UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Optimización de empaquetado de piezas

Proyecto Final de Carrera

Ingeniería en Informática

Autor: Moisés Baly Rodríguez

Director: Miguel Sánchez López

10/04/2014

AGRADECIMIENTOS

Quisiera agradecer a mi director, Miguel Sánchez, por su constante apoyo a lo largo de la realización de este proyecto, su pronta respuesta a incontables e-mails a todas horas del día y de la noche y su orientación para llevar a buen puerto este modesto esfuerzo algorítmico. Quisiera agradecer también a todos los profesionales desconocidos que han provisto a través de su trabajo la motivación para la realización de este proyecto, y en muchos casos, los datos reales de los que nos hemos servido.

RESUMEN

El objetivo de este proyecto fin de carrera es poner a disposición pública una librería de código abierto para la resolución del problema del empaquetado automático de piezas no regulares en planchas 2D (conocido como Bin Packing Problem), con la menor cantidad de desperdicio de material posible y sin que exista solapado entre piezas. Se utilizan algoritmos relativamente simples que permiten servir como punto de partida para la reutilización y aplicación del código a problemas tales como empaquetado de piezas de caucho, metal o textiles. Más que proponer un algoritmo novedoso, nos hemos centrado en intentar paliar la escasez de librerías de código abierto para la resolución del problema en cuestión, intentando ofrecer al público una primera herramienta a partir de la cual puedan desarrollar una solución que cubra sus necesidades particulares.

Palabras clave: empaquetado, piezas, planchas, 2D, algoritmo, irregulares, código abierto, librería.

TABLA DE CONTENIDOS

CONTENIDOS

Capítulo 1 INTRODUCCIÓN

1.1 INTRODUCCIÓN

El empaquetado óptimo de piezas irregulares es un conocido problema NP-Difícil que aparece en multitud de situaciones a lo largo de varias industrias. Así, por ejemplo, puede darse el caso en la industria metalúrgica en que se necesite cortar piezas metálicas de variadas formas a partir de planchas rectangulares, y ello con el menor desperdicio de material posible. Otros casos, como el de la industria textil son ejemplos que ilustran la utilidad de contar con algoritmos eficientes que permitan obtener soluciones con un alto nivel de optimización.

Existen gran número de investigaciones que intentan dar solución al problema del empaquetado de piezas irregulares. Una de las estrategias más utilizadas es la del empaquetado de piezas rectangulares, usando el rectángulo circunscrito a la forma de una pieza; otras técnicas emplean heurísticas que parten de una solución inicial de baja calidad, pero que presenta la ventaja de tener un tiempo de cómputo bajo para, a continuación, introducir iterativamente mejoras hasta que se alcanza un tiempo límite de cálculo o se obtiene un valor de la función objetivo aceptable. También hemos encontrado trabajos donde la solución se modelaba mediante algoritmos genéticos o simulación de sistemas físicos de partículas, aunque para este último caso no hemos sido capaces de localizar una propuesta algorítmica concreta.

En este proyecto hemos empleado una mezcla de técnicas, intentando explotar la rapidez de algoritmos de empaquetado de piezas rectangulares 2D, con el valor añadido de usar estrategias de compresión o simulación básica de gravedad para aprovechar al máximo la irregularidad en la forma de las piezas. Debido al uso de esta mezcla de métodos, creemos que el algoritmo provee resultados aceptables para variedad de casos tales como conjuntos de piezas muy irregulares, conjuntos muy regulares (rectangulares) o conjuntos con alta repetición en la forma de las piezas.

1.2 OBJETIVO

El objetivo de este proyecto es doble: de un lado, el estudio de las distintas propuestas existentes y el empleo de una solución aceptable; de otro, poner a disposición pública una librería que pueda servir de punto de partida para aquellos profesionales/particulares que no dispongan de los recursos para el desarrollo de librerías de empaquetado propias desde cero, ya sea en términos de tiempo o económicos.

1.3 MOTIVACIÓN

El proyecto surge de la necesidad de profesionales de Valencia, España, de cortar de manera eficaz piezas de caucho, con el menor desperdicio posible. Se dieron cuenta de que no existían soluciones de código abierto adaptables disponibles al público, únicamente soluciones propietario con un alto coste. Es entonces que se propone el desarrollo de un proyecto que pueda servir tanto a ellos, como punto de partida para futuras mejoras, como a otros profesionales en la misma situación.

1.4 DEFINICIÓN DEL PROBLEMA

El empaquetado de piezas (Bin Packing Problem) es un problema NP-Difícil que puede ser informalmente enunciado de la siguiente forma:

Dado un conjunto de piezas de forma arbitraria, debemos ubicarlas en planchas rectangulares de dimensiones conocidas, con las siguientes restricciones:

- No existe solapado entre ningún par de piezas.
- El desperdicio de material debe ser el mínimo posible.

Podemos describir más formalmente la segunda restricción de la siguiente forma:

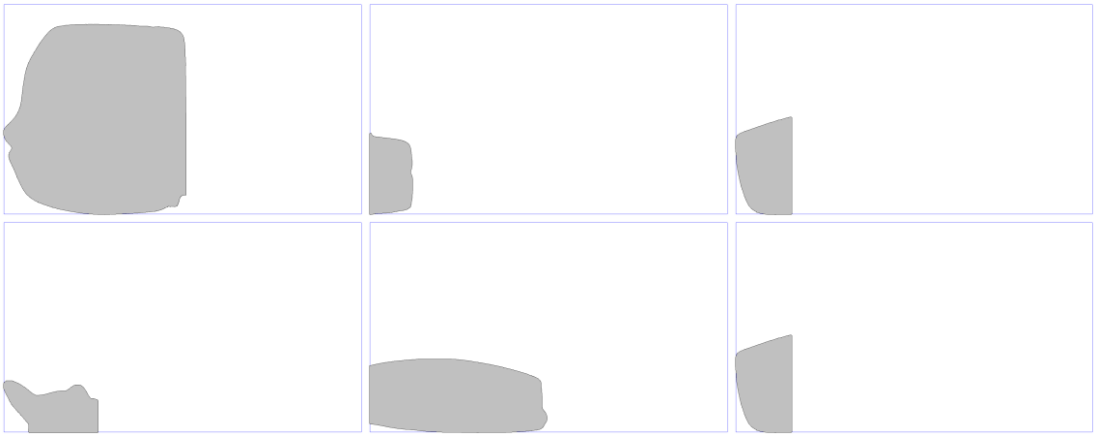
Sea **B** el conjunto de planchas bidimensionales utilizadas por el algoritmo y **P** el conjunto de piezas irregulares a ubicar. Entonces definiremos $\mathbf{A(B_i)}$ como el área de la i-ésima plancha utilizada y $\mathbf{A(P_j)}$ como el área de la j-ésima pieza ubicada. Definiremos nuestro desperdicio **D** de material como sigue:

$$\mathrm{D} = \sum_{i=1}^{n} \mathbf{A}(B_i) - \sum_{j=1}^{m} \boldsymbol{A}(P_j)$$

Entonces nuestro objetivo es conseguir el mínimo desperdicio posible:

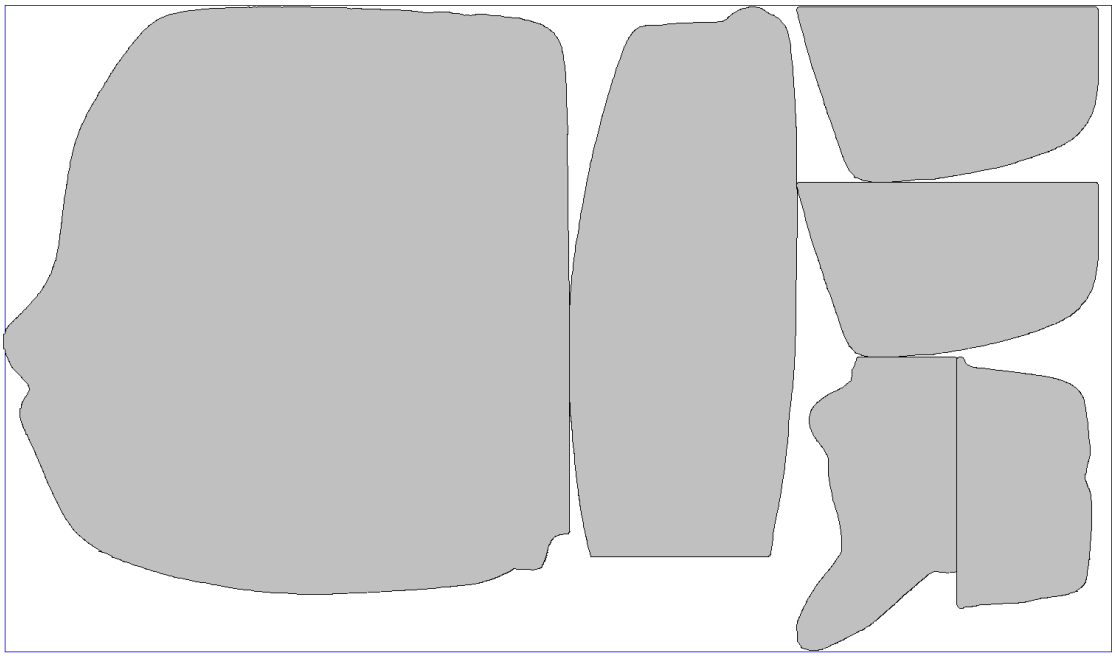
$$\mathbf{min}(\mathrm{D})$$

Así, por ejemplo, imaginemos que tenemos el siguiente set de piezas irregulares:



Nuestro objetivo consiste en agrupar las piezas usando la menor cantidad de espacio posible. En dependencia de la calidad de nuestro algoritmo, conseguiremos acercarnos más o menos a la solución óptima (sin poder garantizar nunca la optimalidad de la solución, ver apartado 1.5). Una solución trivial sería dejar cada pieza en una plancha, con lo cual obtendríamos un resultado como el ilustrado por las imágenes precedentes (6 planchas utilizadas). Dicho resultado, a pesar de cumplir nuestra restricción de no solapado, no sería aceptable, y a lo largo de este proyecto intentamos ilustrar una de las tantas maneras de calcular la segunda restricción (desperdicio mínimo).

Así pues, una vez aplicado nuestro algoritmo y respetando ambas restricciones, un posible resultado sería el siguiente:



Como se puede observar se obtiene un emplazamiento en el que no existe solapado entre ningún par de piezas, y se ha utilizado la menor cantidad de espacio posible (en este caso, 1 plancha).

1.5 SOLUCIÓN OPTIMA

Al tratarse de un problema NP-Difícil, no podemos garantizar que la solución encontrada sea óptima o no mejorable. Sin embargo, podremos establecer comparaciones en términos de una función objetivo que nos proveerá con el nivel de "bondad" de la solución.

1.6 CONCLUSIÓN

A lo largo de esta memoria expondremos las diferentes técnicas que han sido intentadas para la resolución del problema. A continuación, enunciaremos la estrategia que ha sido llevada a cabo finalmente, así como las optimizaciones aplicadas. Finalmente, expondremos y analizaremos los resultados obtenidos con varios conjuntos de prueba, en términos de utilización y tiempo de ejecución del algoritmo.

Capítulo 2 ESTRATEGIAS CANDIDATAS

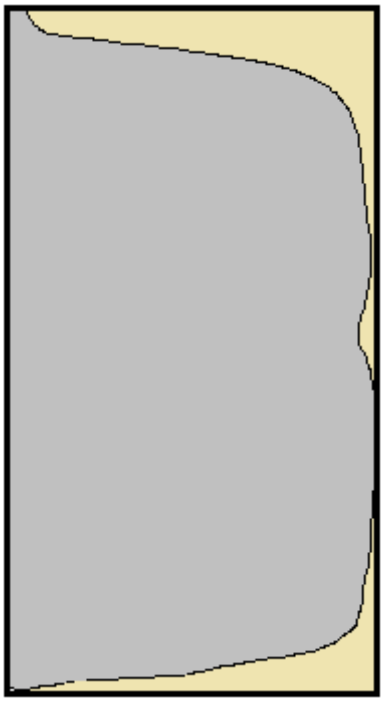
2.1 INTRODUCCIÓN

El problema del empaquetado automático de piezas se encuentra en multitud de industrias. Para la mayor parte de ellas, las piezas a empaquetar tienen forma rectangular. Es por ello que a lo largo de los últimos años se han desarrollado una gran cantidad de algoritmos de alta calidad que resuelven efectivamente la instancia del problema con piezas rectangulares. Sin embargo, el problema del empaquetado de piezas con formas arbitrarias o irregulares continúa siendo difícil de abordar: la inexistencia de alguna convención o patrón en la forma de las piezas produce que no sepamos que cálculos aplicar para garantizar una cierta efectividad a lo largo de todos los posibles sets. Ello conduce a que los algoritmos propuestos necesiten de estrategias que involucran una gran cantidad de cálculos geométricos, y por ende, tengan un alto coste computacional, independientemente de la calidad del resultado.

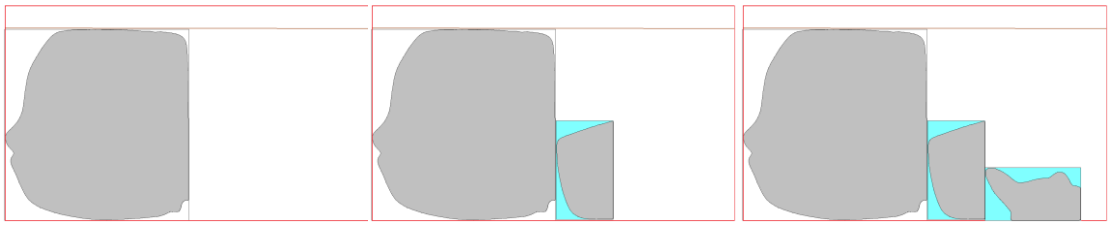
A continuación haremos un rápido resumen de algunas de las estrategias propuestas para abordar el problema del empaquetado de piezas irregulares.

2.2 ESTRATEGIA DE RECTÁNGULOS (BOUNDING BOX)

Una posible aproximación al problema es usar el rectángulo suscrito alrededor de la pieza y, a continuación, aplicar una de las estrategias de empaquetado Bounding Box conocidas (Shelf, Guillotine, Maximal Rectangles, Skyline).



Por ejemplo, el algoritmo Shelf va rellenando “estanterías” horizontales de piezas hasta llegar al tope de la plancha. A cada iteración se agrega una pieza a la estantería o se comienza una nueva encima. Así, tres iteraciones podrían ser:

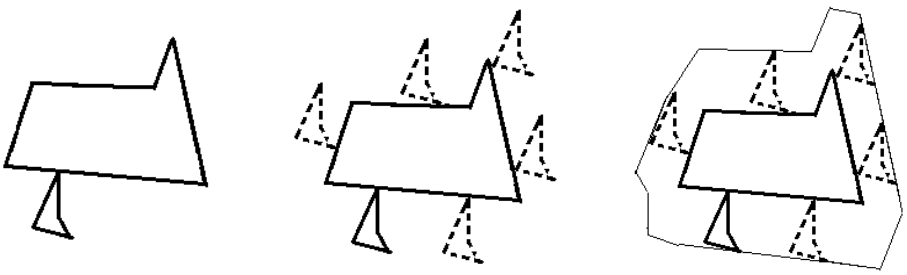


Cada vez que se coloca una pieza se pierde espacio, tanto debido a la estrategia de rectángulos en sí misma, como al particionado del espacio elegido (en este caso usando la estrategia de estanterías).

Estos algoritmos permiten obtener una solución rápida y según la aproximación utilizada, un coste de implementación bajo. Sin embargo, el uso de esta estrategia por si sola conlleva a soluciones pobres para sets de piezas altamente irregulares, debido al desaprovechamiento del espacio entre la pieza y su rectángulo suscrito.

2.3 METODO DE NON-FIT POLYGON

Método por excelencia para la resolución de problemas de empaquetado de formas irregulares, el Non-Fit Polygon se define como la forma que se obtiene de tomar dos figuras A y B, dejar A fija y hacer que B se deslice alrededor del perímetro de A, sin producir solapado.



Una vez obtenido el NFP para todo par de piezas, el algoritmo se reduce a encontrar una posición de NFP global óptima. Sin embargo, este método presenta el inconveniente del alto coste de cálculo del NFP, y la complejidad de su implementación.

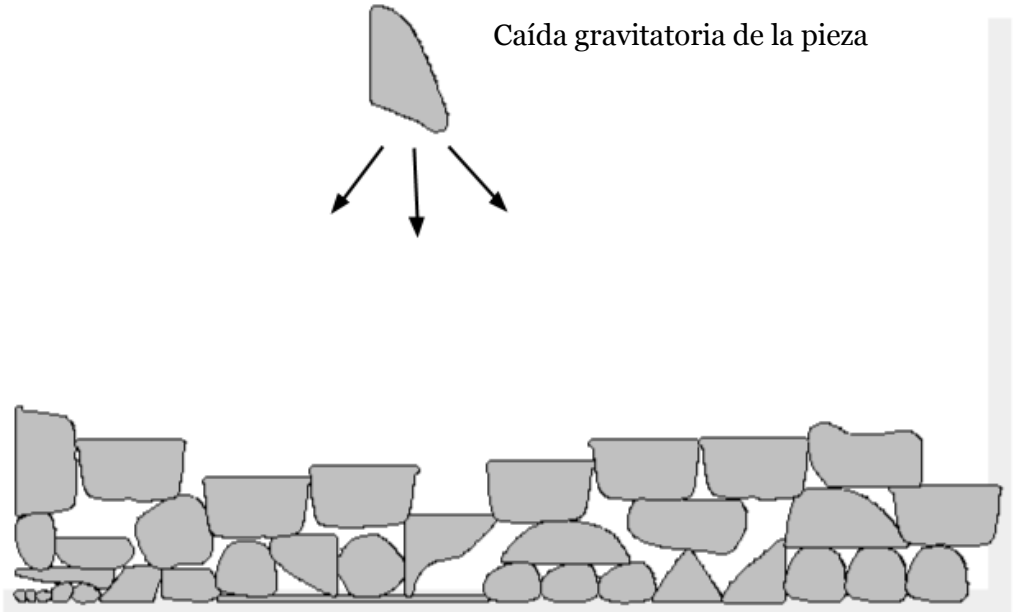
2.4 ALGORITMO GENÉTICO POSICIONAL

En este tipo de algoritmos, las coordenadas y orientación de las piezas son codificadas en un "cromosoma" del algoritmo genético. La evaluación de un cromosoma se hace utilizando diferentes parámetros, dando premios a los cromosomas que mejor se ajustan al objetivo y penalizando la violación de restricciones tales como el solapado. El algoritmo genético intenta converger hacia la población con mejor calificación.

La mejor explotación de este tipo de algoritmo se obtiene con ordenadores de gran potencia de cálculo paralelo, puesto que se pueden calcular al mismo tiempo diferentes "descendencias" o posicionamientos de las figuras, haciendo la convergencia hacia la mejor más rápida.

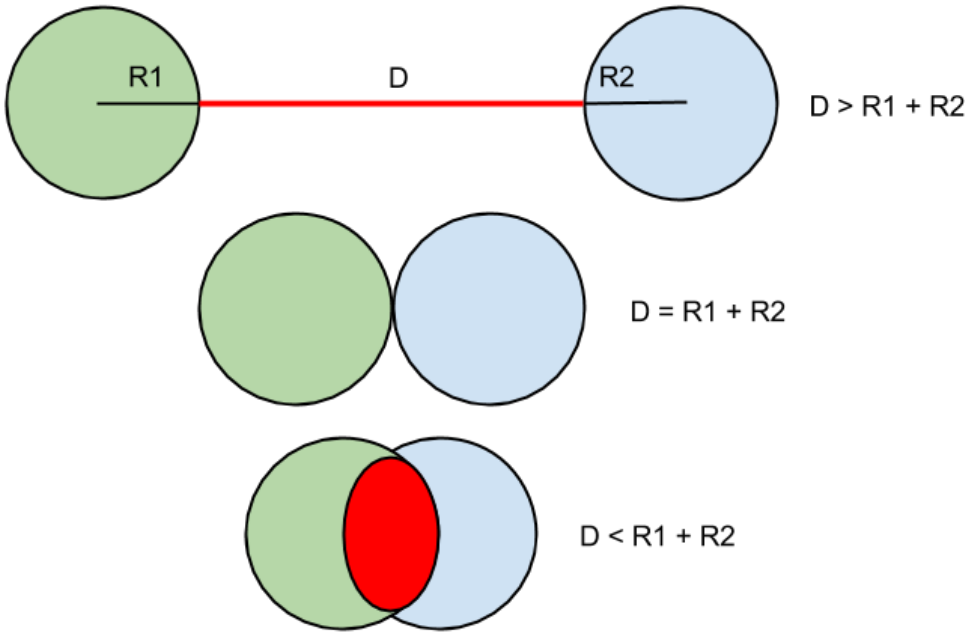
2.5 SIMULACIÓN DE SISTEMAS DE PARTÍCULAS

En este tipo de estrategias, intentamos simular un sistema donde cada pieza es una partícula. De esta manera, podemos poblar una plancha simulando colisiones entre las diferentes partículas o piezas, o haciendo simulación de fuerzas gravitatorias, entre otras. El objetivo es que el sistema (la plancha), con las fuerzas que interactúan en él, converja hacia una situación de máxima población (colocación de la mayor cantidad posible de piezas).



El principal problema que se deriva de esta aproximación es la complejidad de simular una partícula de forma arbitraria. Por ejemplo, para un conjunto de partículas

aproximadas mediante formas circulares, podemos fácilmente calcular cuando entran en contacto cualquier par de ellas, y por ende, simular con facilidad cualquier tipo de fuerza aplicada a las mismas. Lo mismo se aplica a formas relativamente regulares y conocidas (cuadrados, rectángulos, triángulos etc.).



El problema se hace prácticamente intratable cuando se agrega un número indeterminado de aristas o bordes a nuestra partícula, y que ésta no sea igual a su vecina: ello hace que los cálculos de colisiones, fuerzas y otros estén provistos de una complejidad que hace la simulación de un sistema una tarea pesada y lenta.

En general no hemos encontrado una propuesta algorítmica concreta que use esta estrategia, aunque para nuestro algoritmo se han utilizado algunas ideas provenientes de este tipo de estrategias. En el capítulo de resultados presentamos algunos experimentos realizados con estrategias que simulan caída gravitatoria de piezas.

2.6 CONCLUSIÓN

En general, cada profesional que necesita resolver el problema adapta su solución al problema particular entre manos, dado la alta cantidad de casos o restricciones que pueden venir impuestas por una industria concreta, y que pueden diferir de otra. Para este proyecto hemos usado una mezcla de aproximaciones que toman lo mejor de los métodos de Bounding Box, lo mejora según nuestro mejor criterio para este tipo de

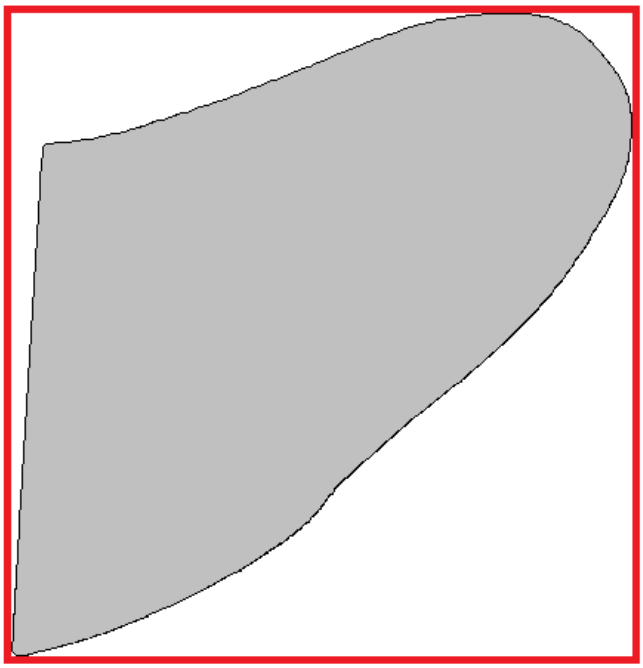
problemas, y finalmente agrega una implementación simple de compresión de piezas y de simulación de gravedad.

Capítulo 3 ALGORITMO DESARROLLADO

3.1 ALGORITMO DE RECTÁNGULOS MÁXIMOS (MAXIMAL RECTANGLES)

Para generar el primer emplazamiento de piezas partiendo de una plancha vacía utilizaremos una estrategia de ubicación basada en rectángulos. Luego, en sucesivas optimizaciones intentaremos mejorar dicho emplazamiento inicial.

Para la aplicación de dicha estrategia consideraremos el rectángulo que se puede trazar alrededor de cualquier figura geométrica. Por ejemplo, la siguiente forma será considerada en nuestro algoritmo como el rectángulo trazado a su alrededor.



Una vez calculado el rectángulo que rodea cada pieza de nuestro set, procederemos a aplicar el Algoritmo de Rectángulos Máximos. Dicho algoritmo se basa en mantener una lista de espacios vacíos rectangulares máximos en la plancha que serán utilizados y recalculados cada vez que se ubique una pieza. El algoritmo procede de la siguiente forma:

1. Partimos de una plancha vacía, inicializando la lista de espacios vacíos con un hueco rectangular de dimensiones iguales a las de la plancha;
2. Por cada pieza a ubicar:
    a. Se determina un espacio vacío donde ubicar dicha pieza y la orientación de la misma. Si no se puede ubicar dicha pieza, se pasa a la siguiente.
    b. Si la pieza puede ser ubicada, se procede a su colocación en la esquina inferior izquierda del hueco rectangular elegido.

c. Se hace la partición del espacio libre del hueco rectangular donde hemos ubicado la pieza.
d. Se recalcula la lista de huecos para obtener los rectángulos no disjuntos de dimensión máxima.

Dicho proceso se llevará a cabo hasta que no sea posible agregar más piezas, es decir, cuando no quede ningún hueco libre que pueda contener alguna pieza por ubicar de nuestro conjunto.

Conviene hacer notar que para un funcionamiento óptimo de nuestro algoritmo se intenta colocar las piezas en orden descendente respecto del área de las mismas, es decir, que intentamos ubicar las piezas de mayor a menor área.

3.1.1 DETERMINAR DÓNDE UBICAR LA PIEZA

Para ubicar una pieza procederemos a encontrar el espacio vacío que mejor se ajusta a la misma. Definiremos el mejor ajuste como aquel hueco cuyas dimensiones son las más similares a las de la pieza. Para ello procederemos de la siguiente forma:

Recorreremos la lista de huecos. Para cada hueco:

a. Determinamos si la pieza a ubicar cabe en dicho hueco con su orientación actual. Si es así, determinamos su ajuste como el mínimo entre la diferencia de ancho hueco-pieza y la diferencia de alto hueco-pieza.

$$Ajuste = \mathbf{min}(\text{hueco.}\mathbf{ANCHO} - \text{pieza.}\mathbf{ANCHO}, \text{hueco.}\mathbf{ALTO} - \text{pieza.}\mathbf{ALTO})$$

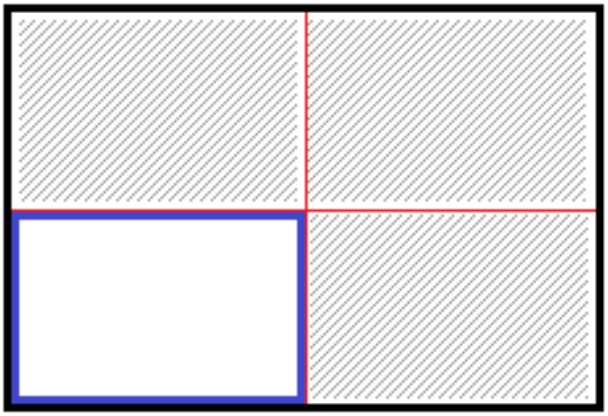
   Si el ajuste calculado supera en calidad a nuestro mejor ajuste, actualizamos nuestro mejor ajuste al hueco actual.
b. Determinamos si la pieza a ubicar cabe en el hueco actual, después de haberle aplicado una rotación de 90 grados. Si es el caso, procedemos a calcular el ajuste y actualizar, si corresponde.

Una vez hemos terminado tendremos el hueco donde se colocará la pieza, si se ha encontrado.

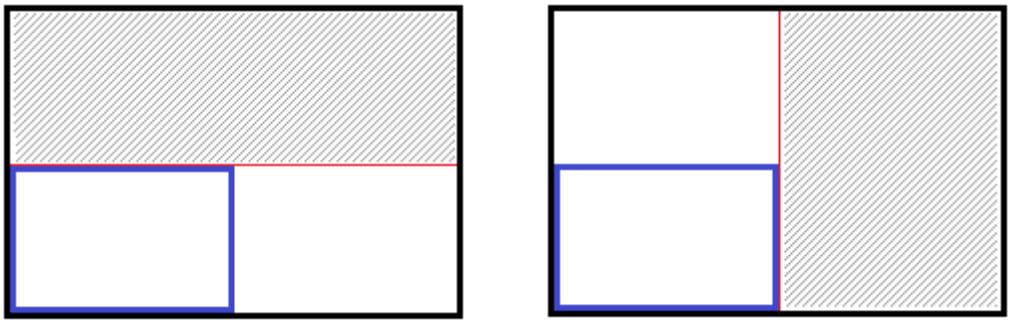
3.1.2 PARTICIONADO DE UN HUECO RECTANGULAR

Cuando hemos determinado dónde ubicar la pieza, colocamos la misma en la esquina inferior izquierda del hueco rectangular. Cabe hacer notar que la elección de la esquina donde ubicar la pieza es una mera convención: podríamos elegir colocarla en cualquier otra esquina del espacio rectangular, siempre y cuando fuéramos consistentes a todo lo largo del algoritmo. Procederemos entonces a particionar para obtener dos espacios

rectangulares máximos no disjuntos. Dichos rectángulos se obtienen extendiendo dos ejes perpendiculares de la pieza:



De esta manera tomamos los dos rectángulos que se crean mediante la intersección entre el hueco rectangular y el nuevo rectángulo producto de la extensión del eje:

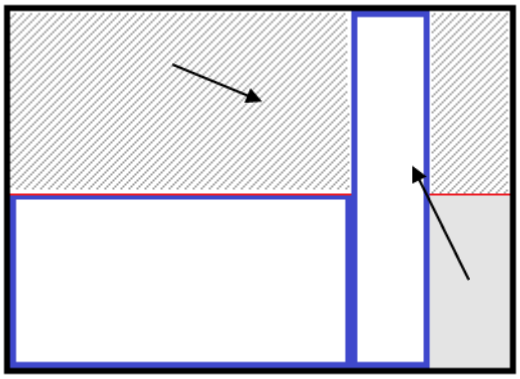


Suprimiremos entonces el hueco donde hemos ubicado nuestra pieza de la lista e introduciremos los dos nuevos huecos obtenidos después de la partición.

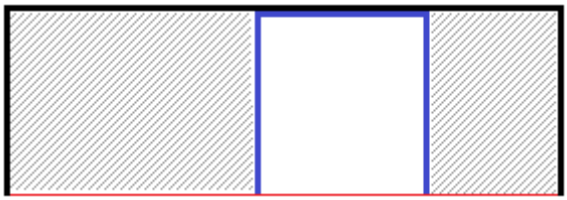
3.1.3 RE-CÁLCULO DE LOS HUECOS RECTANGULARES.

Uno de los problemas que surgen de mantener una lista de huecos no disjuntos es que ubicar una pieza puede producir que existan colisiones entre dicha pieza y espacios rectangulares que no han sido utilizados en la iteración en curso: es por ello que debemos recalcular los nuevos espacios teniendo en cuenta las posibles intersecciones con la pieza recién ubicada.

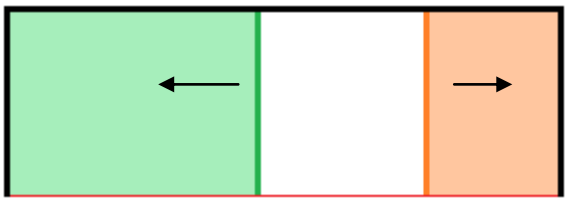
Por ejemplo, la pieza rectangular central recién ubicada entra en colisión con el espacio rectangular libre superior:



Para resolver este problema, cada vez que una nueva pieza sea ubicada, recorreremos la lista de huecos buscando aquellos que intersecten con ella. Entonces para cada intersección que encontremos aplicaremos la estrategia de partición descrita en la sección anterior, pero en este caso sobre los ejes que procedan de la figura. Tomemos por ejemplo la situación descrita en la imagen anterior, el espacio superior ya no es máximo y necesita ser recalculado. Obtenemos la intersección de la pieza con el hueco rectangular:



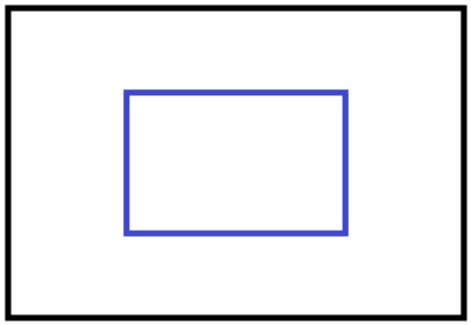
Y procedemos a particionar de la siguiente manera:



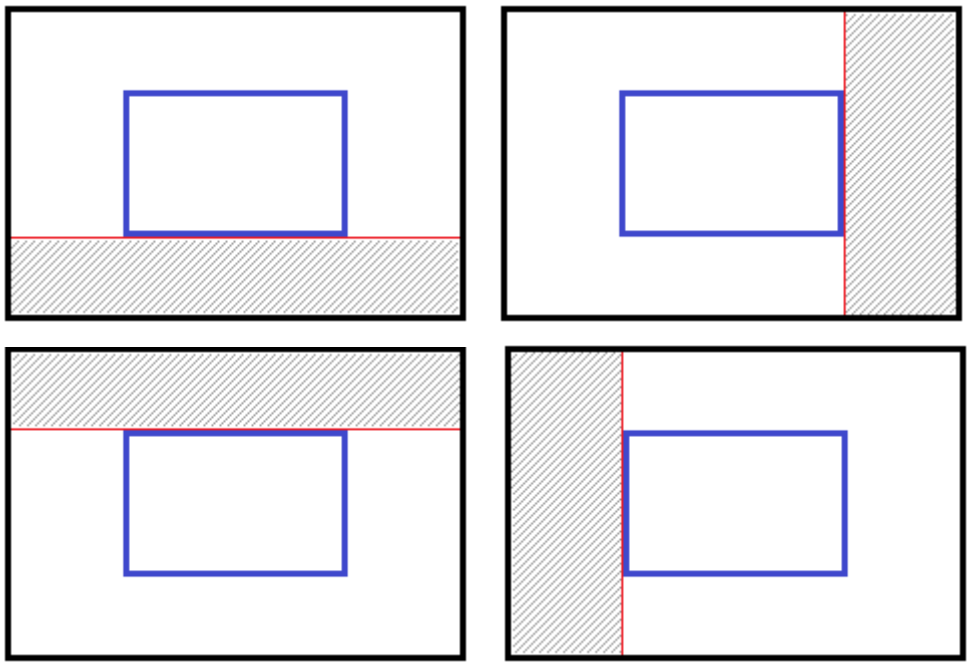
Como se puede observar, hemos obtenido dos nuevos huecos. Actualizaremos entonces nuestra lista de espacios vacíos sustrayendo el hueco que colisionaba con la pieza e introduciendo los dos nuevos obtenidos.

De una forma más general, la aplicación del particionado entre una pieza y un espacio rectangular con el que colisione podrá producir como máximo cuatro nuevos rectángulos. Téngase en cuenta que para nuestro algoritmo dicha situación es imposible dado que solo se daría si no hubiésemos ubicado correctamente nuestra pieza en el hueco. No obstante, el proceso sirve como ilustración de la manera de

proceder en cada según el caso concreto. Supongamos que la pieza intersecta con el hueco en su centro:



Entonces en este caso el particionado de la pieza con el correspondiente hueco rectangular produciría cuatro nuevos espacios:



Finalmente, cuando el conjunto de piezas es totalmente disjunto del conjunto de huecos vacíos, procederemos a eliminar de éste último todos los espacios vacíos no máximos que pueden haberse creado en esta iteración. Para ello, simplemente verificamos y eliminamos aquellos huecos rectangulares que se encuentren contenidos en otros más grandes.

3.2 MEJORA DE LA ESTRATEGIA DE RECTÁNGULOS MÁXIMOS

Para mejorar la cantidad de piezas ubicadas en una plancha y explotar al máximo la posible existencia de piezas con huecos en su interior, hemos procedido a expandir el algoritmo de Rectángulos Máximos utilizando una estrategia de barrido/sustitución. Debido a la estructura del algoritmo de Rectángulos Máximos utilizado, una vez se ha completado un primer emplazamiento de la plancha las últimas piezas ubicadas han sido las de menor tamaño. Nuestra estrategia consistirá, partiendo de las piezas en la plancha, en recolocar dichas piezas en el interior de otras, cumpliendo el doble propósito de mejorar la ocupación en la plancha y de liberar algunos espacios rectangulares para la ubicación de más piezas. La estrategia, de manera precisa, funciona de la siguiente forma:

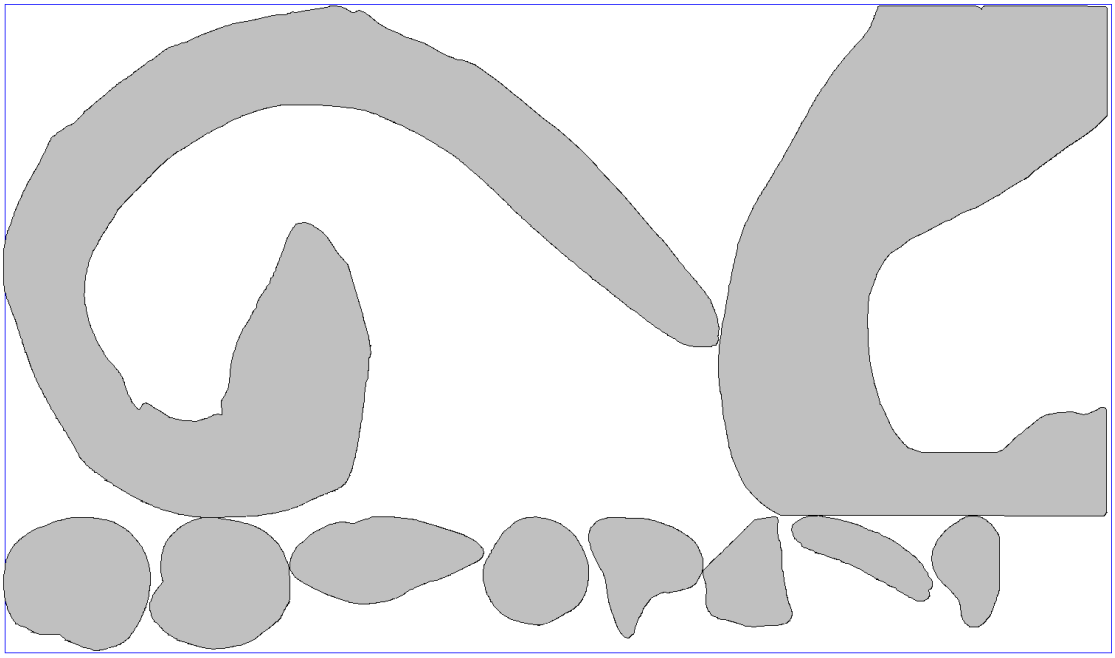
Por cada pieza ubicada en la plancha (en orden ascendente de área):

1. Intentamos ubicarla en el interior de otra más grande con un barrido o sweep a todo lo largo del interior de la potencial pieza contenedora. Es importante notar que para el barrido consideramos la figura original de la pieza, no su rectángulo exterior.
2. Si hemos tenido éxito, liberamos el espacio rectangular donde antes se encontraba la pieza desplazada, y recalculamos los huecos para asegurarnos de que todos son máximos (última etapa de la sección anterior).

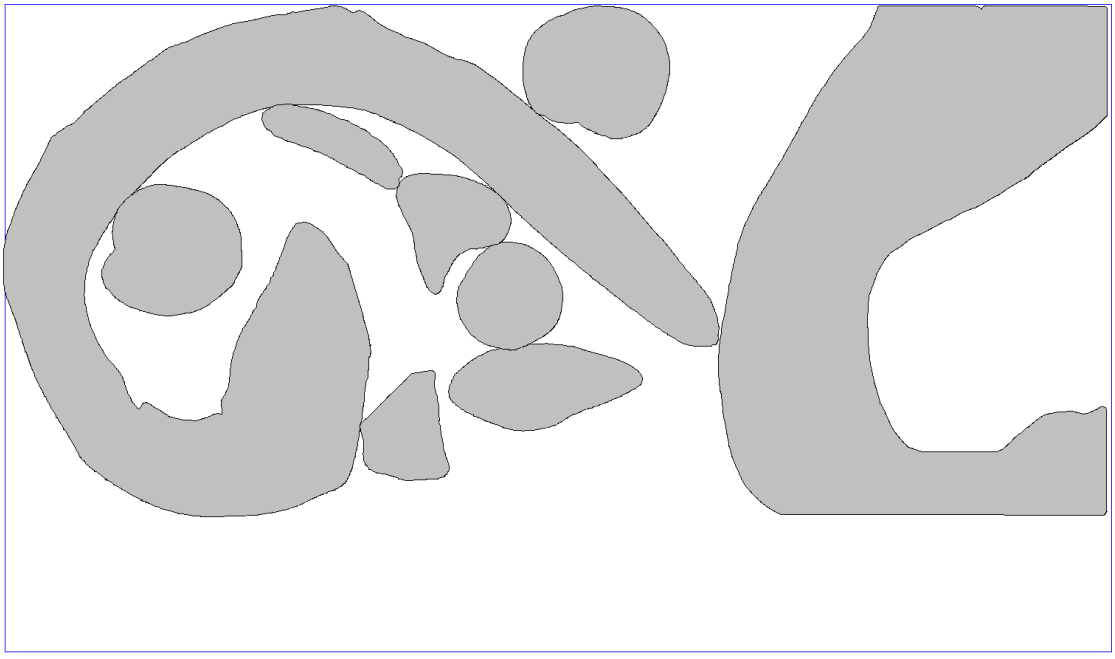
Una vez se ha completado el proceso para cada pieza ubicada, relanzamos el algoritmo de Rectángulos Máximos con las piezas no ubicadas en un intento de aprovechar los espacios recién liberados.

Todo el proceso (Barrido + relanzamiento de Rectángulos Máximos) se repetirá mientras la estrategia produzca nuevos huecos y se logren agregar más piezas a la plancha.

A modo de ilustración, tomemos una plancha justo después de haber sido rellenada con el algoritmo de rectángulos máximos, pero sin aplicarle ninguna mejora:



Como se puede observar, se desperdicia cierta cantidad de espacio debido a la forma extremadamente irregular de las piezas más grandes. Para paliar esta situación, aplicaremos la estrategia de barrido – reemplazamiento descrita anteriormente, la plancha llegando a verse siguiente forma:



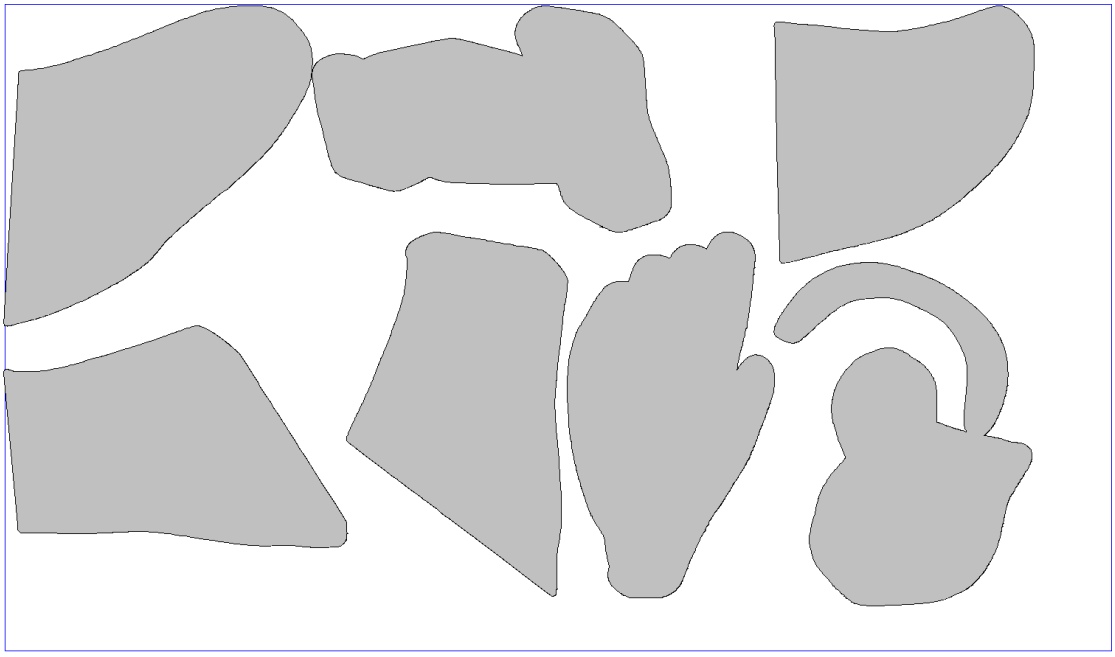
Nótese que hemos conseguido aprovechar el espacio en el interior de la forma espiral izquierda, así como liberar varios espacios rectangulares en la parte inferior de la plancha, espacio que será útil más adelante cuando procedamos a aplicar las técnicas de compresión y simulación de caída.

3.3 COMPRESIÓN DE LAS PIEZAS

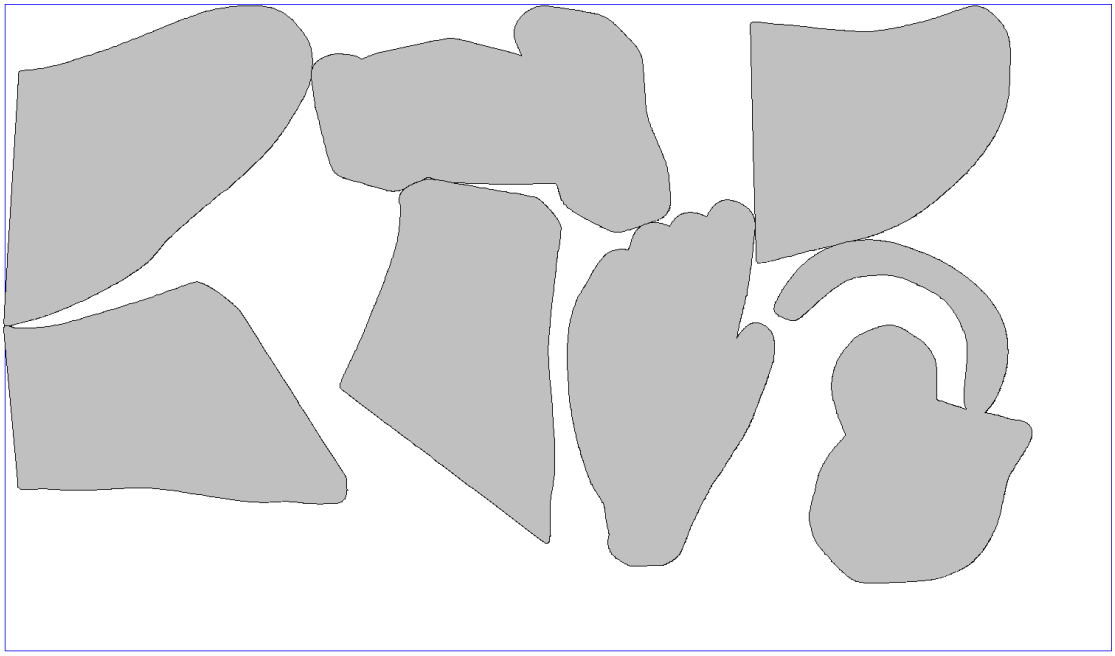
Una vez hemos aplicado con éxito la estrategia de Rectángulos Máximos mejorada, tendremos ya en la plancha una serie de piezas. Dado que durante la mayor parte de la estrategia anterior trabajamos con las aproximaciones rectangulares de las piezas, existirá espacio entre las piezas ubicadas, que será mayor cuanto más irregulares sean las mismas (mientras más irregular o hueca es la pieza, mayor será la diferencia entre su área real y el área del rectángulo que la rodea). Para paliar esta situación, y para prepararnos para la próxima fase de optimización, haremos una compresión de las piezas ubicadas en la plancha, buscando a liberar la mayor cantidad de espacio posible.

La estrategia de compresión consistirá en desplazar lo más posible todas las piezas ubicadas hacia una de las esquinas de la plancha, buscando a liberar la mayor cantidad de espacio a lo largo de los otros tres vértices. Como es habitual, no podrá existir solapado entre ningún par de piezas ya ubicada.

Así tendremos, por ejemplo, la siguiente disposición de piezas justo después de aplicar nuestra estrategia de Rectángulos Máximos:



Luego de haber comprimido el emplazamiento de piezas, tendremos:



Como se puede observar, hemos liberado una cierta cantidad de espacio a lo largo de los ejes derecho e inferior.

3.4 SIMULACIÓN SIMPLE DE CAÍDA GRAVITATORIA DE PIEZAS

Habiendo liberado cierto espacio a lo largo de los ejes, procederemos a continuación a simular una simple caída de las piezas. Si imaginamos la plancha como un recipiente abierto por la parte superior, podemos dejar "caer" cada pieza no ubicada encima de las otras y verificando si se mantiene dentro de la plancha. Asimismo, si la pieza que hemos dejado caer no ha encontrado una posición válida, podemos reintentar dicha caída rotando la pieza (para este proyecto se han considerado solo rotaciones de 90 grados).

Para cada pieza no ubicada, intentaremos varias posiciones iniciales de caída a lo largo del eje superior de la plancha, cada una considerando varios ángulos de rotación, hasta encontrar una posición válida para la pieza, o haber agotado todas las opciones.

3.5 CONCLUSIÓN

Las estrategias utilizadas en este proyecto han sido aplicadas en el orden en que han sido enunciadas, y creemos que proveen una buena flexibilidad para trabajar con conjuntos de piezas de formas o patrones variados, desde sets muy regulares, para los cuales la estrategia de Rectángulos Máximos supondrá un buen punto de partida, hasta sets altamente irregulares y con huecos, donde las estrategias de barrido, compresión y caída de piezas permiten aprovechar lo máximo posible dichas irregularidades.

También es conveniente tener en cuenta que los algoritmos usados son en su mayor parte de tipo voraz, y en consecuencia los tiempos de ejecución del algoritmo están acotados de manera razonable. Pensamos que se ha alcanzado un compromiso razonable entre la calidad de la solución obtenida y sus tiempos de obtención. En el capítulo de resultados proveeremos datos concretos que estudian relación entre el tamaño de los sets y los tiempos de ejecución alcanzados por nuestro algoritmo.

Capítulo 4 ESPECIFICACIÓN TÉCNICA

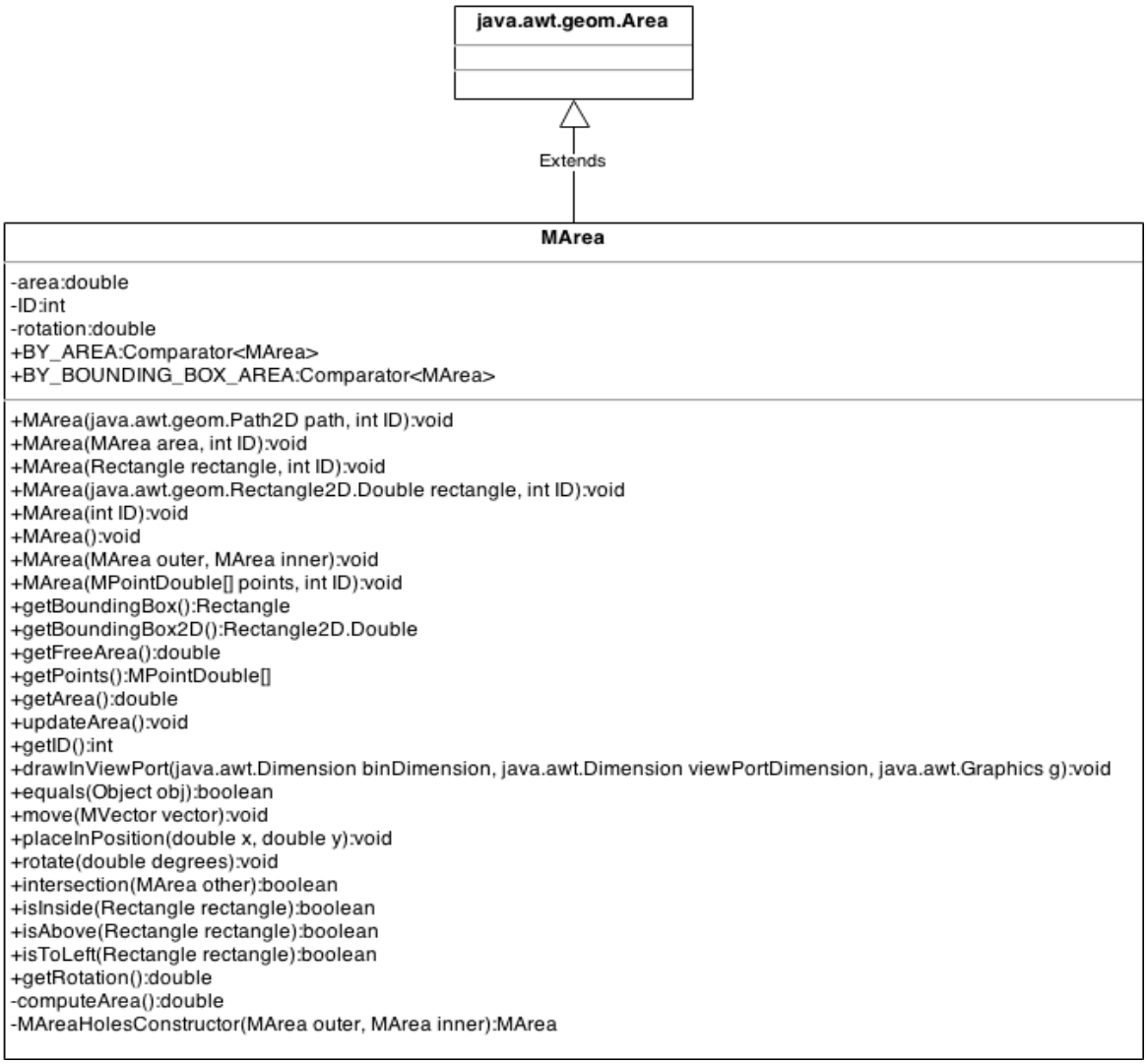
4.1 INTRODUCCIÓN

Desde un punto de vista técnico, nuestro proyecto no conlleva grandes complicaciones. Se ha optado por la utilización del paradigma Orientado Objetos por las facilidades que el mismo aporta a la concepción y desarrollo de software de fácil manutención y reutilización. En concreto se ha decidido la utilización del lenguaje de programación Java, que contiene en su núcleo de librerías herramientas para la realización del trabajo geométrico de bajo nivel, concentrándose el proyecto en el algoritmo de alto nivel y optimizaciones. Desde el punto de vista de tiempo de desarrollo, se ha elegido un lenguaje que permitiera ahorrar tiempo. Además, nos ha parecido que, dado el objetivo de poner el código del proyecto a disposición publica, la enorme popularidad de Java sería un punto positivo a favor.

La organización de paquetes de nuestro proyecto es la siguiente:

4.2 EL NÚCLEO

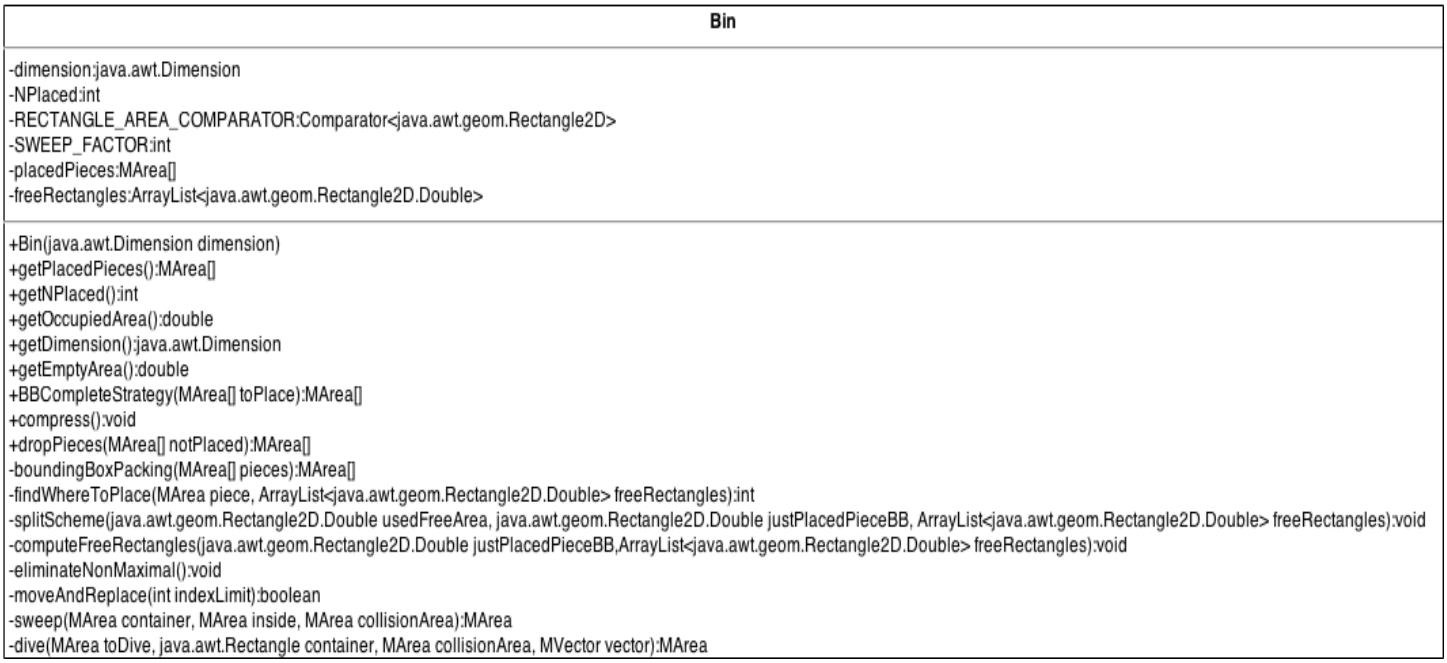
El núcleo de este proyecto cuenta principalmente con tres clases: una para la definición y manipulación de las piezas, a la que hemos llamado **MArea**, otra para la definición y manipulación de planchas, a la que hemos llamado **Bin,** y finalmente una clase **Constants** que contiene diferentes parámetros de la librería. La segunda es la contenedora de los algoritmos necesarios para la población de una plancha con piezas.



La clase **MArea** es una extensión de la clase de java java.awt.geom.Area, que provee muchas de las funcionalidades básicas asociadas al manejo de figuras. Apoyándonos en esta clase hemos ampliado sus capacidades hasta tener un mecanismo de manejo de figuras adaptado a nuestras necesidades. Como resultado, la clase MArea permite guardar, modificar, desplazar, clasificar y en general la mayor parte de interacciones con una pieza necesarias a este proyecto.

Una descripción detallada de la clase puede encontrarse a continuación:

+MArea (Path2D path, int ID)	Creación de MArea partir de un objeto Path2D.
+MArea (MArea area , int ID)	Creación de MArea usando los parámetros definidos en otra MArea. Puede pensarse en esta función como el equivalente de un constructor de copia.
+MArea (Rectangle rectangle, int ID)	Creación de MArea a partir de un objeto Rectangle.
+MArea (Rectangle2D.Double rectangle, int ID)	Creación de MArea a partir de un objeto Rectangle en doble precisión.
+MArea (int ID)	Creación de un MArea vacía, definiendo únicamente su ID.
+MArea ()	Creación de un MArea vacía y anónima.
+MArea (MArea outer, MArea inner)	Creación de un MArea hueca, definida como la diferencia del área exterior (outer) y el área interior (inner).
+MArea(MPointDouble[] points, int ID)	Creación de un MArea a partir de una colección de puntos.
+getBoundigBox():Rectangle	Devuelve el rectángulo suscrito a esta MArea.
+getBoundigBox2D():Rectangle2D	Devuelve el rectángulo suscrito a esta MArea, en doble precisión
+getFreeArea():double	Devuelve el área libre de esta MArea, computada como la diferencia entre la superficie del rectángulo suscrito y la superficie real de esta área.
+getPoints():MPointDouble[]	Devuelve los puntos del contorno que definen esta MArea.
+getArea():double	Devuelve la superficie ocupada por esta MArea.
+updateArea():void	Recalcula la superficie de esta MArea a partir de sus puntos.
+getID():int	Devuelve el ID de esta MArea.
+drawInViewPort(Dimension binDimension, Dimension viewPortDimension, Graphics g):void	Función utilitaria. Permite dibujar una pieza teniendo en cuenta la dimensión del viewport y las dimensiones reales de la plancha.
+equals(Object obj): boolean	Sobrecarga del operador de igualdad. En este caso dos MArea son iguales si sus IDs lo son.
+move(MVector vector): void	Mueve esta MArea siguiendo un vector.
+placeInPosition(double x, double y) :void	Coloca esta MArea en los puntos indicados, usando como referencia la esquina inferior izquierda de la pieza.
+rotate(double degrees):void	Rota esta MArea en grados.
+intersection(MArea other): Boolean	Calcula si esta MArea intersecta con otra.
+isInside(Rectangle rectangle): Boolean	Determina si esta MArea se encuentra completamente dentro de un rectángulo.
+isAbove(Rectangle rectangle): Boolean	Determina si el punto más al sur de esta MArea está por encima del punto más al sur del rectángulo.
+isToLeft(Rectangle rectangle): Boolean	Determina si esta MArea contiene algún punto a la izquierda del rectángulo, usando como referencia el límite oriental del rectángulo.
+getRotation() : double	Devuelve la rotación aplicada a esta plancha desde su creación.
-computeArea():double	Calcula la superficie de esta MArea a partir de sus puntos.
-MAreaHolesConstructor(MArea outer, MArea inner): MArea	Función auxiliar para la construcción de MArea con hueco.
-double area	Amacena la superficie de esta MArea.
-int ID	Almacena el ID de esta MArea.
-double rotation	Almacena la rotación de esta MArea.
+Comparator<MArea> BY_AREA	Contiene la definición de un comparador de MArea basado en superficie.
+Comparator<MArea> BY_BOUNDING_BOX_AREA	Contiene la definición de un comparador de MArea basado en la superficie de su Bounding Box.



La clase **Bin** contiene la definición de una plancha donde ubicamos las piezas. Es la abstracción de todas las operaciones que podemos realizar al colocar nuestras piezas: contiene las piezas que están ubicadas en ella, calcula los algoritmos de emplazamiento y mantiene datos relativos a ocupación, espacios libres entre otros.

Una descripción detallada de la clase puede encontrarse a continuación:

+Bin(Dimension dimension)	Creación de la plancha con las dimensiones especificadas.
+getPlacedPieces():MArea[]	Devuelve las piezas ubicadas en la plancha.
+getNPlaced() : int	Devuelve el número de piezas ubicadas en la plancha.
+getOccupiedArea():double	Devuelve la suma de las áreas de las piezas ubicadas en esta plancha.
+getDimension() : Dimension	Devuelve las dimensiones de esta plancha.
+getEmptyArea():double	Devuelve el área no utilizada en la plancha, es decir, la diferencia entre el área de la plancha y la suma de áreas de las piezas ubicadas.
+BBCompleteStrategy(MArea[] toPlace): MArea[]	Aplica la estrategia de rectángulos completa, es decir, Maximal Rectangles y su mejora de barrido. Devuelve las piezas que no pudieron ser ubicadas.
+compress():void	Aplica la estrategia de compresión sobre las piezas ubicadas en la plancha.
+dropPieces(MArea[] notPlaced): MArea[]	Aplica la estrategia simple de caída gravitatoria. Devuelve las piezas que no pudieron ser ubicadas.
-boundingBoxPacking(MArea[] pieces): MArea[]	Aplica la estrategia de Maximal Rectangles, sin ninguna mejora adicional
-findWhereToPlace(MArea piece, ArrayList<Rectangle2D.Double> freeRectangles): int	Dada una lista de huecos, determina el mejor para ubicar la pieza, si existe.
-splitScheme(Rectangle2D.Double usedFreeArea, Rectangle2D.Double justPlacedPieceBB, ArrayList<Rectangle2D.Double> freeRectangles):void	Realiza la partición del hueco donde se ha insertado una pieza.

-computeFreeRectangles(Rectangle2D.Double justPlacedPieceBB,ArrayList<Rectangle2D.Double> freeRectangles): void	Recalcula los huecos rectangulares luego de la inserción de una pieza.
-void eliminateNonMaximal():void	Elimina los espacios rectangulares no máximos que se pueden haber producido luego de la inserción de una pieza.
-moveAndReplace(int indexLimit): boolean	Aplica la estrategia de barrido. Indica si alguna pieza fue recolocada.
-sweep(MArea container, MArea inside, MArea collisionArea): MArea	Aplica el barrido de una pieza en el interior de otra. Devuelve la pieza en su nueva posición, si fue recolocada.
-dive(MArea toDive, Rectangle container, MArea collisionArea, MVector vector): MArea	Deja caer la pieza siguiendo la dirección y sentido provistos por el vector. Devuelve la pieza en su nueva posición, si existe.
- Dimension dimension	Almacena las dimensiones de la plancha.
- MArea[] placedPieces	Contiene las piezas ubicadas en esta plancha.
-int NPlaced	Contiene el número de piezas colocadas en esta plancha.
-ArrayList<Rectangle2D.Double> freeRectangles	Contiene los rectángulos huecos en la plancha.
-Comparator<Rectangle2D> RECTANGLE_AREA_COMPARATOR	Provee un comparador de rectángulos, basado en sus áreas.
- int SWEEP_FACTOR	Provee un factor de desplazamiento horizontal de las piezas cuando se está aplicando la estrategia de barrido.

La clase **Constants** contiene una serie de parámetros utilizados en los diferentes algoritmos de la librería:

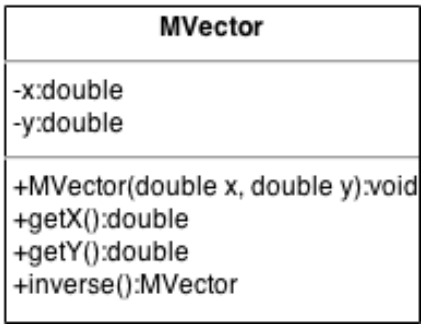
+ DIVE_HORIZONTAL_DISPLACEMENT_FACTOR	Contiene el factor aplicado al cociente de desplazamiento horizontal cuando se lanza la pieza desde diferentes posiciones en la simulación de partículas.
+ DX_SWEEP_FACTOR	Contiene el factor aplicado al cociente de desplazamiento horizontal para el algoritmo de barrido.
+ DY_SWEEP_FACTOR	Contiene el factor aplicado al cociente de desplazamiento vertical para el algoritmo de barrido.
+ ROTATION_ANGLES	Vector que contiene los ángulos de rotación de la pieza a intentar cuando se aplica el algoritmo de simulación de partículas.

Las tres clases anteriormente mencionadas contienen la principal lógica utilizada a lo largo del proyecto.

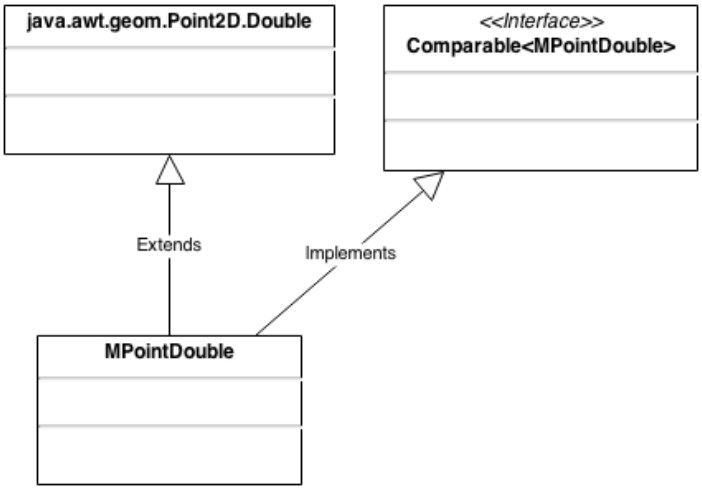
4.3 CLASES ÚTILES

Además de las clases que constituyen el núcleo del proyecto, se utilizan otras que constituyen el grupo utilitario del código.

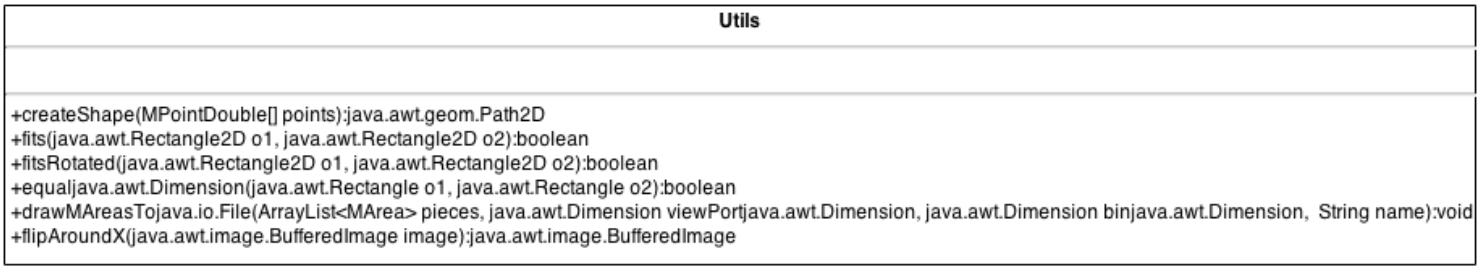
La clase **MVector** es utilizada para la definición de movimientos de piezas, y constituye la definición de un vector matemático V = (x,y).



La clase **MPointDouble** define un punto cuyas coordenadas se encuentran en doble precisión, y es la unidad básica de nuestro proyecto, ya que las piezas se definen como un set de puntos. Todas las operaciones realizadas sobre piezas necesitan la clase MPointDouble:



Finalmente, la clase de útiles varios, que permite la realización de operaciones variadas, geométricas y de otra índole:



Una descripción detallada de la clase puede encontrarse a continuación:

+ createShape(MPointDouble[] points): Path2D	Crea un objeto Path2D a partir de una colección de puntos.
+ fits(Rectangle2D o1, Rectangle2D o2): Boolean	Determina si el rectángulo o1 cabe completamente en el rectángulo o2.
+fitsRotated(Rectangle2D o1, Rectangle2D o2): Boolean	Determina si el rectángulo o1 cabe completamente en el rectángulo o2 luego de haber aplicado una rotación de 90° al primero.
+equalDimension(Rectangle o1, Rectangle o2): Boolean	Determina si ambos rectángulos tienen las mismas dimensiones.
+ drawMAreasToFile(ArrayList<MArea> pieces, Dimension viewPortDimension, Dimension binDimension, String name) : void	Vuelca una lista de MArea en una imagen, con un nombre especificado y de acuerdo con las proporciones entre el viewport y las dimensiones de una plancha.
+flipAroundX(BufferedImage image): BufferedImage	Invierte una imagen a lo largo de su eje de ordenadas.

4.4 LANZAMIENTO DE LA LIBRERÍA.

Para la puesta en marcha de la librería es necesario tener instalado el Java Runtime Environment (JDK 1.6.0_07 mínimo). Para la puesta en marcha del programa basta con ejecutar el comando:

$java –jar 2DBinPacking.jar <<archivo-de-piezas>>

El fichero de piezas debe contener una estructura precisa que se detalla a continuación:

- *Primera línea:* Dimensiones de la plancha separadas por un espacio > **x y**
- *Segunda línea:* Número de piezas contenidas en el archivo > **N**
- *Siguientes N líneas:* Una pieza por línea. Cada línea contendrá la lista de puntos que conforman el contorno de una pieza. Los puntos deberán ir separados por un espacio simple y las coordenadas de cada punto por una coma. **x1,y1 x2,y2 x3,y3 … xn,yn**

Los puntos de cada pieza deben venir dados en sentido **anti horario.**

Para especificar piezas con huecos, el procedimiento es el siguiente: definir en una línea el contorno exterior de la pieza, como se ha descrito anteriormente; en la siguiente línea especificar los puntos del contorno interior de la pieza, precedidos por @ y un espacio. En resumen la especificación de una pieza hueca será de la siguiente manera:

```
x1,y1 x2,y2 x3,y3 … xn,yn                    //contorno exterior
@x1',y1' x2',y2' x3',y3' … xn',yn'            //contorno interior
```

El programa produce varias salidas. La primera es una imagen por cada plancha utilizada, con el objetivo de proveer una idea de la disposición de las piezas por plancha una vez ejecutado el algoritmo. De la misma manera, por cada plancha se produce un

fichero de texto. En la primera línea de dicho fichero se indica el número de piezas en la plancha (N piezas); a continuación hay N líneas, cada una con 4 números:

1- ID de la pieza.
2- Rotación final aplicada a la pieza. Asumimos que esta rotación es con respecto a la disposición de la pieza como vino dada en el fichero de puntos inicial.
3- Coordenada X de la posición de la pieza (esquina inferior izquierda).
4- Coordenada Y de la posición de la pieza (esquina inferior izquierda).

Un ejemplo de fichero de salida pudiera ser:

```
%Bin-1.txt
6
1 0.0 0.0,107.0
5 90.0 1023.1914300000001,175.0
6 90.0 1433.85403,853.0
3 90.0 1433.85403,536.0
4 90.0 1434.3589,6.0
2 0.0 1723.522,82.0
```

Existen dos clases adicionales necesarias para el lanzamiento de la librería, que hacen el papel de “cliente” para nuestra aplicación:

- BinPacking.java: Se encarga del lanzamiento del cálculo de planchas, provista las piezas y dimensiones necesarias.
- Client.java: El cliente propiamente dicho. Procesa el fichero de entrada, crea la abstracción de las piezas y produce las salidas luego del cálculo.

Capítulo 5 RESULTADOS

5.1. ENTORNO DE SIMULACIÓN

Para el desarrollo y testeo de este proyecto se ha usado un ordenador con un procesador Intel ® Core ™ 2 Duo T6500 @2.10Ghz con 2.96 GB de memoria RAM utilizable. Se han contemplado varios conjuntos de piezas, la mayor parte de ellos obtenidos directamente de data sets utilizados en la industria. El proyecto toma como input la definición de las piezas en ficheros que contienen el outline de puntos de cada una, así como las dimensiones de la plancha y el número de piezas a ubicar.

Los conjuntos utilizados, en término de cantidad de piezas, se presentan a continuación:

Set	Nº Piezas
Set 0	178
Set 1	12
Set 2	22
Set 3	114
Set 4	90
Set 5	126
Set 6	131
Set 7	132
Set 8	123
Set 9	70
Set 10	105
Set 11	120
Set Rectángulos	150

Los conjuntos 0...11 son grupos de piezas reales obtenidos de los profesionales del área de Valencia y el set de rectángulos se ha producido aleatoriamente.

A continuación analizaremos los resultados del algoritmo utilizado, partiendo de una solución inicial sencilla y aplicando las diferentes mejoras desarrolladas con el objetivo de estudiar el comportamiento del programa con diferentes sets de piezas.

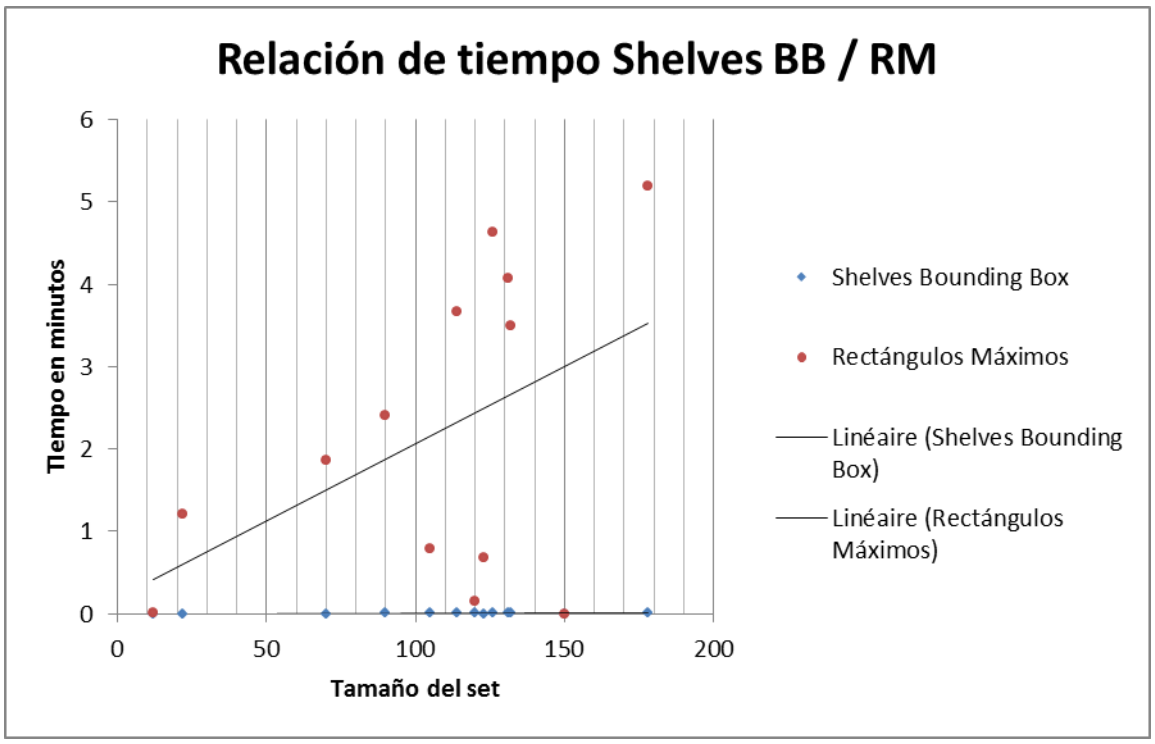
5.2. PRIMERA EVOLUCION. SHELF BOUNDING BOX – MÁXIMOS RECTANGULOS

Como punto de partida de contraste se ha utilizado un algoritmo de Bounding Box muy simple basado en la estrategia de Shelves, en la que los rectángulos son ubicados a lo largo de "shelves" o "estanterías" horizontales, partiendo desde abajo y ascendiendo a medida que se rellena una estantería. Este algoritmo es el más simple de las estrategias de empaquetado de rectángulos y el utilizado en el momento en que se decidió el desarrollo de este proyecto. Presentaremos entonces los resultados obtenidos con este algoritmo para, a continuación observar la evolución hacia la versión final de la estrategia desarrollada.

En la siguiente tabla se puede observar el número de planchas utilizadas por el algoritmo Shelf Bounding Box, en contraste con la aplicación de la estrategia de Máximos Rectángulos.

SET	Nº de Planchas obtenidas		Mejora
	Shelf Bounding Box	Rectángulos Máximos	
Set 0	24	21	**3**
Set 1	3	3	**0**
Set 2	6	5	**1**
Set 3	19	17	**2**
Set 4	16	14	**2**
Set 5	18	15	**3**
Set 6	31	23	**8**
Set 7	27	21	**6**
Set 8	40	39	**1**
Set 9	13	12	**1**
Set 10	32	31	**1**
Set 11	25	21	**4**
Set Rectángulos	43	42	**1**

Observamos que para la mayor parte de los casos de prueba se han obtenido que van desde una plancha hasta ocho, en el caso del Set 6. Si analizamos los tiempos de ejecución medios vemos que por regla general Máximos Rectángulos es más costoso, pero que en general el coste temporal no está directamente asociado con el número de piezas, sino más bien con factores varios como el tamaño medio de las piezas, su regularidad, entre otros.



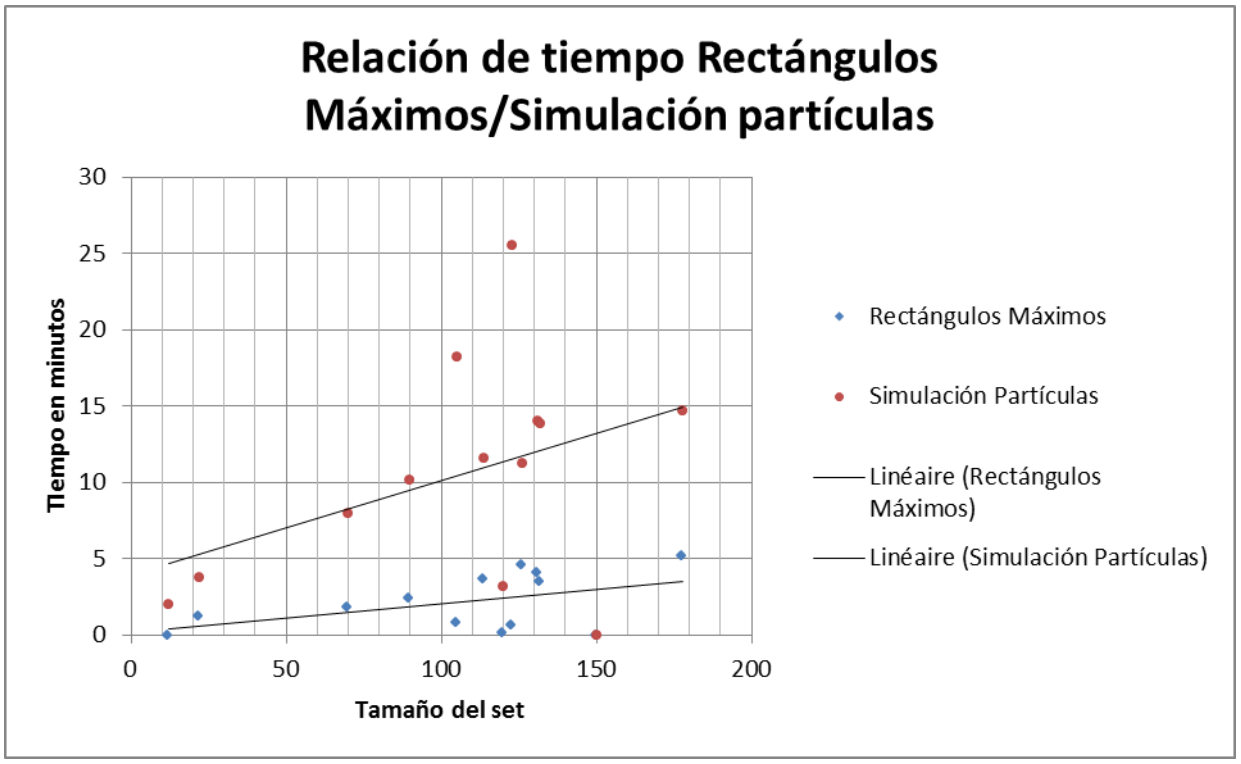
5.3. SEGUNDA EVOLUCIÓN. COMPRESIÓN Y SIMULACIÓN DE GRAVEDAD.

Como etapa final, veremos las mejoras conseguidas sobre nuestros sets de piezas después de aplicar la compresión de piezas seguido de la simulación de gravedad. En dependencia de las formas de las piezas de nuestro set, conseguiremos liberar más o menos espacio al comprimir cada plancha, lo que ayudará a obtener mejores resultados cuando dejemos caer las piezas.

En concreto, las mejoras conseguidas al aplicar esta nueva evolución son:

SET	Nº de Planchas obtenidas		Mejora
	Rectángulos Máximos	Simulación Partículas	
Set 0	21	20	**1**
Set 1	3	3	**0**
Set 2	5	5	**0**
Set 3	17	16	**1**
Set 4	14	14	**0**
Set 5	15	15	**0**
Set 6	23	22	**1**
Set 7	21	20	**1**
Set 8	39	37	**2**
Set 9	12	11	**1**
Set 10	31	30	**1**
Set 11	21	20	**1**
Set Rectángulos	42	42	**0**

Y una vez más se registra un incremento de tiempo asociado a la aplicación de la nueva técnica:

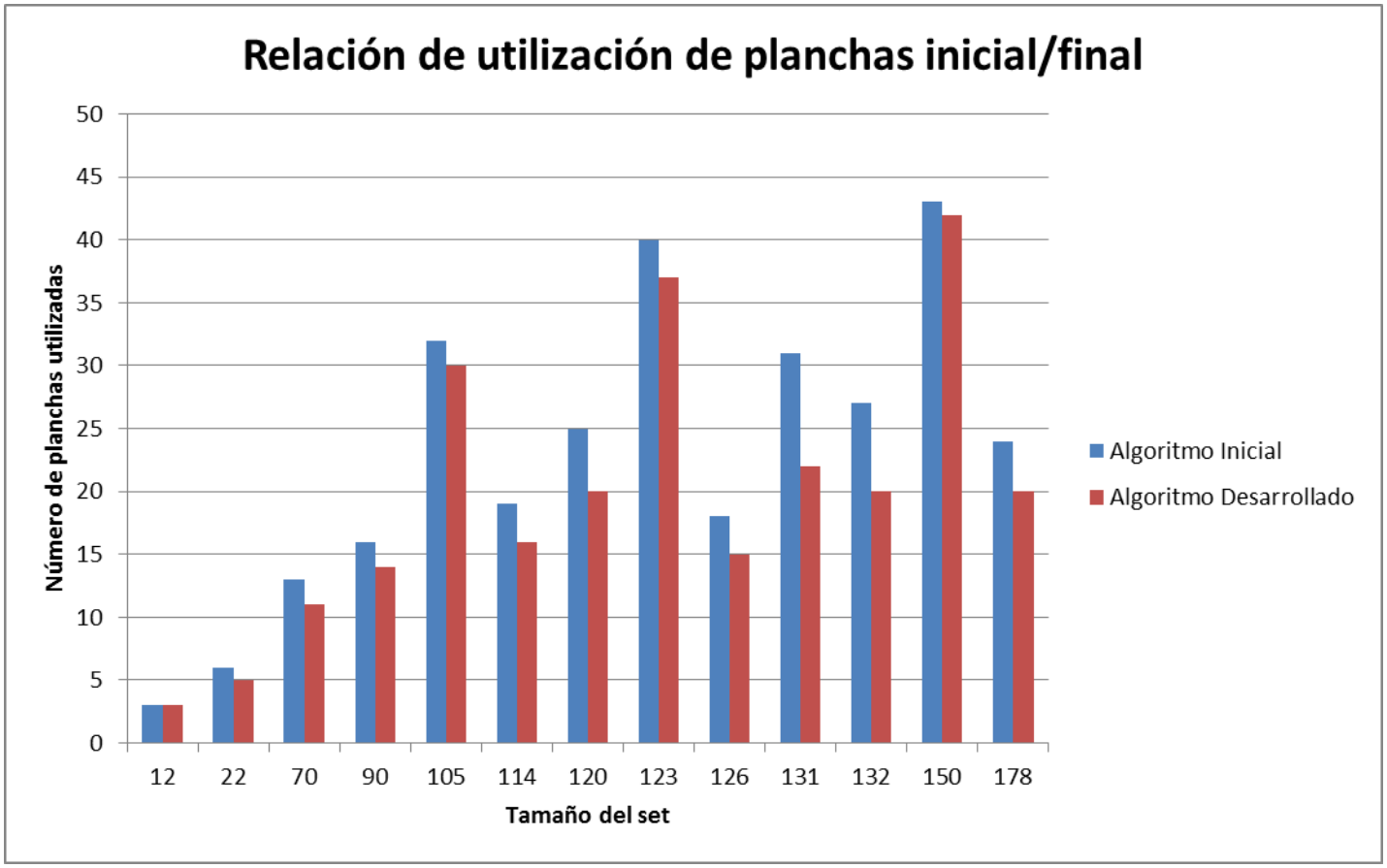


5.4. RESUMEN DE RESULTADOS

Para los juegos de piezas estudiados, el algoritmo ha producido los siguientes resultados:

Sets	Nº de Piezas	Planchas utilizadas	Tiempo (en minutos)
Set 0	178	20	14,73
Set 1	12	3	2,00
Set 2	22	5	3,74
Set 3	114	16	11,56
Set 4	90	14	10,14
Set 5	126	15	11,21
Set 6	131	22	14,03
Set 7	132	20	13,83
Set 8	123	37	25,56
Set 9	70	11	7,96
Set 10	105	30	18,26
Set 11	120	20	3,17
Set Rectángulos	150	42	0,02

Estos resultados han producido una mejora significativa respecto al algoritmo trivial utilizado inicialmente, si bien no en términos de tiempo de cómputo, en ahorro de número de planchas utilizadas para cada juego de piezas.



5.5. MEJORA DE LOS RESULTADOS.

Es posible mejorar los resultados de nuestra solución retocando algunos parámetros de la aplicación. Existen dos maneras de conseguir esto, que pueden ser usadas por separado o en conjunto: la primera, incrementando los coeficientes de desplazamiento usados en los algoritmos de barrido de piezas como parte de la estrategia de rectángulos; la segunda incluyendo más ángulos de rotación y prueba para su uso durante la estrategia de simulación de partículas.

Cuando en nuestra estrategia se aplican desplazamientos de piezas en busca de posiciones de colocación validas, los incrementos de posición vertical u horizontal se hacen como el cociente entre una dimensión de la pieza (alto o ancho) y un coeficiente fijo. Mientras más pequeño este coeficiente, más grande es el salto aplicado a la pieza cuando se desplaza, y por consecuencia menos exactitud en el desplazamiento y menos probabilidades de encontrar una posición sin solapado con el resto del conjunto. Sin embargo, si incrementamos el valor del coeficiente fijo, los saltos de desplazamientos se hacen más finos, explorando la pieza más posiciones y por lo tanto con más posibilidades de encontrar una posición valida, incluyendo aquellas en que la pieza deba caber casi exactamente. Es también una excelente manera de explotar los sets de piezas que contienen muchas piezas huecas, donde queremos aprovechar al máximo esos espacios interiores.

Durante el lanzamiento con parámetros estándar de nuestra estrategia de simulación de partículas, se intenta lanzar una pieza con dos ángulos: el ángulo inicial de la pieza (es decir aplicando una rotación de 0 grados) y 90°. Podemos indicar a nuestro algoritmo que queremos intentar más ángulos de prueba, lo que permitirá explotar particularidades de juegos de piezas especialmente difíciles (piezas oblicuas, largas etc.)

Ambas estrategias significan un aumento de tiempo de cálculo considerable, y por lo tanto deben ser aplicadas a discreción del operador de la librería. Los valores a utilizar dependerán en gran parte del tipo de set de piezas que se esté utilizando y de la potencia de cálculo de la maquina a utilizar. Se pueden probar diferentes combinaciones para intentar determinar cuál es el mejor compromiso resultados/tiempo al que se puede llegar.

A continuación presentamos los resultados obtenidos con un set de piezas de las siguientes características:

Numero de piezas	220
Ancho de la plancha	2010 mm
Alto de la plancha	1305 mm

	Algoritmo Shelf	Algoritmo desarrollado		Diferencia general
		Parametraje normal	Parametraje incrementado	
Planchas utilizadas	42	33	30	**12**
Tiempo (en minutos)	0,19	3,92	40,26	**40,06**
Coeficiente barrido horizontal	-	10	10	
Coeficiente barrido vertical	-	2	10	
Coeficiente caida gravitatoria	-	3	10	
Rotaciones	-	{0°,90°}	{0°,10°,20°,30°,40°,50°,60°,70°,80°,90°}	

Como se puede observar, con esta modificación de parámetros en particular ganamos hemos ganado 3 planchas con respecto al parametraje estándar de la librería, y encontraste hay una diferencia de tiempo acumulado de 40 minutos de cálculo.

En conclusión, la librería propone una solución inicial estándar que mejorará de manera importante un algoritmo trivial para después permitir un ajuste fino de los parámetros de manera que se obtengan los mejores resultados posibles, en función de la capacidad de cálculo del usuario y las características particulares del juego de piezas a empaquetar.

Capítulo 6 TRABAJO ADICIONAL

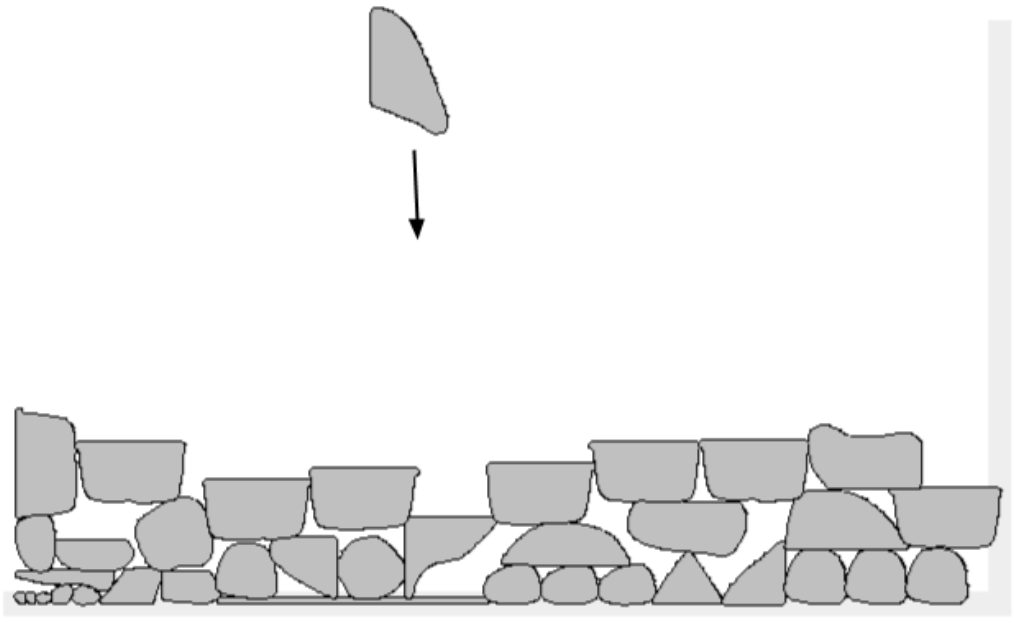
6.1 INTRODUCCIÓN.

A lo largo de este proyecto se han explorado algoritmos alternativos antes de llegar al resultado final presentado. Dichas estrategias por sí solas han presentado resultados más pobres que los conseguidos con la versión final, pero no obstante nos parece importante su mención aquí con el objetivo de ilustrar las posibles aproximaciones al problema.

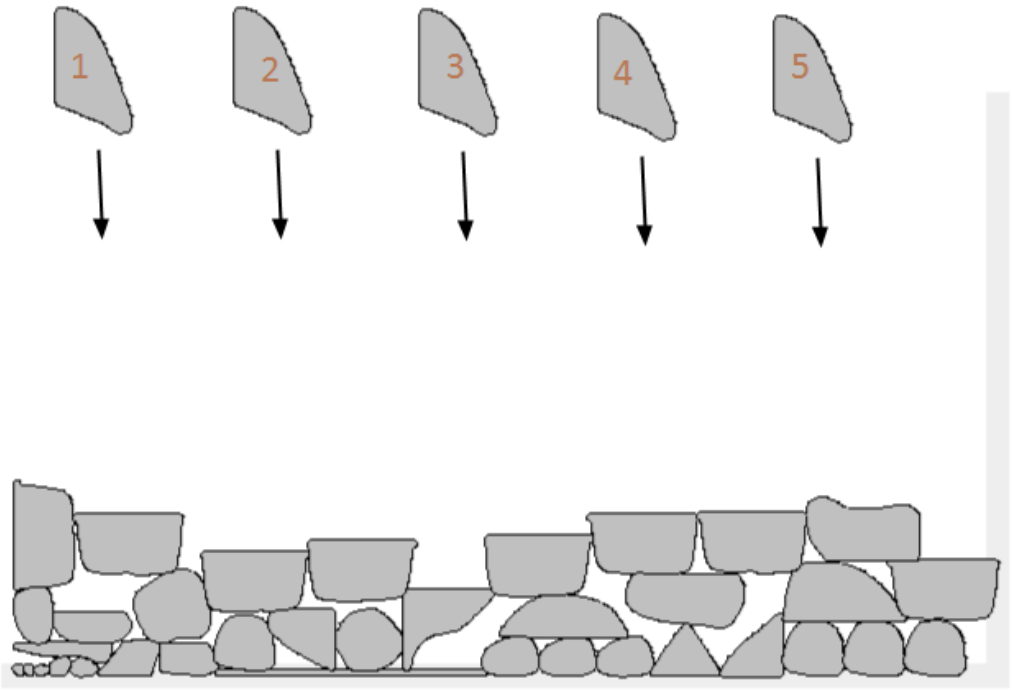
Se han intentado, en general, dos algoritmos alternativos: el primero una simulación de sistemas de partículas con gravedad; y el segundo un intento de eliminar la componente voraz de la solución intentando calcular cuales serían las mejores piezas candidatas a ubicar en una plancha. A continuación presentaremos sucintamente ambas ideas, así como los resultados obtenidos con ellas.

6.2 SISTEMAS DE PARTÍCULAS

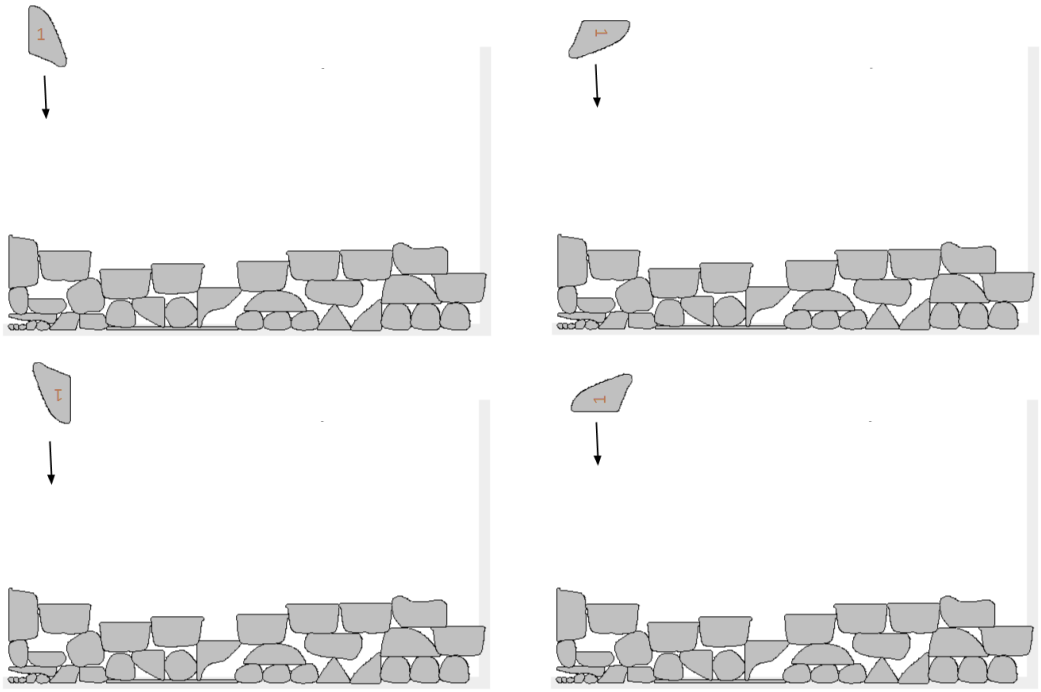
La idea consiste en pensar en una pieza como en una partícula, y en la plancha como en un recipiente bidimensional abierto en su lado superior. Se procede entonces a dejar caer cada pieza no ubicada en la plancha y aplicar algunas estrategias para encontrar la posición adecuada para dicha pieza.



El objetivo es encontrar la mejor posición para la pieza, por lo que se dejará caer la misma desde diferentes posiciones y en diferentes ángulos. En concreto, cada pieza será lanzada desde diferentes puntos iniciales:

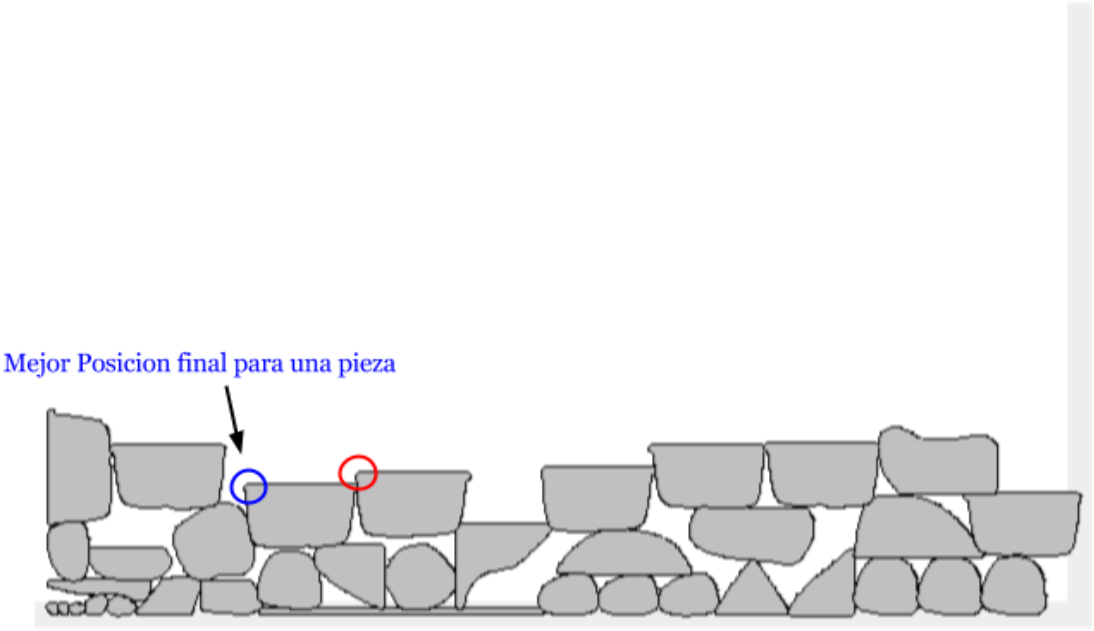


De la misma manera, por cada posición de partida de la pieza se intentarán diferentes ángulos; en el caso de nuestro experimento se han intentado 4 rotaciones: 0°,90°,180° y 270°.



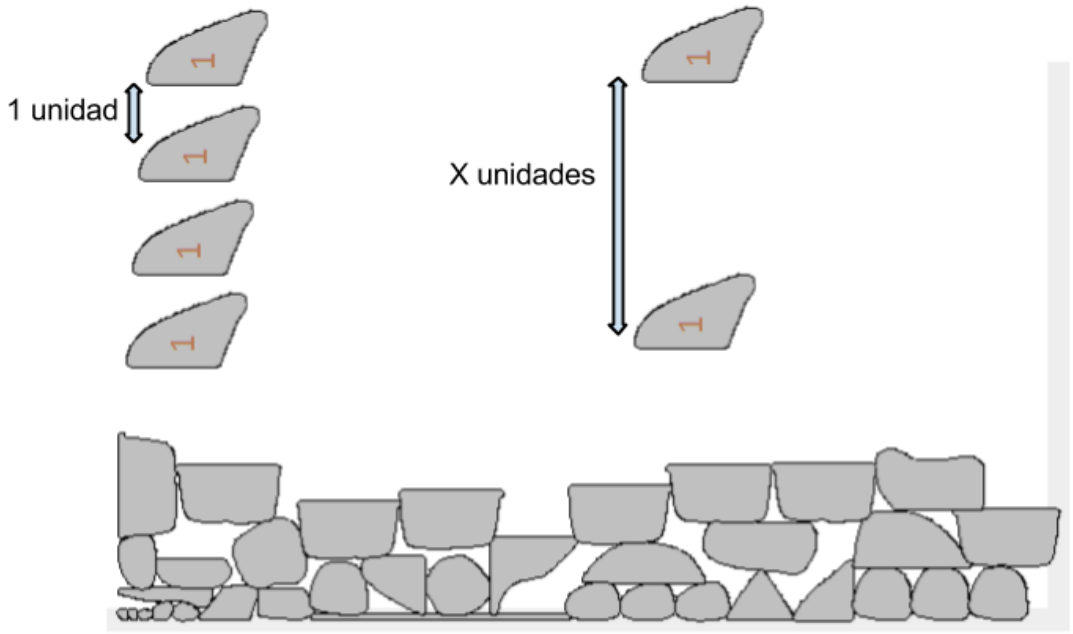
Finalmente, una vez que la pieza ha intersectado con el resto de piezas ya colocadas en el fondo de la plancha, aplicaremos un “deslizamiento” hacia el origen de coordenadas cartesiano, con el objetivo de alcanzar máxima compresión, si fuera posible.

Dadas todas las posibles posiciones finales en las que puede acabar una pieza, determinaremos la mejor de ellas como aquella posición en la que pieza lanzada está lo más cerca posible del fondo de la plancha. En concreto, será determinada por el punto de coordenada Y mínima de todos los posibles puntos finales alcanzados por una pieza (tomando como referencia en nuestro algoritmo la esquina superior izquierda del Bounding Box de la pieza). El objetivo de este proceder es doble: por un lado proveer con una estrategia de selección de posiciones y no quedarnos simplemente con la primera posición válida encontrada; y de otro intentar alcanzar la mayor cantidad de compresión posible en el fondo de la plancha.

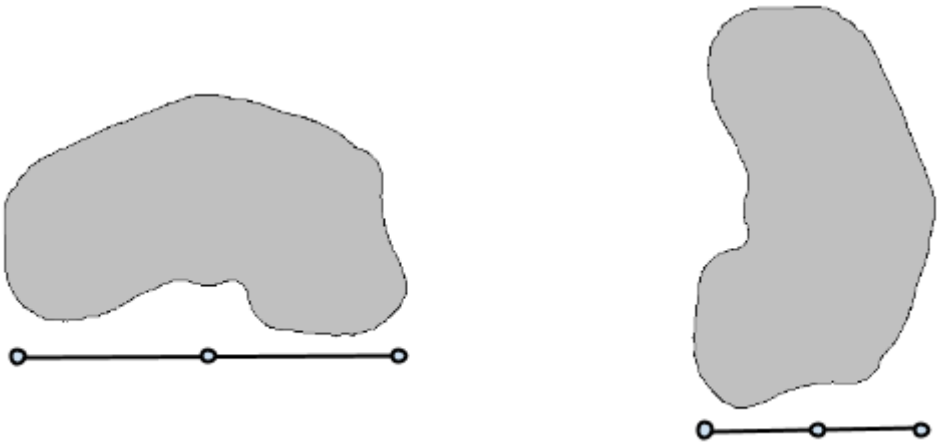


6.2.1 OPTIMIZACIONES APLICADAS

A cada instante de la caída debemos verificar si hemos intersectado con el conjunto de piezas ya ubicada o si hemos llegado al tope inferior de la plancha. Cada operación de cálculo de intersección es altamente costosa, dada la naturaleza irregular y desconocida de la pieza, y su número incrementará mientras menor sea el tamaño de la pieza lanzada. Podemos intentar paliar esta situación simulando que los intervalos de caída de la pieza sean más amplios, pero esta no es una solución viable si queremos que la caída sea suave y precisa: es decir, cada paso de la caída se consigue como la disminución de las coordenadas Y de los puntos que definen a la pieza; si disminuimos de unidad en unidad tendremos una caída suave y exacta, y si por el contrario cada descenso es de 100 unidades perderemos compresión en la posición final de la pieza.



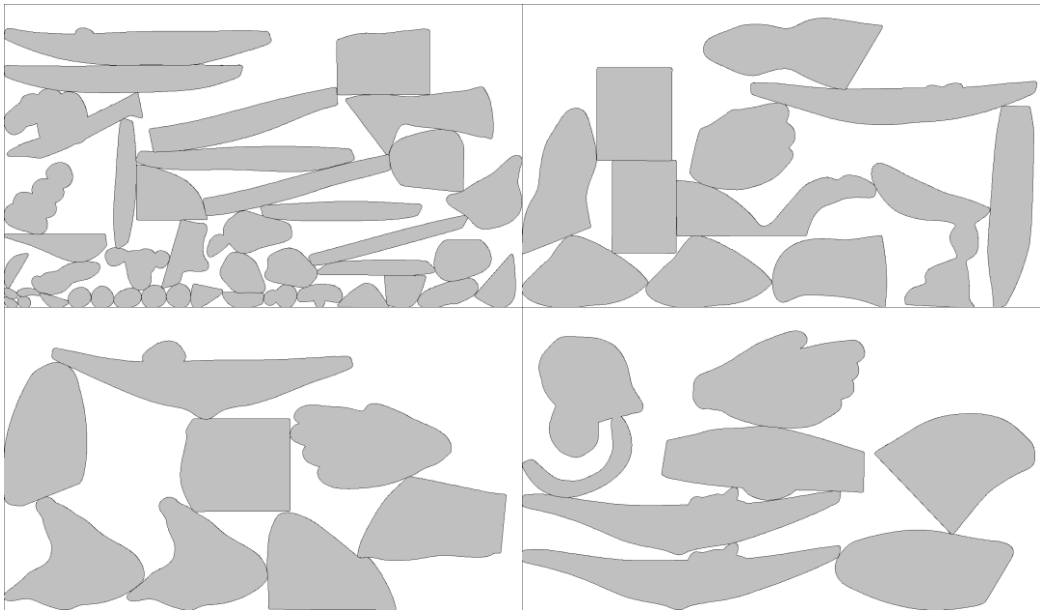
Para paliar el problema de un número excesivo de cálculos de colisión en la caída de la pieza, intentaremos disminuir la complejidad del mismo: la caída de la pieza será calculada como la caída de un segmento de 3 puntos que aproxima la parte inferior de la misma, y los cálculos de colisión se limitarán a determinar si alguno de dichos tres puntos intersecta con el set de piezas ya ubicadas.



En lugar de calcular la colisión para cada punto de la pieza, lo haremos únicamente para los 3 puntos calculados de manera simultánea, lo cual incrementará la velocidad del algoritmo de manera significativa.

6.2.2 RESULTADOS

A pesar de lo atractivo de este algoritmo, los resultados obtenidos en términos de planchas utilizadas son ligeramente inferiores a los obtenidos por la versión final presentada en este proyecto. Si elegimos lanzar las piezas en orden creciente de áreas, se obtienen planchas muy pobladas inicialmente, para luego obtener planchas ocupadas por pocas piezas de más tamaño, lo cual redunda en una utilización poco equilibrada. Véase una pequeña evolución del algoritmo, y obsérvese como va disminuyendo el equilibro de ocupación a medida que se generan planchas:



Si decidimos lanzar empezando por las piezas más grandes, se obtienen planchas mejor equilibradas a nivel de población, pero que también terminan sufriendo de inutilización del espacio a medida que van quedando menos piezas.

En conclusión, los resultados son inferiores, si bien no siempre en términos de tiempo, si en términos de utilización.

Hemos resaltado aquellos casos donde uno u otro algoritmo es mejor para nuestro sets de piezas. Como se puede observar, el algoritmo de sistemas de partículas queda en inferioridad en la mayor parte de los casos.

		Simulación Partículas		Algoritmo final	
Sets	**N° de Piezas**	**Planchas utilizadas**	**Tiempo (en minutos)**	**Planchas Utilizadas**	**Tiempo (en minutos)**
Set 0	178	20	18,473	20	14,73
Set 1	12	2	0,081	3	2,00
Set 2	22	5	2,355	5	3,74
Set 3	114	17	12,776	16	11,56
Set 4	90	14	7,660	14	10,14
Set 5	126	15	8,745	15	11,21
Set 6	131	23	6,704	22	14,03
Set 7	132	22	5,841	20	13,83
Set 8	123	38	3,493	37	25,56
Set 9	70	12	4,612	11	7,96
Set 10	105	31	7,601	30	18,26
Set 11	120	20	12,679	20	3,17
Set Rectángulos	150	44	0,127	42	0,02

6.3 ESTRATEGIA NO VORAZ

Esta estrategia ha sido intentada como un complemento para intentar mejorar el algoritmo final obtenido. El objetivo ha consistido en intentar determinar, antes de colocar las piezas en la plancha, cual conjunto de piezas de todas las que están por ubicar constituiría el set que más aprovecharía el espacio en la plancha.

Para ello se ha utilizado un algoritmo adaptado de resolución del problema de la mochila unidimensional: consideramos el área de la plancha donde colocaremos las piezas como la capacidad de la mochila; consideramos la superficie de cada pieza como su peso y valor. La función objetivo consiste entonces en conseguir el conjunto de piezas que maximiza la utilización de la plancha, de manera unidimensional (usando las áreas).

A pesar de que en términos cuantitativos se obtienen sets de piezas cuyo sumatorio de áreas se aproxima al espacio libre en la plancha, y en teoría si se lograra ubicar cada uno de estos conjuntos se obtendría el número mínimo de planchas, cuando se intentan colocar dichos sets óptimos se observa que las formas de las piezas no permiten que se ubique todo el grupo. Esto redunda en que a medida que se van utilizando planchas, se van acumulando aquellas piezas que no pudieron ser ubicadas con el resto de su conjunto óptimo, por lo que al final el performance del algoritmo termina siendo

extremadamente pobre en términos de ocupación, y de manera general lento si se le relaciona con los resultados que se obtienen.

En conclusión, intentar determinar el mejor conjunto aplicando solo cálculos unidimensionales no se ajusta correctamente a la realidad de las formas de las piezas en el momento de ubicarlas en la plancha.

Capítulo 7 CONCLUSIÓN

Este proyecto propone un algoritmo simple para el empaquetado automático de piezas de forma arbitraria. La arbitrariedad de formas en las piezas hace que encontrar un algoritmo óptimo y eficiente sea extremadamente difícil, por lo que se debe llegar a un nivel de compromiso basado en los requerimientos aplicativos de una solución. En nuestro caso, hemos intentado proponer una aproximación al problema que se mantenga razonable en términos de tiempo de ejecución, proveyendo al mismo tiempo una solución de calidad aceptable.

Hemos estudiado algunos de los algoritmos más utilizados hoy en día, y de ellos hemos extraído un algoritmo ajustado a nuestras necesidades y objetivos. A continuación hemos estudiado paso a paso los resultados obtenidos por nuestra propuesta y visto la mejora respecto de la solución inicial existente.

Finalmente, esperamos haber cumplido nuestro objetivo inicial de poner a disposición de cualquier profesional una solución codificada y optimizada que pueda servir como punto de partida rápido y fiable para la resolución de problemas de empaquetado automático.

BIBLIOGRAFÍA

[1] **A two-level search algorithm for 2D rectangular packing problem**. Mao Chen, Wenqi Huang. School of Computer Science, Huazhong University of Science and Technology. 18 April 2007

[2] **Nesting of two-dimensional irregular parts: an integrated approach.** S. Q. XIE, G. G. WANG and Y. LIU.Department of Mechanical Engineering, University of Auckland. Auckland, New Zealand. Department of Industrial and Manufacturing Engineering, The University of Manitoba, Canada

[3] **Solving 2d-strip packing problem using genetic algorithm.** Zahid Hossain. 28/08/2006, Dhaka,Bangladesh. Project report submitted to the Department of Computer Science

[4] **A Thousand Ways to Pack the Bin – A Practical Approach to Two-Dimensional Rectangle Bin Packing.** Jukka Jylänki. February 27, 2010

[5] **Nesting Problems and Steiner Tree Problems.** Benny Kjær Nielsen. University of Copenhagen. November, 2007.

[6] **NFP-based Nesting Algorithm for Irregular Shapes.** Liu Hu Yao - Department of Computer Science and Technology,Shanghai Jiaotong University. He Yuan Jun, Department of Computer Science and Technology, Shanghai Jiaotong University P.R.China