes tipo guillotina. Existen muchas variaciones de este problema. Es considerado como un problema NP- hard debido a que el número de soluciones posibles, entre las que se encuentra indistinguiblemente la óptima, está relacionado con el número de permutaciones de objetos a "*empaquetar*." Buscamos entonces distintos algorítmos heurísticos que resuelven con un grado aceptable de eficacia cuál es el empaquetado de objetos de menor extensión espacial en un tiempo de computación razonable.

2.1 Función Objetivo

Para el cálculo de la función objetivo debemos conocer primero:

- Las dimensiones de la caja (todas tienen la misma medida).
- Las dimensiones de los distintos rectángulos.
- Coordenadas de los distintos rectángulos a lo largo y ancho de la caja cuando estemos pasando las diferentes heurísticas.
- Espacio en una caja no vacía.

Con estos datos tenemos suficiente información para lograr nuestro objetivo. Minimizar el número de cajas frente a un número de rectángulos. Usando el algoritmo ***Finite First Fit***, vamos colocando cada rectángulo, según el orden establecido por la permutación, en las posiciones disponibles intentando minimizar el espacio desperdiciado.

2.2 Solución

Representamos las soluciones de este problema mediante una permutación que indica el orden en que los ítems son considerados para su inclusión en el objeto, junto a un procedimiento que indica la posición que ocupan los ítems en el objeto.

2.3 Heurísticas

Mediante las distintas heurísticas intentamos minimizar la función objetivo del problema. Se prefiere abordar, por medio de heurísticas, una representación más ajustada del problema planteado que una versión menos realista de tal problema que pueda resolverse de forma exacta. Hemos utilizado distintos tipos de heurísticas, las cuales explicaremos mas adelante.

Métodos constructivos
G.R.A.S.P.
Métodos de búsqueda
Búsquedas Locales.
Búsquedas Aleatoria Pura.
Búsquedas por Recorrido al Azar.
Búsquedas Multiarranque.
Recocido Simulado.
Búsqueda Tabú.
VND.
BVNS.
Búsqueda Dispersa.

3. Diseño e implementación del problema

A continuación describiremos las distintas clases implementadas.

3.1 Problema

Clase con los datos necesarios para representar un problema de Bin Packing. Lee de ficheros cada rectángulo, y las medidas de la caja.

3.1.1 Información generada por javadocs

app
Class Problema

java.lang.Object **app.Problema**

---

public class **Problema**
extends java.lang.Object
Clase con los datos necesarios para representar un problema de Bin Packing.
Version:
2.0, 0.1
Author:
Eduardo Perez Mederos, Miguel Monterrey Varela, Jaime Gonzalez Valdes, Oscar Mateos Lopez

---

Field Summary	
private int	**AltoCaja** Alto de la Caja H.
private int	**AnchoCaja** Ancho de la caja W.
private java.util.ArrayList<Rectangulo>	**Rec** Array con todos los rectangulos del problema.

Constructor Summary
Problema(java.lang.String fileName) Constructor de la clase Problema.

Method Summary	
int	**getAltoCaja**() Metodo que devuelve el alto de la caja.
int	**getAnchoCaja**() Metodo que devuelve el ancho de la caja.
java.util.ArrayList<Rectangulo>	**getRectangulos**() Metodo que devuelve los Retangulos del problema.
java.util.ArrayList<Rectangulo>	**readFile**(java.lang.String fileName) Metodo que devuelve un array de Retangulos despues de leerlos de un fichero.
java.lang.String	**toString**() Metodo que resume en una cadena de caracteres el problema.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Field Detail

Rec
private java.util.ArrayList<Rectangulo> **Rec**
Array con todos los rectangulos del problema.

AltoCaja
private int **AltoCaja**
Alto de la Caja, H.

AnchoCaja
private int **AnchoCaja**
Ancho de la caja, W.

Constructor Detail

Problema
public **Problema**(java.lang.String fileName)
Constructor de la clase Problema.
Parameters:
fileName - Nombre del archivo.

Method Detail

getRectangulos
public java.util.ArrayList<Rectangulo> **getRectangulos**()
Metodo que devuelve los Retangulos del problema.
Returns:
ArrayList - Array de rectangulos.

---

getAltoCaja
public int **getAltoCaja**()
Metodo que devuelve el alto de la caja.
Returns:
AltoCaja - El alto de la caja.

---

getAnchoCaja
public int **getAnchoCaja**()
Metodo que devuelve el ancho de la caja.
Returns:
AnchoCaja - El ancho de la caja.

---

readFile
public java.util.ArrayList<Rectangulo> **readFile**(java.lang.String fileName)
Metodo que devuelve un array de Retangulos despues de leerlos de un fichero.
Parameters:
fileName - Nombre del archivo.
Returns:
ArrayList - Array de rectangulos.

---

toString
public java.lang.String **toString**()
Metodo que resume en una cadena de caracteres el problema.
Overrides:
toString in class java.lang.Object
Returns:
String – Problema.

3.2 Solución

La Solución es la que contiene con los datos necesarios para representar una solución de Bin Packing.

Utilizaremos una estrategia aleatoria de cara a conseguir una solución inicial valida. Cuanto mejor sea la solución inicial. Menor será la cantidad de iteraciones necesarias durante una heurística.

Existen implementadas 3 formas de para obtener una solución inicial (permutación), determinista, aleatoria y mixta. Se encuentra también implementado ***Finite First Fit***, algoritmo de colocación inicial, que detallaremos en el siguiente punto.

Por ultimo tenemos los métodos que calculan la función objetivo, y muestran la solución con su valor, su permutación y el área total ocupada.

3.2.1 Finite First Fit

Empaqueta el actual ítem en el estante más bajo del primer bin en el que quepa. Si no cabe en ningún estante, crear un estante en el primer bin disponible o inicializar un nuevo bin.

3.2.2 Información generada por javadocs

app
Class Solucion

java.lang.Object **app.Solucion**
All Implemented Interfaces:
java.lang.Comparable<Solucion>

---

public class **Solucion**
extends java.lang.Object
implements java.lang.Comparable<Solucion>
Clase con los datos necesarios para representar una solucion de Bin Packing.
Since:
0.1
Version:
2.0
Author:
Eduardo Perez Mederos, Miguel Monterrey Varela, Jaime Gonzalez Valdes, Oscar Mateos Lopez

---

Field Summary	
static int	**ALEATORIA** Constantes para indicar el tipo de generacion de solucion inicial.
private int	**AreaOcupada**

	Sumatorio del area ocupada por todos los rectangulos de todas las cajas.
private java.util.ArrayList<Caja>	**Cajas** Array de Cajas.
static int	**DETERMINISTA**
static int	**FINITE** Constantes para indicar el algoritmo de colocacion.
static int	**GRASP**
static int	**MIXTA**
private int	**Objetivo** Valor de la funcion objetivo.
private java.util.ArrayList<java.lang.Integer>	**PermuOcupada**
private int[]	**Permutacion** Solucion del problema.

Constructor Summary
Solucion() Constructor dado el numero de rectangulos.
Solucion(int[] permutacion, java.util.ArrayList<Rectangulo> rec, int altoCaja, int anchoCaja, int colocacion) Constructor dado el tipo de generacion de solucion inicial y el numero de rectangulos.
Solucion(int permType, java.util.ArrayList<Rectangulo> rec, int altoCaja, int anchoCaja, int colocacion) Constructor dado el tipo de generacion de solucion inicial y el numero de rectangulos.

Method Summary	
private int	**AreaTotal**() Calculo de el area total usada.
int	**compareTo**(Solucion s) Metodo que compara dos soluciones y determina cual es el mayor en funcion de su valor de la F.
private void	**FiniteFirstFit**(int altoCaja, int anchoCaja, java.util.ArrayList<Rectangulo> rec) Algoritmo de colocacion inicial de los rectangulos en una caja

	tras obtener la permutacion inicial.
int	**getAreaTotalMU**() Calculo de el area total usada menos la ultima caja.
int	**getFObjetivo**() Metodo que devuelve el valor de la funcion objetivo de la solucion.
int[]	**getPermutacion**() Metodo que devuelve la permutacion de la solucion.
private void	**Grasp**(int altoCaja, int anchoCaja, java.util.ArrayList<Rectangulo> rec) Algoritmo de colocacion para el metodo Grasp.
private void	**permAleatoria**(int size) Metodo que se encarga de la inicializacion de una solucion.
private void	**permDeterminista**(int size) Metodo que se encarga de la inicializacion de una solucion.
private void	**permMixta**(int size) Metodo que se encarga de la inicializacion de una solucion.
private void	**swap**(int[] array, int posOne, int posTwo) Metodo que se encarga de intercambiar elementos para las inicializaciones mixtas.
java.lang.String	**toString**() Metodo para resumir la informacion de una solucion en una cadena de caracteres.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Field Detail

ALEATORIA
public static final int **ALEATORIA**

Constantes para indicar el tipo de generacion de solucion inicial.

See Also:
Constant Field Values

DETERMINISTA
public static final int **DETERMINISTA**

See Also:
Constant Field Values

MIXTA
public static final int **MIXTA**

See Also:
Constant Field Values

FINITE
public static final int **FINITE**
Constantes para indicar el algoritmo de colocacion.
See Also:
Constant Field Values

GRASP
public static final int **GRASP**
See Also:
Constant Field Values

Cajas
private java.util.ArrayList<Caja> **Cajas**
Array de Cajas.

Permutacion
private int[] **Permutacion**
Solucion del problema.

PermuOcupada
private java.util.ArrayList<java.lang.Integer> **PermuOcupada**

Objetivo
private int **Objetivo**
Valor de la funcion objetivo.

AreaOcupada
private int **AreaOcupada**
Sumatorio del area ocupada por todos los rectangulos de todas las cajas.

Constructor Detail

Solucion
public **Solucion**()
Constructor dado el numero de rectangulos.

Solucion
public **Solucion**(int permType, java.util.ArrayList<Rectangulo> rec, int altoCaja, int anchoCaja, int colocacion)
Constructor dado el tipo de generacion de solucion inicial y el numero de rectangulos.

Parameters:
permType - Tipo de permutacion de datos.
rec - Array de rectangulos.
altoCaja - Tamano del alto de la caja.
anchoCaja - Tamano del ancho de la caja.

Solucion
public **Solucion**(int[] permutacion, java.util.ArrayList<Rectangulo> rec, int altoCaja, int anchoCaja, int colocacion)
Constructor dado el tipo de generacion de solucion inicial y el numero de rectangulos.
Parameters:
rec - Array de rectangulos.
altoCaja - Tamano del alto de la caja.
anchoCaja - Tamano del ancho de la caja.
colocacion - Tipo de algoritmo de colocacion.

Method Detail

permAleatoria
private void **permAleatoria**(int size)
Metodo que se encarga de la inicializacion de una solucion. Para un mismo conjunto de N rectangulos, crea una permutacion aleatoria.
Parameters:
size - Numero de rectangulos.

---

permDeterminista
private void **permDeterminista**(int size)
Metodo que se encarga de la inicializacion de una solucion. Para un mismo conjunto de N rectangulos, crea una permutacion ordenada en funcion de su area (de mayor a menor).
Parameters:
size - Numero de rectangulos.

---

permMixta
private void **permMixta**(int size)
Metodo que se encarga de la inicializacion de una solucion. Para un mismo conjunto de N rectangulos, crea una permutacion mixta, resultante de aplicar un metodo determinista y un metodo aleatorio.
Parameters:
size - Numero de rectangulos.

---

swap
private void **swap**(int[] array, int posOne, int posTwo)
Metodo que se encarga de intercambiar elementos para las inicializaciones mixtas.
Parameters:
array - Array que contiene la permutacion.
posOne - Posicion del primer elemento a intercambiar.

posTwo - Posicion del segundo elemento a intercambiar.

---

getPermutacion
public int[] **getPermutacion**()
Metodo que devuelve la permutacion de la solucion.
Returns:
Permutacion

---

getFObjetivo
public int **getFObjetivo**()
Metodo que devuelve el valor de la funcion objetivo de la solucion.
Returns:
Objetivo

AreaTotal
private int **AreaTotal**()
Calculo de el area total usada. Sumatorio del area total de cada caja.
Returns:
areaTotal - Area total usada en todas las cajas.

---

getAreaTotalMU
public int **getAreaTotalMU**()
Calculo de el area total usada menos la ultima caja. Sumatorio del area total de cada caja.
Returns:
areaTotal - Area total usada en todas las cajas menos la ultima.

FiniteFirstFit
private void **FiniteFirstFit**(int altoCaja, int anchoCaja, java.util.ArrayList<Rectangulo> rec)
Algoritmo de colocacion inicial de los rectangulos en una caja tras obtener la permutacion inicial.
Parameters:
altoCaja - Alto de las cajas.
anchoCaja - Ancho de las cajas.
rec - Array de rectangulos a introducir.

Grasp
private void **Grasp**(int altoCaja, int anchoCaja, java.util.ArrayList<Rectangulo> rec)
Algoritmo de colocacion para el metodo Grasp. El cual elije al azar uno de los tres mejores puntos disponibles para el rectangulo a insertar.
Parameters:
altoCaja - Alto de las cajas.
anchoCaja - Ancho de las cajas.
rec - Array de rectangulos a introducir.

---

compareTo
public int **compareTo**(Solucion s)

Metodo que compara dos soluciones y determina cual es el mayor en funcion de su valor de la F. Objetivo y su area total ocupada.

Specified by:
compareTo in interface java.lang.Comparable<Solucion>

Parameters:
s - Solucion a comparar.

Returns:
1, 0 o -1 en funcion de si la solucion es mayor, igual o menor la solucion comparada.

---

toString
public java.lang.String **toString**()

Metodo para resumir la informacion de una solucion en una cadena de caracteres.

Overrides:
toString in class java.lang.Object

Returns:
String - Solucion.

3.3 Caja

La caja (*bin*) es el contenedor donde guardaremos los distintos rectángulos. Esta clase dispondrá de las operaciones básicas de configuración (establecer tamaño de la caja, área, etc.). También dispondrá de herramientas para conocer si el rectángulo cabe en la caja, inserción de un nuevo rectángulo en la caja, etc.

3.3.1 Información generada por javadocs

app
Class Caja

java.lang.Object **app.Caja**

---

public class **Caja**
extends java.lang.Object
Clase con los datos necesarios para representar un contenedor.

Since:
0.1

Version:
2.0

Author:

Eduardo Perez Mederos, Miguel Monterrey Varela, Jaime Gonzalez Valdes, Oscar Mateos Lopez

---

Field Summary	
private int	**alto** Alto de la caja.
private int	**ancho** Ancho de la caja.
private int	**area** Area de la caja.
private int	**Nu**
private java.util.ArrayList<PuntoCota>	**PC** Punto libres y sus respectivas cotas.
private java.util.ArrayList<Rectangulo>	**RecIn**

Constructor Summary
Caja(int alto, int ancho) Constructor para la clase Caja.

Method Summary	
void	**AddPuntosLibres**(Punto puntoRec, int cota, int altoRec, int anchoRec) Anade los puntos libres que se generan al anadir un rectangulo en el contenedor.
Rectangulo[]	**CabeAlgunRectangulo**(Rectangulo r) Metodo que devuelve un array de rectangulos, en el cual almacenamos las 3 mejores posibilidades de insercion para un rectangulo r que pasamos por parametro.
boolean	**CabeRectangulo**(Rectangulo r) Metodo que comprueba si cabe un rectangulo dentro de la caja.
int	**getAncho**() Metodo que devuelve el ancho de la caja.
int	**getArea**() Metodo que devuelve el area de la caja.
int	**getAreaRestante**() Metodo que devuelve el area restante de la caja.
int	**getNu**()

	Metodo que devuelve el valor Nu.
void	**NuevoRectangulo**(Rectangulo r) Introduce un rectangulo en el contenedor y genera los puntos libres.
private int	**PosicionInsertar**(Rectangulo[] R) Metodo que devuelve una posicion para insertar.
void	**setNu**(int i) Metodo que modifica el valor Nu.
java.lang.String	**toString**() Salida del contenido de la caja .
private int	**ValorMax**(Rectangulo[] R) Metodo que devuelve dentro de un array de rectangulos, el valor del rectangulo con mayor desperdicio de espacio.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Field Detail

alto
private int **alto**
Alto de la caja.

---

ancho
private int **ancho**
Ancho de la caja.

---

area
private int **area**
Area de la caja.

Nu
private int **Nu**

PC
private java.util.ArrayList<PuntoCota> **PC**
Punto libres y sus respectivas cotas.

RecIn
private java.util.ArrayList<Rectangulo> **RecIn**

Constructor Detail

Caja
public **Caja**(int alto, int ancho)
Constructor para la clase Caja.
Parameters:
alto - Alto de la caja.
ancho - Ancho de la caja.

Method Detail

getAncho
public int **getAncho**()
Metodo que devuelve el ancho de la caja.
Returns:
ancho - Ancho de la caja.

---

getNu
public int **getNu**()
Metodo que devuelve el valor Nu. Utilizado en el metodo Grasp.
Returns:
Nu

---

setNu
public void **setNu**(int i)
Metodo que modifica el valor Nu.
Parameters:
i -

---

getArea
public int **getArea**()
Metodo que devuelve el area de la caja.
Returns:
area - Area de la caja.

---

getAreaRestante
public int **getAreaRestante**()
Metodo que devuelve el area restante de la caja.
Returns:
aux - Area restante.

CabeRectangulo
public boolean **CabeRectangulo**(Rectangulo r)
Metodo que comprueba si cabe un rectangulo dentro de la caja.
Parameters:
r - Rectangulo a introducir en la caja.
Returns:
boolean

CabeAlgunRectangulo
public Rectangulo[] **CabeAlgunRectangulo**(Rectangulo r)

Metodo que devuelve un array de rectangulos, en el cual almacenamos las 3 mejores posibilidades de insercion para un rectangulo r que pasamos por parametro.
Parameters:
r - Rectangulo a introducir en la caja.
Returns:
Rectangulo[]

PosicionInsertar
private int **PosicionInsertar**(Rectangulo[] R)

Metodo que devuelve una posicion para insertar. Usado en el metodo GRASP.
Parameters:
R - Array de candidatos a insertar.
Returns:
int

ValorMax
private int **ValorMax**(Rectangulo[] R)

Metodo que devuelve dentro de un array de rectangulos, el valor del rectangulo con mayor desperdicio de espacio. Candidato a salir del array.
Parameters:
R - Array de candidatos a insertar.
Returns:
int

AddPuntosLibres
public void **AddPuntosLibres**(Punto puntoRec, int cota, int altoRec, int anchoRec)

Anade los puntos libres que se generan al anadir un rectangulo en el contenedor.
Parameters:
puntoRec - Punto donde se va a colocar el rectangulo.
cota - Altura maxima del punto.
altoRec - Alto del rectangulo.
anchoRec - Ancho del rectangulo.

NuevoRectangulo
public void **NuevoRectangulo**(Rectangulo r)

Introduce un rectangulo en el contenedor y genera los puntos libres.
Parameters:
r - Nuevo rectangulo a anadir.

---

toString
public java.lang.String **toString**()
Salida del contenido de la caja .
Overrides:
toString in class java.lang.Object
Returns:
String - Cadena de caracteres que muestra cada caja y los rectangulos introducidos en cada una.

3.4 Rectángulo

Es la clase con los datos necesarios para representar un rectángulo. Dispone de las herramientas de edición básica de los rectángulos, (punto en la caja, alto y ancho, comparar dos rectángulos, etc.).

3.4.1 Información generada por javadocs

app
Class Rectangulo

java.lang.Object **app.Rectangulo**
All Implemented Interfaces:
java.lang.Comparable<Rectangulo>

---

public class **Rectangulo**
extends java.lang.Object
implements java.lang.Comparable<Rectangulo>
Clase con los datos necesarios para representar un rectangulo.
Since:
0.1
Version:
2.0
Author:
Eduardo Perez Mederos, Miguel Monterrey Varela, Jaime Gonzalez Valdes, Oscar Mateos Lopez

---

Field Summary	
private int	**alto** Alto del rectangulo.
private int	**ancho** Ancho del rectangulo.

private int	**area** Area del rectangulo.
private Punto	**pos** Posicion que ocupa en la caja.

Constructor Summary
Rectangulo() Constructor de la clase Rectangulo.
Rectangulo(int alto, int ancho) Constructor de la clase Rectangulo.

Method Summary	
int	**compareTo**(Rectangulo r) Metodo que compara dos rectangulos y determina cual es el mayor en funcion de su altura.
int	**getAlto**() Metodo para obtener el alto de un rectangulo.
int	**getAncho**() Metodo para obtener el ancho de un rectangulo.
int	**getArea**() Metodo para obtener el area de un rectangulo.
Punto	**getPos**() Metodo para obtener la posicion de un rectangulo.
void	**setAlto**(int alto) Metodo para asignar el alto a un rectangulo.
void	**setAncho**(int ancho) Metodo para asignar el ancho a un rectangulo.
void	**setPos**(Punto pos) Metodo para asignar la posicion a un rectangulo.
java.lang.String	**toString**() Metodo para resumir la informacion de un rectangulo en una cadena de caracteres.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Field Detail

alto
private int **alto**
Alto del rectangulo.

---

ancho
private int **ancho**
Ancho del rectangulo.

---

area
private int **area**
Area del rectangulo.

---

pos
private Punto **pos**
Posicion que ocupa en la caja.

Constructor Detail

Rectangulo
public **Rectangulo**()
Constructor de la clase Rectangulo. Crea un rectangulo dado su alto y su ancho.

Rectangulo
public **Rectangulo**(int alto, int ancho)
Constructor de la clase Rectangulo. Crea un rectangulo dado su alto y su ancho.
Parameters:
alto - Alto del rectangulo.
ancho - Ancho del rectangulo.

Method Detail

compareTo
public int **compareTo**(Rectangulo r)
Metodo que compara dos rectangulos y determina cual es el mayor en funcion de su altura.
Specified by:
compareTo in interface java.lang.Comparable<Rectangulo>
Parameters:
r - Rectangulo a comparar.
Returns:
1, 0 o -1 en funcion de si el area del rectangulo comparador es mayor, igual o menor que el area del rectangulo comparado.

---

getAlto
public int **getAlto**()
Metodo para obtener el alto de un rectangulo.
Returns:
Alto del rectangulo.

---

setAlto
public void **setAlto**(int alto)
Metodo para asignar el alto a un rectangulo.
Parameters:
alto - Nuevo alto del rectangulo.

---

getAncho
public int **getAncho**()
Metodo para obtener el ancho de un rectangulo.
Returns:
Ancho del rectangulo.

---

setAncho
public void **setAncho**(int ancho)
Metodo para asignar el ancho a un rectangulo.
Parameters:
ancho - Nuevo ancho del rectangulo.

---

getArea
public int **getArea**()
Metodo para obtener el area de un rectangulo.
Returns:
Area del rectangulo.

---

getPos
public Punto **getPos**()
Metodo para obtener la posicion de un rectangulo.
Returns:
Posicion del rectangulo.

---

setPos
public void **setPos**(Punto pos)
Metodo para asignar la posicion a un rectangulo.
Parameters:
pos - Nueva posicion del rectangulo.

---

toString
public java.lang.String **toString**()
Metodo para resumir la informacion de un rectangulo en una cadena de caracteres.
Overrides:
toString in class java.lang.Object
Returns:
String - Cadena de caracteres con la informacion.

3.5 Punto

Clase con los datos necesarios para representar un punto (x, y).

3.5.1 Información generada por javadocs

app
Class Punto

java.lang.Object **app.Punto**
All Implemented Interfaces:
java.lang.Comparable<Punto>

---

public class **Punto**
extends java.lang.Object
implements java.lang.Comparable<Punto>
Clase con los datos necesarios para representar un punto (x, y), y | | (0,0) -> |_____ x
Since:
0.1
Version:
2.0
Author:
Eduardo Perez Mederos, Miguel Monterrey Varela, Jaime Gonzalez Valdes, Oscar Mateos Lopez

---

Field Summary	
private int	**x**
private int	**y**

Constructor Summary
Punto() Constructor por defecto de la clase Punto.
Punto(int x, int y) Constructor de la clase Punto.

Method Summary	
int	**compareTo**(Punto p) Metodo que compara dos puntos y determina cual es prioritario segun su coordenada y.
int	**getX**() Metodo que obtiene la coordenada x de un punto.

int	**getY**() Metodo que obtiene la coordenada y de un punto.
void	**setX**(int x) Metodo que asigna la coordenada x de un punto.
void	**setY**(int y) Metodo que asigna la coordenada y de un punto.
java.lang.String	**toString**() Metodo para resumir la informacion de un Punto en una cadena de caracteres.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Field Detail

x
private int **x**

---

y
private int **y**

Constructor Detail

Punto
public **Punto**()
Constructor por defecto de la clase Punto. Crea un punto situado en el origen de coordenadas.

Punto
public **Punto**(int x, int y)
Constructor de la clase Punto. Crea un punto dadas sus coordeandas x e y.
Parameters:
x - Coordenada x.
y - Coordenada y.

Method Detail

setX
public void **setX**(int x)
Metodo que asigna la coordenada x de un punto.
Parameters:
x - Nueva coordeanda x.

---

getX
public int **getX**()
Metodo que obtiene la coordenada x de un punto.

Returns:
Coordenada x del punto.

---

setY
public void **setY**(int y)
Metodo que asigna la coordenada y de un punto.
Parameters:
y - - Nueva coordeanda y.

---

getY
public int **getY**()
Metodo que obtiene la coordenada y de un punto.
Returns:
Coordenada y del punto.

---

compareTo
public int **compareTo**(Punto p)
Metodo que compara dos puntos y determina cual es prioritario segun su coordenada y.
Specified by:
compareTo in interface java.lang.Comparable<Punto>
Parameters:
p - Punto con el que compararse.
Returns:
1, 0 o -1 en funcion de si el comparador es mayor, igual o menor al comparado.

---

toString
public java.lang.String **toString**()
Metodo para resumir la informacion de un Punto en una cadena de caracteres.
Overrides:
toString in class java.lang.Object
Returns:
cadena de caracteres con el punto.

3.6 PuntoCota

Clase con los datos necesarios para representar la altura máxima de un punto.

Sencilla clase para averiguar la altura máxima para un punto disponible de una caja.

También compara dos puntos para averiguar su prioridad frente a otro punto.

3.6.1 Información generada por javadocs

app
Class PuntoCota

java.lang.Object **app.PuntoCota**
All Implemented Interfaces:
java.lang.Comparable<PuntoCota>

---

public class **PuntoCota**
extends java.lang.Object
implements java.lang.Comparable<PuntoCota>
Clase con los datos necesarios para representar la altura maxima de un punto.
Since:
0.2
Version:
2.0
Author:
Eduardo Perez Mederos, Miguel Monterrey Varela, Jaime Gonzalez Valdes, Oscar Mateos Lopez

---

Field Summary	
private int	**cota** Altura maxima del punto.
private Punto	**punto**

Constructor Summary
PuntoCota(Punto p, int cota) Constructor de la clase PuntoCota.

Method Summary	
int	**compareTo**(PuntoCota pc) Metodo que compara dos puntos y determina cual es prioritario segun su coordenada y.
int	**getCota**() Metodo que devuelve la cota.
Punto	**getPunto**() Metodo que devuelve el punto.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

punto
private Punto **punto**

---

cota
private int **cota**
Altura maxima del punto.

Constructor Detail

PuntoCota
public **PuntoCota**(Punto p, int cota)
Constructor de la clase PuntoCota.
Parameters:
p - - Punto
cota - - Altura maxima que se puede alcanzar desde un punto.

Method Detail

getPunto
public Punto **getPunto**()
Metodo que devuelve el punto.
Returns:
punto

---

getCota
public int **getCota**()
Metodo que devuelve la cota.
Returns:
cota

---

compareTo
public int **compareTo**(PuntoCota pc)
Metodo que compara dos puntos y determina cual es prioritario segun su coordenada y.
Specified by:
compareTo in interface java.lang.Comparable<PuntoCota>
Parameters:
pc - - PuntoCota con el que compararse.
Returns:
1, 0 o -1 en funcion de si el comparador es mayor, igual o menor al comparado.

4. Diseño de las heurísticas

4.1 Busqueda aleatoria pura

- Para la búsqueda aleatoria pura, elegiremos una muestra de soluciones al azar y se devolvemos la mejor.
- El entorno de la solución es todo el espacio de búsqueda.
- Utilizaremos dos criterios de parada:
    - No mejora la solución optima en n-veces.
    - Se ejecutara la función n-veces.

```
public Solucion BAP(int criterio, int veces) {
	int clembuterol = veces;
	Solucion solInicial = new Solucion(Solucion.ALEATORIA,
	Problema.getRectangulos(),
	Problema.getAltoCaja(), Problema.getAnchoCaja(), Solucion.FINITE);

	Solucion mejorSolucion = solInicial;

	do {
		Solucion solActual = new Solucion(Solucion.ALEATORIA,
		Problema.getRectangulos(),
		 Problema.getAltoCaja(), Problema.getAnchoCaja(),0);

		if (solActual.compareTo(mejorSolucion) < 0) {
			mejorSolucion = solActual;

			if (criterio == NOMEJORA) {
				clembuterol = veces;
			}
		}
		clembuterol--;
	} while (clembuterol > 0);
	return mejorSolucion;
}
```

4.2 Búsqueda por recorrido al azar

- Persigue una mayor capacidad de exploración o diversificación de la búsqueda.
- El procedimiento consiste en realizar transformaciones de la solución actual hasta encontrar una mejora; cuando esto ocurre, se toma la nueva solución como solución actual y se continúa la búsqueda desde ella.
- En la elección de estas transformaciones o movimientos es conveniente tener en cuenta también una serie de aspectos: en que dirección realizar los movimientos, que lejanía abarcar en las transformaciones. Si se repiten los fracasos en la búsqueda de una solución mejor, se puede probar otra forma distinta de realizar las transformaciones.

```
public Solucion BRA(int criterio, int veces) {
      int clembuterol = veces;
      Solucion solInicial = new Solucion(Solucion.ALEATORIA,
      Problema.getRectangulos(),
      Problema.getAltoCaja(), Problema.getAnchoCaja(), Solucion.FINITE);

      Solucion solActual = solInicial;
      Solucion mejorSolucion = solActual;

      do {
            solActual = GeneraSVecina(solActual);
            if (solActual.compareTo(mejorSolucion) < 0) {
                  mejorSolucion = solActual;

                  if (criterio == NOMEJORA) {
                        clembuterol = veces;
                  }
            }
            clembuterol--;
      } while (clembuterol > 0);
      return mejorSolucion;
}
```

4.3 Búsqueda local

- Dada una solución se obtienen sus vecinos por medio de una regla.
- Se sustituye la solución por el mejor vecion si cumple el criterio de aceptación.
- Se repiten los pasos anteriores mientras se satisfaga el criterio de continuación.
- También usamos una función auxiliar llamada **GeneraSVecina** (solución actual) :
    - Esta función devuelve una solucion vecina a partir de otra solución.
    - Una solución vecina de otra se obtiene al intercambiar el órden en que se consideran dos rectángulos.

```
public Solucion BL(Solucion solActual) {
     Solucion mejorSolucion = solActual;
     boolean mejor = false;

     do {
          mejor = false;
          solActual = GeneraSVecina(solActual);

          if (solActual.compareTo(mejorSolucion) < 0) {
               mejorSolucion = solActual;
               mejor = true;
          }
     } while (mejor);
```

4.4 GRASP

- GRASP son las iniciales en ingles de *Greedy Randomize Adaptive Search Procedures* (*Procedimientos de Búsqueda basados en funciones Ávidas, Aleatorias y Adaptativas*); se dieron a conocer a finales de los ochenta en el trabajo de Feo T.A. y Resende M.G.C. (1989), pero han tenido un desarrollo más eficiente que los otras meta-heurísticas.
- GRASP es una técnica simple aleatoria e iterativa, en la que cada iteración provee una solución al problema que se esté tratando. La mejor solución de todas las iteraciones GRASP se guarda como resultado final. Hay dos fases en cada iteración GRASP: la primera construye secuencial e inteligentemente una solución inicial por medio de una función ávida, aleatoria y adaptativa; en la segunda se fase aplica un procedimiento de búsqueda local a la solución construida, con la esperanza de encontrar una mejora.
- En la fase de construcción se va añadiendo en cada paso un elemento, hasta obtener la solución completa. En cada iteración, la elección del próximo elemento para ser añadido a la solución parcial, viene determinado por una función ávida (greedy). Esta función mide el beneficio, según la función objetivo, de añadir cada elemento.

```
public Solucion GRASP(Solucion solInicial, int veces) {
     int clembuterol = veces;
     Solucion mejorSolucion = solInicial;

     do {
          Solucion solActual = new
          Solucion(Solucion.ALEATORIA,
          Problema.getRectangulos(),
          Problema.getAltoCaja(), Problema.getAnchoCaja(),1);

          if (solActual.compareTo(mejorSolucion) < 0) {
               mejorSolucion = solActual;
               clembuterol = veces;
          }

          clembuterol--;

     } while (clembuterol > 0);

     return mejorSolucion;
}
```

4.5 Búsqueda con arranque múltiple

- Los procedimientos de búsqueda con Arranque Múltiple (Multi-Start) realizan varias búsquedas monótonas partiendo de diferentes soluciones iniciales.
- La búsqueda monótona implicada puede ser cualquiera de las anteriormente descritas. Una de las formas más simples de llevar esto a cabo consiste en generar una muestra de soluciones iniciales o de arranque.
- Esto es equivalente a generar al azar una nueva solución de partida cada vez que la búsqueda quede estancada en el entorno de una solución óptima local.

```
public Solucion BAM(int veces) {
     Solucion solInicial = new Solucion(Solucion.ALEATORIA,
     Problema.getRectangulos(),
      Problema.getAltoCaja(), Problema.getAnchoCaja(),0);

     int clembuterol = veces;
     Solucion mejorSolucion = solInicial;

   do {
          solInicial = BL(solInicial);

          if (solInicial.compareTo(mejorSolucion) < 0) {
               mejorSolucion = solInicial;
               clembuterol = veces;
          }

          clembuterol--;

   } while (clembuterol > 0);
     return mejorSolucion;
}
```

4.6 Recocido simulado

- Es un meta-algoritmo genérico para la optimización global del problema, es decir, encontrar una buena aproximación al óptimo global de una función en un espacio de búsqueda grande.
- Para ciertos problemas, puede ser muy eficaz a condición de que la meta sea simplemente encontrar una solución aceptablemente buena en una cantidad de tiempo fija, antes que la solución mejor.

```
	public Solucion RS() {
		Solucion mejorSolucion = new Solucion(Solucion.ALEATORIA,
Problema.getRectangulos(), Problema.getAltoCaja(),
				Problema.getAnchoCaja(), Solucion.FINITE);
		Solucion solActual = mejorSolucion;

		Random r = new Random(System.nanoTime());

		int k = 0;
		double alfa = 0.9;
		double mediaObj = 0;

		for (int i = 0; i < 100; i++) {
			Solucion s = new Solucion(Solucion.ALEATORIA,
Problema.getRectangulos(), Problema.getAltoCaja(),
					Problema.getAnchoCaja(), Solucion.FINITE);

			mediaObj += s.getFObjetivo();
		}

		mediaObj = mediaObj / 100;

		double temperatura = 50 * mediaObj;
		double L = (int)Math.pow(Problema.getRectangulos().size(), 2);

		do {
			solActual = GeneraSVecina(solActual);

			for (int m = 0; m < L; m++) {
				if (mejorSolucion.getFObjetivo() > solActual.getFObjetivo()) {
					mejorSolucion = solActual;
				}
				else if (Math.exp((mejorSolucion.getFObjetivo() -
solActual.getFObjetivo()) / temperatura) > r.nextFloat()) {
					mejorSolucion = solActual;
				}
			}

			k++;

			temperatura = temperatura * alfa;
			L = k * 1.05;
		} while (temperatura > 0.1);

		return mejorSolucion;
	}
```

4.7 Búsqueda por entornos variable

4.7.1 VND (Variable neighbourhood descent)

La Búsqueda de Entorno Variable es una metaheurística propuesta recientemente (1995) por Pierre Hansen y Nenad Mladenovi.

Está basada en el principio de cambiar sistemáticamente de estructura de entornos dentro de la búsqueda. Cuando la búsqueda queda atrapada, se cambia a un entorno más amplio.

- Características:
  - Sencillez.
  - Permite extensiones del esquema básico.
- La VND está basada en tres hechos simples:
  - Un mínimo local con una estructura de entornos no lo es necesariamente con otra.
  - Un mínimo global es mínimo local con todas las posibles estructuras de entornos.
  - Para muchos problemas, los mínimos locales con la misma o distinta estructura de entorno están relativamente cerca.
- En la implementación de VND nos apoyaremos en una funcion auxiliar llamada *GeneraSEntorno(solInicial,k).* Método que devuelve la solución del problema, generando entornos, que obtenemos a rodando k posiciones para el final de la permutación, siendo aleatoria la elección de la primera posición a rodar y contiguas a ésta, el resto.

```
public Solucion VND(int veces) {
            Solucion solInicial = new Solucion(Solucion.ALEATORIA,
Problema.getRectangulos(),
                        Problema.getAltoCaja(), Problema.getAnchoCaja(),0);

            int clembuterol = veces;
            int k = 2;
            Solucion mejorSolucion = solInicial;


            do { GeneraSEntorno(solInicial,k)
                  solInicial = GeneraSEntorno(solInicial,k);
                  if (solInicial.compareTo(mejorSolucion) < 0) {
                        mejorSolucion = solInicial;
                        clembuterol = veces;
                        k = 2;
                  }
                  k++;
                  clembuterol--;
            } while (k == 0);

            return mejorSolucion;
      }
```

4.7.2 BVNS: VND Básico. Búsquedas multistart

La búsqueda de entorno variable básica (Basic Variable Neighbourhood Search, BVNS) combina cambios determinísticos y aleatorios de estructura de entornos. No cambia en demasía con el código anterior.

```
      public Solucion BVNS(int veces) {
            Solucion solInicial = new Solucion(Solucion.ALEATORIA,
Problema.getRectangulos(),
                        Problema.getAltoCaja(),
Problema.getAnchoCaja(),0);

            int clembuterol = veces;
            int k = 2;
            Solucion mejorSolucion = solInicial;


            do {
                  solInicial = Agitacion(solInicial,k);
                  solInicial = BL(solInicial);
                  if (solInicial.compareTo(mejorSolucion) < 0) {
                        mejorSolucion = solInicial;
                        clembuterol = veces;
                        k = 2;
                  }
                  k++;
                  clembuterol--;
            } while (k == 0);

            return mejorSolucion;
      }
```

4.8 Busquedas dispersas

La Búsqueda Dispersa es un método evolutivo capaz de obtener soluciones de calidad a problemas difíciles. Algunos aspectos están claramente establecidos y otros requieren de más estudio. Su arquitectura permite el diseño de sistemas independientes del contexto, actualmente está en plena expansión.

Combina las siguientes caracteristicas :

- Trabaja con un conjunto moderado de buenas soluciones, Poblacion inicial, conjunto de referencia, subconjuntos.
- Combina inteligentemente esas soluciones.
- Incorpora criterios de mejora y dispersion, mejora la solucion mediante la busqueda local.
- Y no se deja nada al azar.

Esta heurística utiliza tres funciones auxiliares, *CrearPoblacionInicial* *Combinarsoluciones* y *SeleccionarSubconjunto*.

Crear poblacion inicial: Crea un conjunto de soluciones de calidad y dispersas mediante la heuristica de colocacion inicial GRASP.

Combinar soluciones: Método que combina los subconjuntos de soluciones seleccionadas del conjunto de referencia para obtener una nueva solución.

SeleccionarSubconjunto: Selecciona todos los subcojuntos de soluciones del conjunto de referencia para realizar buenas combinaciones.

```
Solucion BD(int veces) {
	ArrayList<Solucion> Poblacion = new ArrayList<Solucion>();
	ArrayList<Solucion> RefSet = new ArrayList<Solucion>();
	ArrayList<Solucion> Subconjunto = new ArrayList<Solucion>();
	Solucion mejorSolucion = new Solucion();
	int clembuterol = veces;
	int intentos = 0;
	SeleccionarSubconjunto((ArrayList<Solucion>)RefSet.clone(), i, j);
	Solucion solActual = CombinarSoluciones(Subconjunto);
	solActual = BL(solActual); // Mejorar la solucion
	// Actualizar RefSet
	if ((solActual.compareTo(RefSet.get(i)) < 0) ||
(solActual.compareTo(RefSet.get(j)) < 0)) {
		if (solActual.compareTo(RefSet.get(i)) < 0) {
			RefSet.remove(RefSet.get(i));
		} else {
			RefSet.remove(RefSet.get(j));
			}

			RefSet.add(solActual);

			i = 0;
			j = 1;
			} else {
			j++;

			if (j > RefSet.size() - 1) {
				i++;
				j = i + 1;
			}
		}

		if (i == RefSet.size() - 1) {
			gnrs = true;
		}
	} while (!gnrs);

		mejorSolucion = RefSet.get(0);

		for (int x = 1; x < RefSet.size(); x++) {
			if (RefSet.get(x).compareTo(mejorSolucion) < 0) {
				mejorSolucion = RefSet.get(x);
			}
		}
		// Si para un numero de intentos no mejora
		// creamos una nueva poblacion
		intentos--;
		if (intentos == 0) {
			gnp = true;
		}
	} while (!gnp);
	clembuterol--;
} while (clembuterol > 0);

return mejorSolucion;
```

5.9 Busqueda Tabú

La búsqueda tabú es un algoritmo metaheurístico que puede utilizarse para resolver problemas de optimización combinatoria.

La búsqueda tabú utiliza un procedimiento de búsqueda local o por vecindades para moverse iterativamente desde una solución x hacia una solución x' en la vecindad de x, hasta satisfacer algún criterio de parada. Para poder explorar regiones del espacio de búsqueda que serían dejadas de lado por el procedimiento de búsqueda local (ver óptimo local), la búsqueda tabú modifica la estructura de vecinos para cada solución a medida que la búsqueda progresa. Las soluciones admitidas para N * (x), el nuevo vecindario, son determinadas mediante el uso de estructuras de memoria. La búsqueda entonces progresa moviéndose iterativamente de una solución x hacia una solución x'en *N * (x)*.

```
      public Solucion BT(int veces) {
            Solucion solInicial = new Solucion(Solucion.ALEATORIA,
Problema.getRectangulos(),
                        Problema.getAltoCaja(), Problema.getAnchoCaja(),0);

            int permutacion[] = solInicial.getPermutacion();
            Solucion mejorSolucion = solInicial;
            int i = 0, j = 0, clembuterol = veces;
            boolean encontrado = false;

            Integer[] pares;
        pares = new Integer[2];

            Queue<Integer[]> colatabu = new LinkedList<Integer[]>();

            do {
                  do {
                        Random rnd = new Random(System.nanoTime());

                        i = (int)(rnd.nextDouble() * permutacion.length);
                        j = (int)(rnd.nextDouble() * permutacion.length);

                        if (i != j) {
                              pares[0] = i;
                              pares[1] = j;
                              if (!colatabu.contains(pares)) {
                                    if (colatabu.size() < 7) {
                                          colatabu.add(pares);

                                    } else {
                                          colatabu.remove();
                                          colatabu.add(pares);
                                    }
                                    encontrado = true;
                                    swap(permutacion, i, j);
                              }
                        }


                  } while (!encontrado);

                  solInicial = new Solucion(permutacion,
Problema.getRectangulos(), Problema.getAltoCaja(),
                              Problema.getAnchoCaja(), Solucion.FINITE);

                  if (solInicial.compareTo(mejorSolucion) < 0) {
                        mejorSolucion = solInicial;
                        clembuterol = veces;
                  }

                  clembuterol--;

            } while (clembuterol > 0);

            return mejorSolucion;
      }
```

4.9 Información generada por JavaDocs

app
Class Heuristica

java.lang.Object **app.Heuristica**

---

public class **Heuristica**
extends java.lang.Object
Clase con los metodos heuristicos para obtener mejores soluciones.
Since:
1.0
Version:
2.0
Author:
Eduardo Perez Mederos, Miguel Monterrey Varela, Jaime Gonzalez Valdes, Oscar Mateos Lopez

---

Field Summary	
static int	**BAM** Metodos multiarranque
static int	**BAP** Busquedas por entornos
static int	**BD** Heuristicas poblacionales
static int	**BL**
static int	**BRA**
static int	**BT** Heuristicas poblacionales
static int	**BVNS**
static int	**GRASP** GRASP
private Solucion	**MejorSolucion**
static int	**NOMEJORA** Criterios de parada
static int	**NVECES**
private Problema	**Problema**

static int	**RS** Recocido simulado
static int	**VND** Busqueda por entornos variables

Constructor Summary
Heuristica(int tipoHeuristica, Problema p) Constructor dado el tipo de generacion de solucion inicial y el numero de rectangulos.

Method Summary	
Solucion	**Agitacion**(Solucion s, int k) Metodo que devuelve la solucion del problema.
Solucion	**BAM**(int veces) Metodo que devuelve la solucion del problema.
Solucion	**BAP**(int criterio, int veces) Metodo que devuelve la solucion del problema.
(package private) Solucion	**BD**(int veces) Metodo que devuelve la solucion del problema.
Solucion	**BL**(Solucion solActual) Metodo que devuelve la solucion del problema.
Solucion	**BRA**(int criterio, int veces) Metodo que devuelve la solucion del problema.
Solucion	**BT**(int veces) Metodo que devuelve la solucion del problema.
Solucion	**BVNS**(int veces) Metodo que devuelve la solucion del problema.
private Solucion	**CombinarSoluciones**(java.util.ArrayList<Solucion> subconjunto) Metodo que combina los subconjuntos de soluciones seleccionadas del conjunto de referencia para obtener una nueva solucion.
private java.util.ArrayList<Solucion>	**CrearPoblacion**(int tamPoblacion) Metodo que crea una poblacion.
private java.util.ArrayList<Solucion>	**GenerarReferenceSet**(int tamRS, java.util.ArrayList<Solucion> pobAux) Metodo que crea un conjunto de referencia.

Solucion	**GeneraSEntorno**(Solucion s, int k) Metodo que devuelve la solucion del problema, generando entornos que obtenemos a rodando k posiciones para el final de la permutacion, si- endo aleatoria la eleccion de la primera posicion a rodar y contiguas a esta el resto.
Solucion	**GeneraSVecina**(Solucion s) Metodo que devuelve una solucion vecina a partir de otra solucion.
Solucion	**getSolucion**() Metodo que devuelve la solucion del problema
Solucion	**GRASP**(Solucion solInicial, int veces) Metodo que devuelve la solucion del problema.
Solucion	**RS**() Metodo que devuelve la solucion del problema.
private java.util.ArrayList<Solucion>	**SeleccionarSubconjunto**(java.util.ArrayList<Solucion> RefSet, int x, int y) Metodo que crea un subconjunto.
private void	**swap**(int[] array, int posOne, int posTwo) Metodo que se encarga de intercambiar elementos para
java.lang.String	**toString**() Metodo para resumir la informacion de una heuristica en una cadena de caracteres.
Solucion	**VND**(int veces) Metodo que devuelve la solucion del problema.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Field Detail

NOMEJORA
public static final int **NOMEJORA**
Criterios de parada
See Also:
Constant Field Values

NVECES
public static final int **NVECES**
See Also:
Constant Field Values

BAP
public static final int **BAP**
Busquedas por entornos
See Also:
Constant Field Values

BRA
public static final int **BRA**
See Also:
Constant Field Values

BL
public static final int **BL**
See Also:
Constant Field Values

GRASP
public static final int **GRASP**
GRASP
See Also:
Constant Field Values

BAM
public static final int **BAM**
Metodos multiarranque
See Also:
Constant Field Values

RS
public static final int **RS**
Recocido simulado
See Also:
Constant Field Values

VND
public static final int **VND**
Busqueda por entornos variables
See Also:
Constant Field Values

BVNS
public static final int **BVNS**
See Also:
Constant Field Values

BT
public static final int **BT**

Heuristicas poblacionales

See Also:

Constant Field Values

BD
public static final int **BD**

Heuristicas poblacionales

See Also:

Constant Field Values

Problema
private Problema **Problema**

MejorSolucion
private Solucion **MejorSolucion**

Constructor Detail

Heuristica
public **Heuristica**(int tipoHeuristica, Problema p)

Constructor dado el tipo de generacion de solucion inicial y el numero de rectangulos.

Parameters:

tipoHeuristica - Tipo de heuristica.

p - Problema.

Method Detail

BAP
public Solucion **BAP**(int criterio, int veces)

Metodo que devuelve la solucion del problema. Busca la mejor solucion posible cambiando aleatoriamente las permutaciones que le pasamos al algoritmo de colocacion

Parameters:

criterio -

veces -

Returns:

Solucion - Solucion del problema

BRA

public Solucion **BRA**(int criterio, int veces)

Metodo que devuelve la solucion del problema. Busca nuevas vecinas intercambiando al azar algunas posiciones de la permutacion.

Parameters:

criterio -

veces -

Returns:

Solucion - Solucion del problema.

BL

public Solucion **BL**(Solucion solActual)

Metodo que devuelve la solucion del problema. Busqueda local.

Parameters:

solActual -

Returns:

Solucion - Solucion del problema.

GRASP

public Solucion **GRASP**(Solucion solInicial, int veces)

Metodo que devuelve la solucion del problema. Este metodo va seleccionando para cada rectangulo los 3 mejores puntos donde ponerlo, (donde menos desperdicio genera) y a partir de ahÃ inserta en uno de los 3 eligiendo al azar.

Parameters:

solInicial -

veces -

Returns:

Solucion - Solucion del problema.

BAM

public Solucion **BAM**(int veces)

Metodo que devuelve la solucion del problema. Genera una solucion inicial y apartir de ahÃ N soluciones mediante Busquedas Locales haciendo las per tinentes comparaciones para quedarnos con la mejor.

Parameters:

veces -

Returns:

Solucion - Solucion del problema.

BT

public Solucion **BT**(int veces)

Metodo que devuelve la solucion del problema. Genera cambios aleatorios en las permutaciones pero ademÃ¡sa, contiene una cola-memoria que nos ayuda a evitar repetir el mismo movimiento en las siguietnes N (en este caso 7) veces.

Parameters:

veces -

Returns:

Solucion - Solucion del problema.

BVNS

public Solucion **BVNS**(int veces)

Metodo que devuelve la solucion del problema. Metodo de busquedas por entorno el cual primero produce una agitacion en la permutaciÃ³n y posteriormente rea- liza una Busqueda Local.

Parameters:

veces -

Returns:

Solucion - Solucion del problema.

VND

public Solucion **VND**(int veces)

Metodo que devuelve la solucion del problema. Metodo de busquedas por entorno.

Parameters:

veces -

Returns:

Solucion - Solucion del problema.

RS

public Solucion **RS**()

Metodo que devuelve la solucion del problema. Metodo de recocido simulado.

Returns:

Solucion - Solucion del problema.

BD

Solucion **BD**(int veces)

Metodo que devuelve la solucion del problema. Metodo de busqueda dispersa. Realiza dentro la mejora de la solucion combinada. Y actualiza el RefSet.

Parameters:

veces -

Returns:

Solucion - Solucion del problema.

GeneraSVecina

public Solucion **GeneraSVecina**(Solucion s)

Metodo que devuelve una solucion vecina a partir de otra solucion. Una solucion vecina de otra se obtiene al intercambia el orden en que se consideran dos rectangulos.

Parameters:

s - Solucion inicial

Returns:

Solucion - Solucion del problema.

GeneraSEntorno

public Solucion **GeneraSEntorno**(Solucion s, int k)

Metodo que devuelve la solucion del problema, generando entornos que obtenemos a rodando k posiciones para el final de la permutacion, si- endo aleatoria la eleccion de la primera posicion a rodar y contiguas a esta el resto.

Parameters:

s - Solucion inicial

k - Numero de elementos

Returns:

Solucion - Solucion del problema.

Agitacion

public Solucion **Agitacion**(Solucion s, int k)

Metodo que devuelve la solucion del problema. Genera nuevos entornos mediante la agitacion que consiste en intercambiar la posicion de x elementos (valor que nos llega por parametro, k) en al permutacion.

Parameters:

s - Solucion inicial

k - Numero de elementos

Returns:

Solucion - Solucion del problema.

CrearPoblacion

private java.util.ArrayList<Solucion> **CrearPoblacion**(int tamPoblacion)

Metodo que crea una poblacion. Una poblacion es un conjunto de soluciones.

Parameters:

tamPoblacion -

Returns:

ArrayList - Array de soluciones

GenerarReferenceSet

private java.util.ArrayList<Solucion> **GenerarReferenceSet**(int tamRS, java.util.ArrayList<Solucion> pobAux)

Metodo que crea un conjunto de referencia. Un conjunto de referencia es un nuevo conjunto creado a partir de la poblacioÌ□n con soluciones de alta calidad y dispersas.

Parameters:
tamRS -
pobAux -
Returns:
ArrayList - Array de soluciones

SeleccionarSubconjunto
private java.util.ArrayList<Solucion>
SeleccionarSubconjunto(java.util.ArrayList<Solucion> RefSet,
int x, int y)
Metodo que crea un subconjunto. Selecciona todos los subcojuntos de soluciones del conjunto de referencia para realizar buenas combinaciones.
Parameters:
RefSet -
x -
y -
Returns:
ArrayList - Array de soluciones

CombinarSoluciones
private Solucion **CombinarSoluciones**(java.util.ArrayList<Solucion> subconjunto)
Metodo que combina los subconjuntos de soluciones seleccionadas del conjunto de referencia para obtener una nueva solucion.
Parameters:
subconjunto -
Returns:
Solucion - Nueva solucion combinada

---

swap
private void **swap**(int[] array, int posOne, int posTwo)
Metodo que se encarga de intercambiar elementos para
Parameters:
array - - Array que contiene la permutacion.
posOne - - Posicion del primer elemento a intercambiar.
posTwo - - Posicion del segundo elemento a intercambiar.

---

getSolucion
public Solucion **getSolucion**()
Metodo que devuelve la solucion del problema
Returns:
Solucion - Solucion del problema.

---

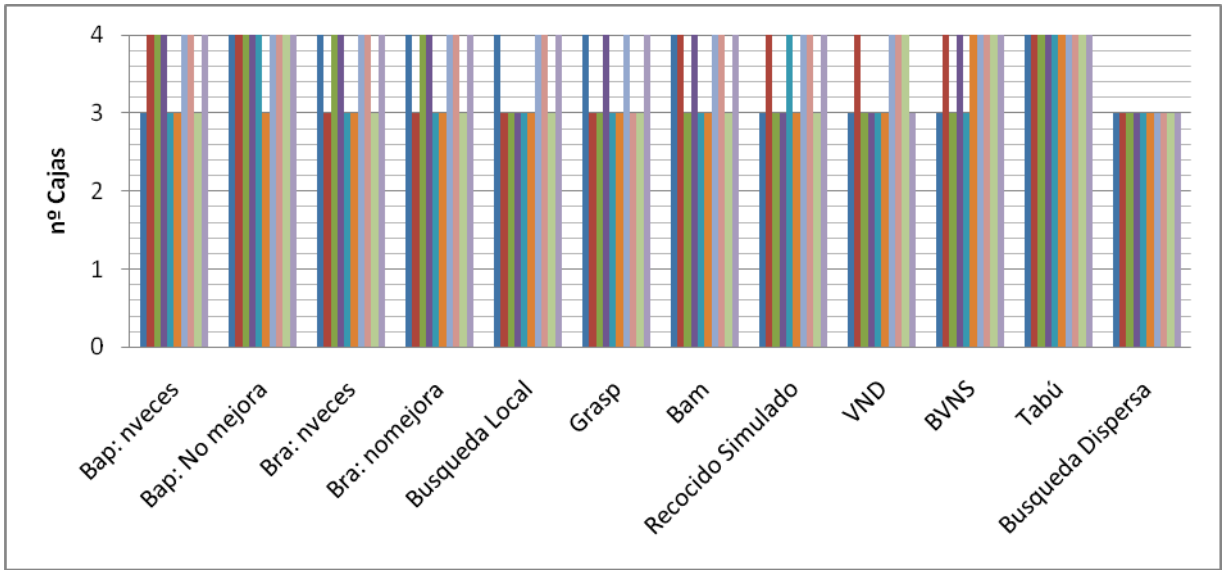
toString
public java.lang.String **toString**()
Metodo para resumir la informacion de una heuristica en una cadena de caracteres.
Overrides:

toString in class java.lang.Object
Returns:
String - Mejor solucion de una heuristica.

5. Conclusiones

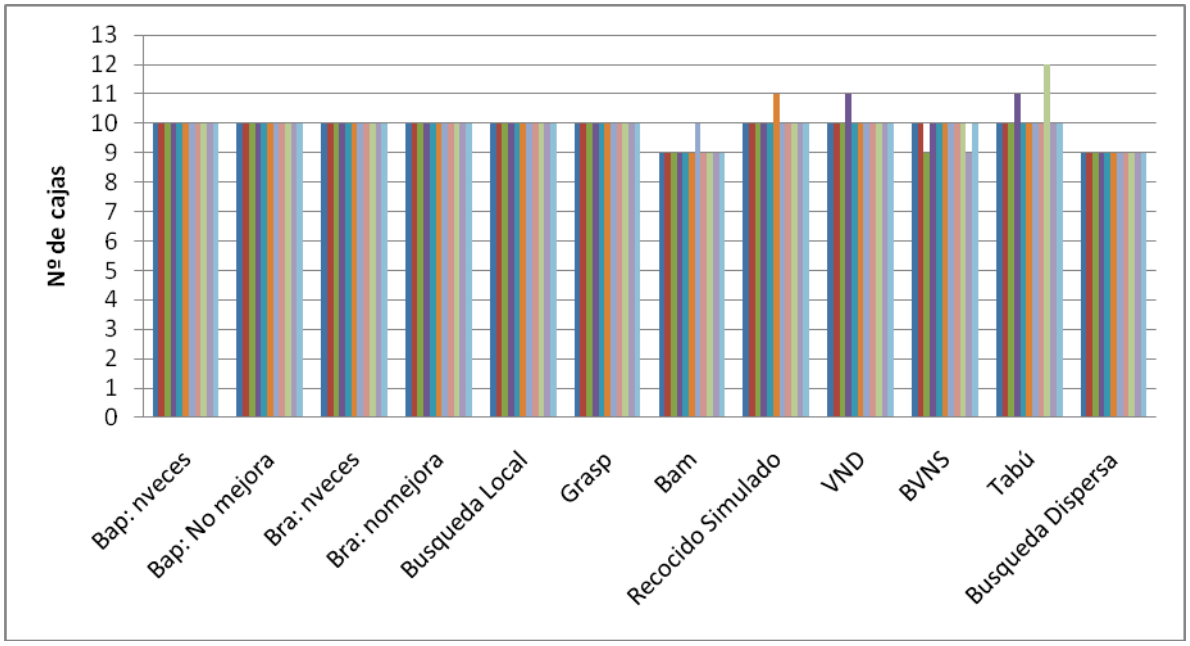
Como las heurísticas usan funciones random() en algunas ocasiones observamos resultados distintos en diferentes ejecuciones, así que a continuación veremos los resultados de las distintas heurísticas durante 10 ejecuciones diferentes. Las heurísticas que además requieran iteraciones quedarán fijadas en 7.

Para el primer módulo (caja 12x12 y 31 elementos) de valores conseguimos los siguientes resultados:



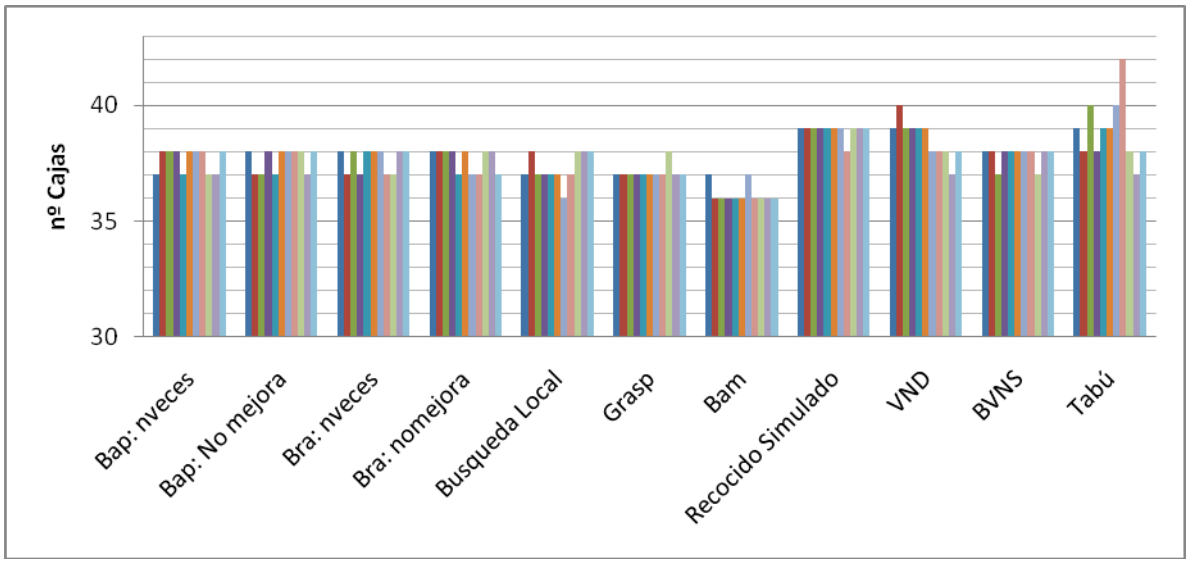
En este caso, observamos que apenas existe mejora entre las diferentes heurísticas, esto se debe a que trabajamos con pocos valores, Aunque observamos como en la Búsqueda Local, tenemos muchas posibilidades de obtener un buen resultado con pocos intentos en un tiempo mucho menor. La búsqueda dispersa, es la que mejores resultados registra, pero resulta casi prohihibitiva debido a su coste de ejecución. Busqueda tabú ofrece unos malos resultados en relación a los demás.

Para el segundo módulo (caja 10x8 y 48 elementos) de valores conseguimos los siguientes resultados:



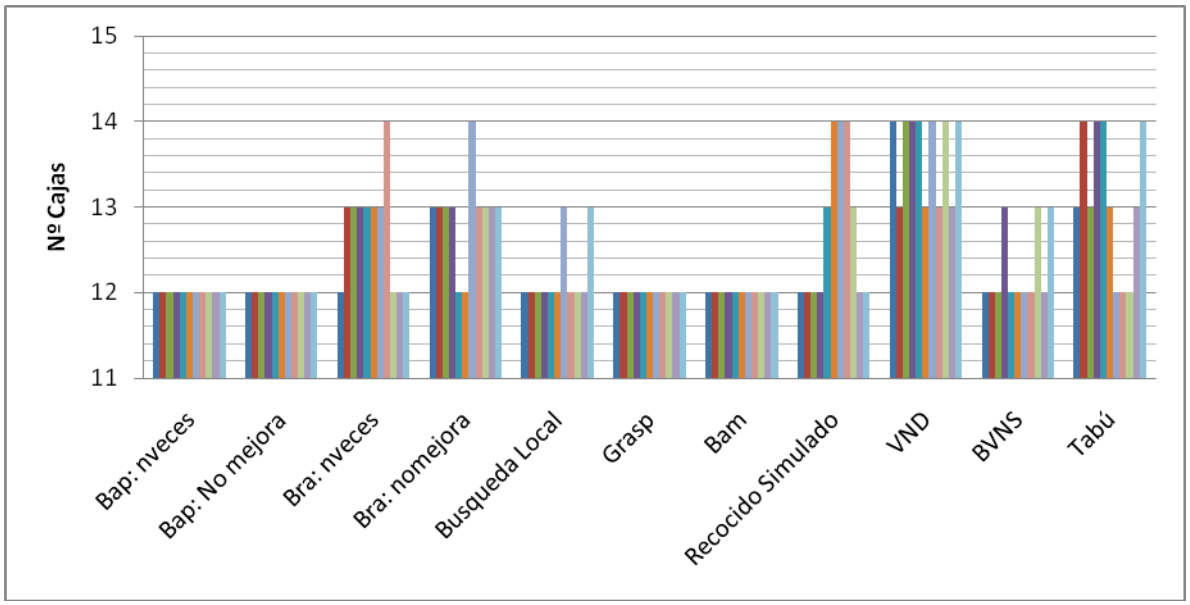
En este caso observamos que la Búsqueda con Arranque Multiple es la mejor opción, con un 90% de casos por debajo de la media y un tiempo aceptable. Busqueda dispersa, nos garantiza un buen resultado, pero su tiempo de ejecución es lamentable. Sorprenden los malos resultados del recocido simulado y el vnd, a la vez que vemos que aunque con escasa probabilidad BVNS también saca buenos resultados.

Para el tercer módulo (caja 10x8 y 161 elementos) de valores conseguimos los siguientes resultados:



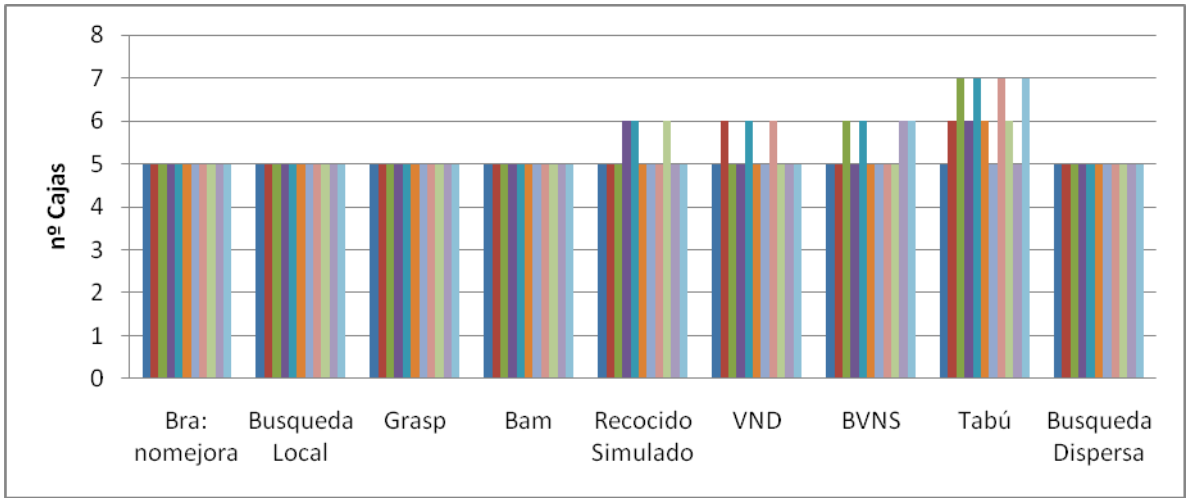
Vemos una vez más que el más fiable es el Bam, aunque su ejecución era mucho más lenta, 2 segundos le costaba dar un resultado correcto al Bam, mientras el resto de los daban de forma instantánea, choca de nuevo ver los resultados de VND viendo que es una heurística capaz de lo mejor, y de lo peor. Grasp se estableció en un buen resultado, con una respuesta más que solución aceptable y rápida. La búsqueda tabú da unos resultados bastante irregulares, y en ningún momento alcanza a la búsqueda multiarranque. Por ultimo nombrar que la búsqueda dispersa, no llego a ningún resultado dentro del tiempo esipulado. (15 min)

Para el cuarto módulo (caja 20x20 y 225 elementos) de valores conseguimos los siguientes resultados:



En este módulo, vemos que casi todas las heurísticas logran una buena solución de forma rápida, Bam tarda entre 2432 milisegundos más la ejecución frente a sus compañeras. Sorprende la diferencia de resultados entre VND y BVNS pese a estar en la misma familia. Tabú da unos resultados muy dispares entr distintas ejecuciones. Y una vez mas búsqueda dispersa se cae de la lista, al no ser capaz de dar un resultado dentro del tiempo estipulado.

Para el quinto módulo (caja 10x8 y 33 elementos) de valores conseguimos los siguientes resultados:



En este caso, al ser pocos elementos, todas las heurísticas llegan a la misma solución menos recocido simulado, VND y BVNS. Busqueda dispersa, y búsqueda multiarranque como siempre, las más lentas.

Ahora veremos una tabla con los tiempos de ejecución de cada heuristica, observamos que las mas rapidas son el VND y el BVNS con apenas 14 milisegundos en su ejecución. Por el otro lado vemos que la busqueda multiarranque tarda un poco mas que las demas, y ya por ultimo vemos como la búsqueda dispersa, dura mas de 2 minutos en ejecutarse :

Bap: No mejora	Bra: nveces	Bra: nveces	Bra: nomejora	Busqueda Local	Grasp
1998	1935	2820	3039	19	1645
Bam	Recocido Simulado	VND	BVNS	Tabú	Busqueda Dispersa
9167	3014	14	14	2695	120824

NOTA FINAL: La heurística que mejores resultados a dado a sido Bam (Búsqueda de acceso múltiple), pero con la pega de ser la más lenta de todas. Una Heurística que por lo general, ha dado buenos resultados y además ha sido rápida en su ejecución es GRASP, que en todas las pruebas a dado unos resultados más que aceptables.

6. Revisión bibliográfica

J. J. Merelo, Universiad de Granada, Técnicas heurísticas de resolución de problemas: computación evolutiva y redes neuronales, <http://geneura.ugr.es/~jmerelo/tutoriales/heuristics101/>

Bonilla, Capitulo II: Marco teórico, <http://catarina.udlap.mx/u_dl_a/tales/documentos/lmnf/bonilla_g_le/capitulo2.pdf>
Instituto de investigación de inteligencia artificial, <http://www.iiia.csic.es/es>

Revistas peruanas, <http://revistas.concytec.gob.pe>

Wikipedia, Bin packing problem, <http://en.wikipedia.org/wiki/Bin_packing_problem>

Wikipedia, Metaheuristics, <http://en.wikipedia.org/wiki/Metaheuristic>

D. Sleator, A 2.5 Times Optimal Algorithm for Packing in Two Dimensions, Information Processing Letters, Vol. 10., No. 1, 1980, <http:// www.springerlink.com>

European Journal of Operational Research, Vol. 141, No. 2. (1 September 2002), pp. 241-252. doi:10.1016/S0377-2217(02)00123-6 Key: citeulike:515305, <http://citeulike.org>

Jose Marcos Moreno Vega, Jose Andres Moreno Perez, Apuntes de la Asignatura, <http://campusvirtual.ull.es/1011/course/view.php?id=2448>