Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Sistema para la optimización del corte en línea de float

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Vicente Adriá Bohigues

Tutor: Federico Barber Sanchis

2015-2016

Resumen

En este TFG se aborda el problema real del procesado de materia prima, desde un origen en forma de material continuo o bobinas gigantes hasta un formato más maleable en forma de láminas rectangulares. Ejemplos de estas casuísticas son el corte de láminas de vidrio (que da nombre a este trabajo) o el corte de bobinas de cartón, en los que al final del proceso se obtienen piezas rectangulares que podrían ser servidas a clientes finales o formar parte de un proceso intermedio en el que se utilizaran para tratar el material de forma más práctica. A lo largo del trabajo se plantea un pequeño repaso de las tipologías de soluciones utilizadas hasta el momento, se especifica el problema y se plantea una metaheurística basada en algoritmos genéticos para su resolución. Para ello, se ha desarrollado un algoritmo en C# que a través de una interfaz gráfica es capaz de abordar múltiples definiciones del problema de entrada y calcular progresivamente mejores soluciones. El desarrollo de este trabajo ha permitido ahondar en los conocimientos actuales de algoritmos evolutivos así como su aplicación práctica en situaciones reales. Los resultados obtenidos corroboran la idoneidad de esta tipología de algoritmos y demuestran que la metaheurística desarrollada es aplicable a entornos reales con un alto factor de reutilización en procesos más complejos.

Palabras clave: metaheurística, algoritmos genéticos, strip packing problema, c#, vidrio float, corte de cartón.

Abstract

This DFW addresses the real problem of raw material processing from a source in the form of continuous material or giant coils to a softer form as rectangular sheets. Examples of these caseloads are cutting glass sheets (which gives this work tittle) or cutting cardboard reels, in which the end of the process rectangular pieces that could be served to end customers or be part of an intermediate process in which were used to treat the material in a more practical way. Throughout the work a little review of the types of solutions used until now, the problem is specified and metaheuristic based on genetic algorithms raises its resolution. For this, we have developed an algorithm in C# through a graphical interface is capable of addressing multiple definitions of the problem input and progressively calculate better solutions. the development of this work has helped deepen current knowledge of evolutionary algorithms and their practical application in real situations. the results confirm the suitability of this type of algorithms and show that the metaheuristic developed is applicable to real environments with high reuse factor in more complex processes.

Keywords : metaheuristics , genetic algorithms , strip packing problem , C#, float glass , cut cardboard .

Tabla de contenidos

1. Introducción

Desde la segunda mitad del siglo XX, la evolución de la computación ha permitido mejorar procesos que hasta el momento se realizaban de forma manual. En cada ámbito donde se aplica la computación, se mejora notablemente el rendimiento de los procesos afectados. Derivado de esta situación, existen hoy en día, multitud de profesiones de las que hace tan solo 20 años, ni los autores de ciencia-ficción fueron capaces de predecir.

Actualmente nos encontramos ante una nueva revolución en el campo de la computación, y esta es la aplicación de algoritmos de Inteligencia Artificial tanto a procesos computarizados previamente como nuevos. Unida a esta revolución, se están mejorando las capacidades de cálculo de los ordenadores a pasos de gigante.

Nos encontramos por tanto en un momento en el que revisar procesos que llevan muchos años funcionando para ver de qué manera se pueden mejorar con la aplicación de algoritmos inteligentes. Este es el caso de este TFG.

En este TFG vamos a abordar problemas reales de procesado de materia prima, desde una forma primaria en forma de material continuo o bobinas gigantes de material hasta una forma más maleable en forma de láminas rectangulares. Ejemplos de estas casuísticas son el corte de láminas de vidrio (que da nombre a este TFG) o el corte de bobinas de cartón, en los que al final del proceso se obtienen piezas rectangulares que podrían ser servidas a clientes finales o formar parte de un proceso intermedio en el que se utilizaran para tratar el material de forma más práctica.

Se plantea en este TFG un pequeño repaso de las tipologías de soluciones utilizadas hasta el momento y se presentara una algoritmo, propio de otra tipología de problemas bastante similar. Adaptaremos dicho algoritmo a nuestra situación particular y realizaremos una implementación en un lenguaje orientado a objetos que nos permitirá realizar las pruebas para concluir si ha sido una buena elección o no.

En el punto 2 se concreta y define el problema que tratamos de solucionar. En el punto 3, Se diseña el algoritmo de referencia, adaptado a nuestro problema particular. En el punto 4 se explica cómo se ha diseñado e implementado el algoritmo en un lenguaje de programación real. En el punto 5 se explica el programa desarrollado que permite interactuar con el algoritmo. En el punto 6, se plantean un conjunto de pruebas para discriminar los aspectos relevantes del algoritmo. En el punto 7 se presentan las ideas de mejora que han surgido durante la elaboración de este TFG y las conclusiones finales del trabajo. En el punto 8 se relacionan las referencias utilizadas para la elaboración de este proyecto. Y por último en el Anexo I, se muestra el detalle completo del diseño de las clases y la declaración/explicación de sus variables, propiedades y funciones.

2. Definición del problema

En esta sección introduciremos el problema concreto que nos enmarca el TFG, desde la perspectiva histórica que dio origen al problema general, hasta la concreción del problema que se trata de resolver en este trabajo. Una vez concretado el problema, se definirá un marco de trabajo cuyos límites servirán para modelar el problema de forma que sea abordable mediante un algoritmo informático.

2.1. Problema histórico

En los procesos industriales, existen determinadas casuísticas en las que la fabricación de un material se hace a gran escala y una vez producido, éste, debe dividirse para obtener conjuntos más pequeños de dicho material para ser aplicable en una parte más concreta del proceso.

Aunque esta descripción es aplicable a todo tipo de materiales, se debe diferenciar la naturaleza del material en función de su dimensionalidad, es decir, no es lo mismo procesar cable (1 dimensión), que láminas de acero (2 dimensiones) que un tronco de árbol (3 dimensiones).

En la industria del cartón, plástico, metal y vidrio (todas ellas con un modelo bi-dimensional) se da esta situación especialmente ya que normalmente se produce el material de forma continua para ser posteriormente almacenado en forma de bobinas o láminas. Este subproducto, se utiliza como materia prima de un proceso más refinado en el cual mediante cortes se obtiene el producto especificado, mucho más reducido y que es aplicable en otros procesos de fabricación o directamente usable por el cliente final.

Este proceso se puede considerar como un proceso de refinamiento desde la fabricación de la materia básica hasta obtener un producto final para el mercado de consumo, pasando por múltiples etapas de procesamiento y refinado en el que cada subproducto obtenido se utiliza como entrada en la siguiente etapa.

Este proceso, tiene asociado una pérdida de material en forma de merma o material no aprovechable al “sobrar” material en cada proceso que no sirve para conformar el subproducto deseado. En algunos casos ese material no aprovechable se recicla y vuelve a formar parte de la materia prima, pero sea cual sea el caso, esta merma se traduce en una pérdida del rendimiento del proceso y por tanto en pérdidas económicas de la industria.

Principalmente esta merma viene determinada por las diferencias entre las distintas maquinarias que efectúan las distintas etapas del proceso, que suelen ser cada vez más pequeñas y adaptadas al objetivo final. Además en función del material y del uso previsto existen otros condicionantes que pueden afectar a la merma como pueden ser las calidades o restricciones en el proceso de corte (guillotinas, orientación…)

A grandes rasgos se puede dividir en dos las situaciones de fabricación, el procesado de material continuo (ej.: la fabricación de vidrio en lámina continua) del cual se obtienen láminas no enrollables y el procesado de material en bobinas, las cuales se podrían considerar como una lámina muy larga y enrollada sobre sí misma. A continuación se muestra una pequeña tabla en la que se ejemplifican algunos de los usos

	Formato de la materia prima	
Material	**Bobinas**	**Láminas/Material continuo**
Papel	Papel continuo, Materia prima para folios…	---
Cartón	Fabricación de embalajes	---
Plástico	Fabricación de embalajes, Film…	Toldos, Cubiertas protectoras…
Vidrio	---	Ventanas, Puertas…
Metal	Acero, Piezas troqueladas, cables…	Grandes piezas para barcos, Estructuras…
Textil	Rollos de tela	---

2.2. El corte de láminas de vidrio en línea FLOAT

La utilización masiva del vidrio en la Arquitectura moderna y en el diseño de interiores más actual es posible gracias a la existencia del vidrio flotado. Un vidrio versátil, de gran calidad y durabilidad muy diferente al conocido como “vidrio estirado” que se fabricaba en España hasta los años 80.

El vidrio flotado es el resultado de los últimos avances tecnológicos en fabricación de vidrio. Un sistema que se basa en el principio físico de un líquido que flota sobre otro de mayor densidad y en que las superficies superior e inferior del primero son perfectamente horizontales y paralelas. A esto se le denomina "planimetría" del vidrio y es la diferencia más visible entre el vidrio estirado y el flotado.

El vidrio está compuesto en un 75% de arena de sílice, aunque no es éste el único componente del vidrio. La sosa y la caliza son, entre otras, las materias primas utilizadas en el proceso de fabricación del vidrio. Todas estas materias primas se funden en un horno que alcanza los 1600 grados de temperatura, que alberga hasta 1800 toneladas de vidrio fundido en su interior y que no cesa su actividad en aproximadamente 17 años.

Una vez fundido, el vidrio líquido se vierte a 1.100 grados sobre una gran piscina de estaño fundido de 170 toneladas y flota sobre él dado que su densidad es menor que la del estaño. A su salida el vidrio tiene una temperatura de 600 º C y se debe enfriar de una forma controlada y paulatina hasta que esté preparado para ser cortado transversal y longitudinalmente. Este enfriamiento controlado es clave en todo el proceso para evitar las tensiones internas que harían imposible el corte normal de la hoja.

Tras este proceso el vidrio pasa por una zona de inspección con láser y a continuación se corta longitudinalmente y transversalmente.

Finalmente, una vez cortado se evacua de la línea con diferentes medios en función del tamaño de la hoja.

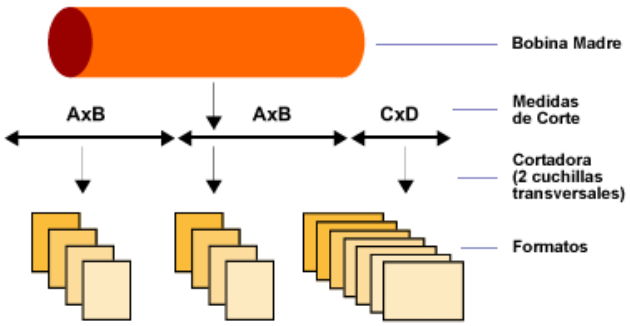
Láminas de vidrio ya cortadas.

Es este último punto el que nos interesa abordar en este TFG, ya que el proceso de corte es un factor determinante en cuanto al aprovechamiento del material que sale del horno. Así pues, una buena optimización en el proceso de planificación puede ahorrar numerosos costes, minimizando la merma, reduciendo las operaciones de manipulación o planificando temporalmente la producción de los pedidos reduciendo el tiempo de almacenaje, por poner algunos ejemplos.

Para el problema de corte de láminas de vidrio hay ciertas limitaciones que se deben considerar principalmente, como por ejemplo la anchura máxima de las piezas obtenidas, el número de cuchillas y guillotinas de corte que posee la máquina y finalmente, su disposición/orientación. Además se podría considerar que puede haber defectos en el material, que solo se va a poder procesar un número máximo de pedidos iniciados en paralelo, el tiempo necesario para la realización de los cortes...

2.3. El corte de bobinas de cartón

Otro ejemplo aplicable a este TFG, es la industria del cartón y el tratamiento de las bobinas como materia prima para obtener pequeñas láminas rectangulares que luego se podrían utilizar por ejemplo para troquelar cajas de embalaje.



En el caso del corte de bobinas de cartón, se añadirían a las restricciones que hemos visto para el vidrio, otras como que hay ciertas tipologías de cartón que no admiten el cambio de orientación de las piezas u otras consideraciones que es más fácil obviar y no tener en cuenta como pueden ser los tiempos de recambio de las bobinas origen, el almacenamiento del producto antes y después del proceso, etc

Ejemplo de maquina real de corte de cartón

Ejemplo de maquina real de corte de cartón

2.4. Delimitando y concretando el problema a tratar en este TFG

Para nuestro proyecto tomaremos un ejemplo real de varios problemas de corte, concretamente el corte de pedidos de láminas de vidrio sobre una lámina continua y el corte de bobinas de cartón con un ancho fijo y una longitud a priori infinita.

En el caso del cartón, se debe considerar la restricción de orientación. Al tratarse de material que no tiene la misma estructura/consistencia en ambas componentes ancho/alto los pedidos deben tratarse con la orientación exacta que se indique ya que el resultado en distintas orientaciones no es el mismo.

Habitualmente en estos problemas se introduce la restricción de la guillotina, pero a diferencia de los problemas de carga en contenedores en los cuales se hacen "capas" de ítems, en este caso sabemos que las máquinas por las que discurre el material constan de:

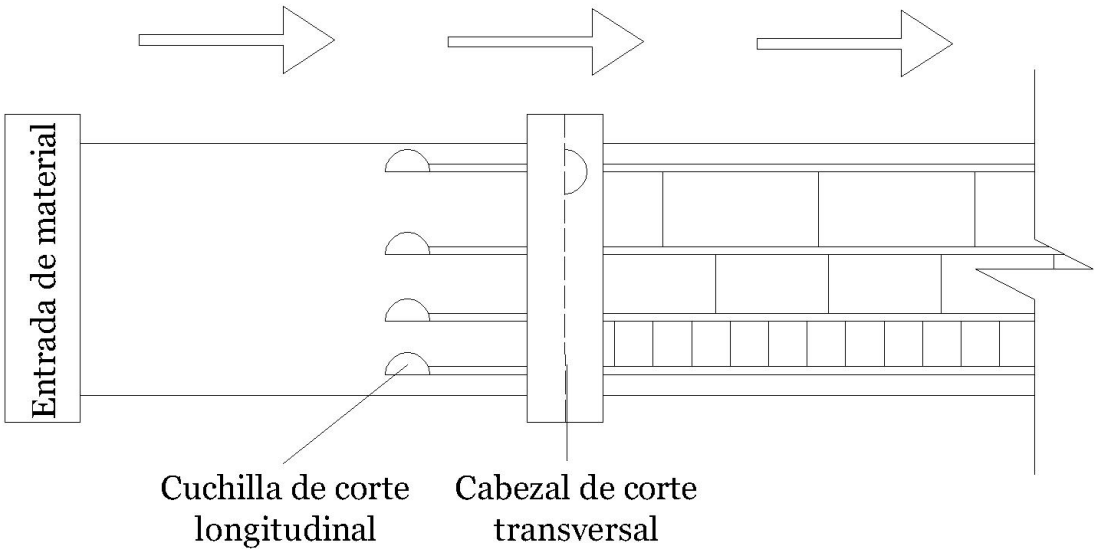
- Cuchillas de corte longitudinal, habitualmente circulares sobre el eje longitudinal que sustraen el borde (y en su caso, la merma restante de la anchura de los pedidos) y de N cuchillas más, también en el eje longitudinal de manera que la máquina puede hacer N+1 piezas a la vez sobre el eje transversal.

- Cuchillas de corte transversal, que en algunos casos son una guillotina que corta completamente el material de parte a parte, y en otros, es un cabezal móvil que corta/marca el material únicamente entre las zonas delimitadas.

No vamos a considerar en nuestro caso, el uso de una guillotina que corte transversalmente toda la materia prima a la vez, sino que una vez hecho el corte longitudinal de los pedidos, estos podrán separarse transversalmente de manera independiente y sin restricción alguna. En el caso del vidrio, se marca primero con un "diamante falso" y luego por medio de una vibración brusca se efectúa el corte transversal de la pieza.

Ejemplo de cuchilla marca transversalmente el material y posteriormente con movimiento seco, se separan las piezas de vidrio

Por tanto la restricción a tener en cuenta en este caso es que en un punto x sobre el eje longitudinal sólo pueden existir como máximo N+1 piezas sobre el eje transversal.



Obviamente al tratarse de materia prima, no puede darse el solapamiento entre distintas piezas.

2.5. Planteamiento del problema

Llegados a este punto, es necesario hacer una conversión del proceso en el mundo real a una representación que se pueda tratar matemáticamente y especialmente en el caso que nos ocupa computacionalmente.

Con la descripción del problema que ya se ha mostrado, es fácil deducir que nos encontramos ante un problema de combinatoria, ya que si queremos atender una serie

de pedidos formados por un conjunto de piezas determinados por sus dimensiones y la cantidad de cada una de ellas la solución al problema será la ordenación de las piezas sobre el material base para que tras el corte, obtengamos las piezas solicitadas. La combinatoria viene determinada por las distintas posibles colocaciones de las piezas sobre el material. Si además, añadimos el aprovechamiento del material base como un objetivo a maximizar (el material desperdiciado equivale a pérdidas económicas) entramos en el terreno de la optimización combinatoria. Estos problemas se caracterizan por tener múltiples soluciones válidas y un criterio para discriminar entre ellas.

En el siguiente punto se presenta una breve introducción a los problemas de optimización combinatoria.

2.5.1. Introducción a los problemas de optimización combinatoria

En la actualidad, podemos encontrar muchos problemas de optimización combinatoria, tanto en la industria, como el que nos ocupa en este TFG, como en la ciencia. Desde problemas clásicos de diseño de redes de telecomunicación hasta la re-ingeniería del software. Tanto problemas teóricos como prácticos.

Si clasificamos dichos problemas por su dificultad podemos encontrar los problemas lineales (en los que tanto la función objetivo como las restricciones son expresiones lineales) que se resuelven con el conocido método Simplex. Por otra parte, existen los problemas que son difícilmente representables por funciones lineales (la mayoría) y además, son muy difíciles de resolver. Estos últimos se denominan habitualmente NP-Hard.

> ***Definición de un problema NP-Hard:*** *Problemas de muy difícil resolución y que pese a contar con algún algoritmo de resolución, no se puede garantizar haber encontrado la mejor solución posible en un tiempo razonable computacionalmente hablando.*

Atendiendo a la forma de su resolución, se pueden clasificar los algoritmos en exactos, aquellos que garantizan la mejor solución, y los heurísticos, aquellos que además de la calidad, se prima la rapidez. Sin embargo, los algoritmos heurísticos no garantizan encontrar la mejor solución.

Resolución por métodos exactos:

- **Simplex** Se busca el máximo de una función lineal sobre un conjunto de variables que satisfaga un conjunto de inecuaciones lineales.

- **Ramificación y poda:** esta técnica se suele interpretar como un árbol de soluciones, donde cada rama nos lleva a una posible solución posterior a la

actual. El algoritmo se encarga de detectar en qué ramificación las soluciones dadas ya no están siendo óptimas, para «podar» esa rama del árbol y no continuar malgastando recursos en casos que se alejan de la solución óptima.

Resolución por métodos heurísticos "clásicos":

Habitualmente estos métodos se desarrollan a medida para un problema concreto. De esta manera se enfocan en encontrar mejores soluciones más rápidamente a cambio de no ser aprovechables en otros problemas.

- **Métodos de Descomposición:** El problema original se descompone en subproblemas más sencillos de resolver, teniendo en cuenta, aunque sea de manera general, que ambos pertenecen al mismo problema.

- **Métodos Inductivos:** La idea de estos métodos es generalizar de versiones pequeñas o más sencillas al caso completo. Propiedades o técnicas identificadas en estos casos más fáciles de analizar pueden ser aplicadas al problema completo.

- **Métodos de Reducción:** Consiste en identificar propiedades que se cumplen mayoritariamente por las buenas soluciones e introducirlas como restricciones del problema. El objeto es restringir el espacio de soluciones simplificando el problema. El riesgo obvio es dejar fuera las soluciones óptimas del problema original.

- **Métodos Constructivos:** Consisten en construir literalmente paso a paso una solución del problema. Usualmente son métodos deterministas y suelen estar basados en la mejor elección en cada iteración. Estos métodos han sido muy utilizados en problemas clásicos como el del viajante.

- **Métodos de Búsqueda Local:** A diferencia de los métodos anteriores, los procedimientos de búsqueda o mejora local comienzan con una solución del problema y la mejoran progresivamente. El procedimiento realiza en cada paso un movimiento de una solución a otra con mejor valor. El método finaliza cuando, para una solución, no existe ninguna solución accesible que la mejore.

Resolución por métodos metaheurísticos:

Los profesores Osman y Kelly (1995) introducen la siguiente definición:

Los procedimientos Metaheurísticos son una clase de métodos aproximados que están diseñados para resolver problemas difíciles de optimización combinatoria, en los que los heurísticos clásicos no son efectivos. Los Metaheurísticos proporcionan un marco general para crear nuevos algoritmos híbridos combinando diferentes conceptos derivados de la inteligencia artificial, la evolución biológica y los mecanismos estadísticos.

- **Templado Simulado:** Esta fue una de las primeras metaheurísticas (Kirkpatrick et al. 1983). Su nombre debe su origen al proceso de recocido de los materiales, si bien en este caso este enfoque permitirá especificar un método para decidir si una solución debe ser aceptada. El recocido simulado es una búsqueda solución a solución, es decir, en cada etapa se evalúa un único candidato. El procedimiento de aceptación y rechazo se realiza de la siguiente forma. Se valúa el candidato, si éste es mejor que el anterior, se acepta y lo sustituye. Si el candidato es peor en lugar de directamente rechazarlo, puede ser aceptado con una probabilidad que depende de los rendimientos de las soluciones y de un parámetro que se denomina temperatura y que va descendiendo de acuerdo con un plan de "enfriamiento" preestablecido a medida que se van realizando las iteraciones del algoritmo.

- **Búsqueda Tabú:** Al igual que en el caso del recocido simulado, se trata de una técnica solución a solución. En cada iteración del algoritmo hay una lista de soluciones que se han visitado en las iteraciones anteriores y por tanto son tabú. El algoritmo busca en el entorno de las soluciones no tabú y elige la mejor (Glover 1989 y 1990).

- **Métodos Evolutivos:** Se basan en conjuntos o poblaciones de soluciones. Así pues en cada iteración no se tiene una única solución como en las metaheurísticas anteriores sino que se tiene un conjunto de éstas. Una de las ventajas de los algoritmos evolutivos es que permiten explorar el espacio de soluciones de forma rápida e "inteligente".

    - **Algoritmos Genéticos:** Los Algoritmos Genéticos constituyen una de las metaheurísticas que evalúa a toda una población o conjunto de individuos en cada iteración. Se basan en la idea de la selección natural de forma que el conjunto de soluciones a lo largo de las iteraciones va evolucionando de acuerdo con unos operadores genéticos: recombinación, clonación y mutación (Goldberg 1989). Es el tipo de algoritmo que utilizaremos en este TFG para optimizar las soluciones generadas.

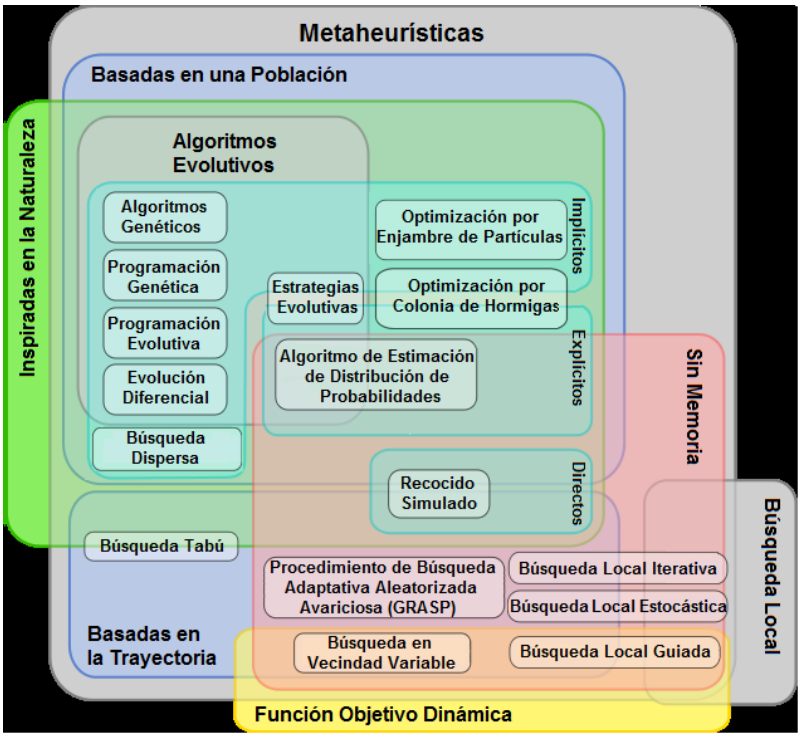
- **Búsqueda Dispersa:** Es otro tipo de metaheurística evolutiva. Al igual que en los algoritmos genéticos se parte de un conjunto de soluciones pero a diferencia de éstos en lugar de partir de poblaciones con un elevado número de individuos, en la búsqueda dispersa se suele trabajar con poblaciones de 10 individuos. Éstos se recombinan, pero en este caso la recombinación se realiza utilizando combinaciones lineales.

- **Métodos Combinados:** Conjunto de métodos que están a medio camino entre los métodos heurísticos y los metaheurísticos.

  - **Métodos GRASP:** Es otra metaheurística muy utilizada que consiste en iniciar la búsqueda desde diferentes puntos de arranque forma que ésta sea más global. En su versión más básica cada iteración consta de dos partes: una primera en la que se busca una solución buena aunque no sea la óptima local y una segunda fase en la que dentro del entorno de esa solución se busca su óptimo local. El procedimiento se repite hasta que se alcanza un criterio de fin (Feo, Resende 1989 y 1995).

  - **Multi-Arranque:** Básicamente consisten en aplicar de forma reiterada un procedimiento de búsqueda a partir de diferentes soluciones iniciales construidas. Se distinguen dos fases en cada iteración. Primero se construye una solución para después mejorarla mediante una heurística de búsqueda.

En la siguiente imagen podemos ver la clasificación de los distintos tipos de algoritmos agrupados por sus características en común



Fuente: https://es.wikipedia.org/wiki/Metaheur%C3%ADstica

2.5.2. Ubicación dentro de la clasificación de los problemas de optimización combinatoria

En la literatura actual, el problema que nos ocupa es denominado como problemas de optimización del corte o **cutting-stock problem**

> *The **cutting-stock problem** is an* NP-complete optimization *problem, essentially reducible to the* knapsack problem*. Specifically, it is an* integer linear programming *problem. It arises from many applications in industry. Imagine that you work in a paper mill and you have a number of rolls of paper of fixed width waiting to be cut, yet different customers want different numbers of rolls of various-sized widths. How are you going to cut the rolls so that you minimize the waste (amount of left-overs)?*
>
> http://en.wikipedia.org/wiki/Cutting_stock_problem

Como se puede ver en su definición, se puede reducir este tipo de problemas al problema de la mochila o **Knapsack problem**

> *The **knapsack problem** or **rucksack problem** is a problem in* combinatorial optimization*: Given a set of items, each with a mass and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size* knapsack *and must fill it with the most valuable items.*
>
> http://en.wikipedia.org/wiki/Knapsack_problem

A su vez, el problema de la mochila forma parte de los **packing problems**

> ***Packing problems** are a class of optimization problems in* mathematics *that involve attempting to pack objects together into containers. The goal is to either pack a single container as densely as possible or pack all objects using as few containers as possible.*
>
> *http://en.wikipedia.org/wiki/Packing_problems*

Debido a su naturaleza dual, ya que se puede buscar el objetivo de minimizar el espacio utilizado o maximizar el número de objetos en un espacio determinado, se dividen en **bin packing problems** y **strip packing problems**

> *In the **bin packing problem**, objects of different volumes must be packed into a finite number of bins or containers each of volume V in a way that minimizes the number of bins used. In* computational complexity theory*, it is a* combinatorial NP-hard *problem. The bin packing problem can also be seen as a special case of the* cutting stock problem*. When the number of bins is restricted to 1 and each item is characterised by both a volume and a value, the*

problem of maximising the value of items that can fit in the bin is known as the knapsack problem.

http://en.wikipedia.org/wiki/Bin_packing_problem

Given a set of rectangular pieces and a container of fixed width and variable length, the two-dimensional ***strip packing problem*** *(2D-SPP) consists of orthogonally placing all the pieces within the container, without overlapping, such that the overall length of the layout is minimised.*

A GENETIC ALGORITHM FOR THE TWO-DIMENSIONAL STRIP PACKING PROBLEM WITH RECTANGULAR PIECES - ANDREAS BORTFELDT

Debido a esta dualidad de enfoques, se puede considerar a los packing problems como problemas de cobertura o **covering problems**

In combinatorics *and* computer science*,* ***covering problems*** *are computational problems that ask whether a certain combinatorial structure 'covers' another, or how large the structure has to be to do that. Covering problems are* minimization problems *and usually* linear programs*, whose* dual problems *are called* packing problems.

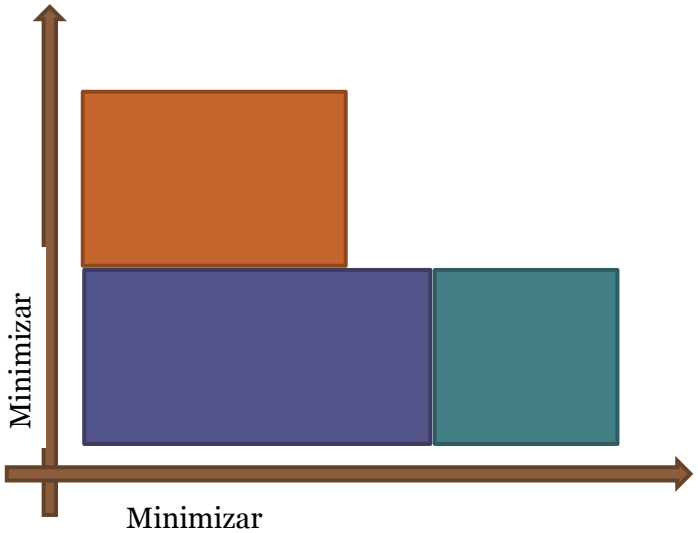
http://en.wikipedia.org/wiki/Covering_problems

Todos los problemas aquí relacionados pueden tratarse de manera n-dimensional, en el TFG trataremos sólo los 2-dimensionales

2.5.3. Definición concreta del problema a resolver

Una vez vistas las distintas categorías en las que se puede encuadrar el problema de la optimización de corte de material descrito en los anteriores puntos, podemos definir el problema como sigue:

Se trata de un strip packing problem en el que se trata de minimizar el área ocupada por las piezas que conforman los pedidos

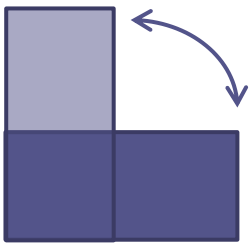


Al problema se le deben agregar las siguientes restricciones y consideraciones:

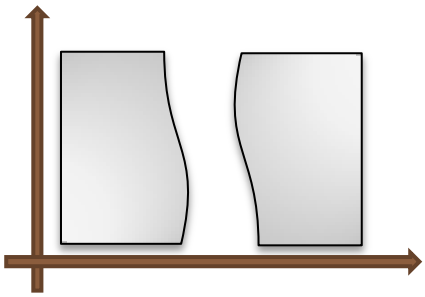
- Todas las piezas indicadas en los pedidos deben estar contenidos en la solución.

- Las piezas no pueden solaparse entre sí por ser piezas físicas que no pueden ocupar el mismo espacio a la vez.

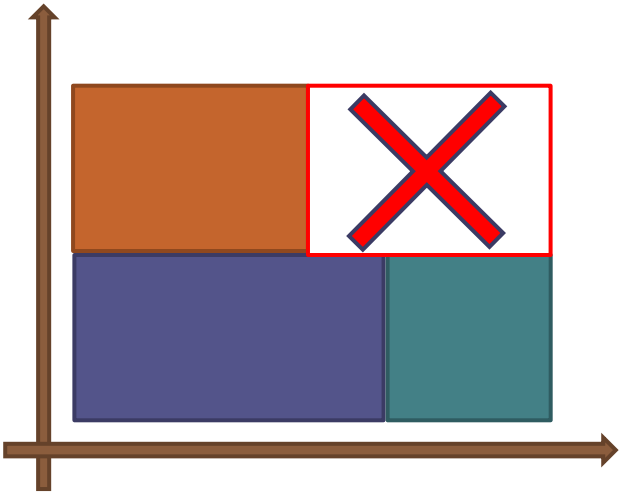
- Si el pedido lo permite, se pueden rotar las piezas para encontrar una mejor solución.



- Hay una dimensión del área fijada por el ancho de la bobina o el material primario. La otra dimensión es uno de los factores a minimizar (longitud de la solución).



- Se considera merma de material el área de material primario no cubierta por piezas de los pedidos y contenida en el área de la solución (ancho de material * longitud de la solución). El objetivo del problema es minimizar la merma.



- Se cuenta con un número limitado de cuchillas longitudinales. Este aspecto se traduce en que no pueden haber más piezas contiguas transversalmente que huecos entre las cuchillas. Ej.: Dos cuchillas para los bordes y dos cuchillas interiores → 3 piezas en paralelo.

- No existe restricción de la guillotina, todas las piezas se pueden separar longitudinalmente sin problemas.

- No se tiene en consideración lo que ocurre antes (de donde viene el material) y después del corte (donde van las piezas)

- Se considerará que una solución es mejor que otra en base al porcentaje de aprovechamiento del material primario. Esto es la solución que menos merma produzca independientemente de su longitud.

3. Diseño de la solución a implementar

Para la elaboración de este TFG se ha tomado como referencia un algoritmo existente basado en el artículo de Andreas Bortfeldt denominado "*A genetic algorithm for the two-dimensional strip packing problem with rectangular pieces*" de 2004 y se han realizado las adaptaciones necesarias para solucionar el problema concreto que nos ocupa.

Se propone por tanto, implementar un algoritmo genético que mediante la generación de una población inicial, vaya mejorando la solución paulatinamente en cada iteración del algoritmo. La población inicial se generará de manera aleatoria, sin embargo se proporcionarán dos métodos de generación para poder comparar la calidad de la solución en función del punto de partida

Para facilitar la comprensión de las soluciones se utilizará una codificación en forma de coordenadas de la ubicación del vértice inferior izquierdo de cada pieza y se controlará el solapamiento por medio de la comparación de los vértices origen más las medidas ancho/alto

3.1. Representación del modelo

Según el artículo de referencia, se modelan las soluciones en base a una representación jerárquica basada en un conjunto de "capas" donde cada capa contiene un conjunto de piezas. Además hay que tener en cuenta que la profundidad de cada capa viene determinada por su primera pieza, de manera que no puede haber ninguna pieza más profunda que la primera, ni la acumulación de varias piezas puede superar este valor.

A continuación se expone el esquema que aparece en el artículo.

A. Bortfeldt / European Journal of Operational Research 172 (2006) 814–837

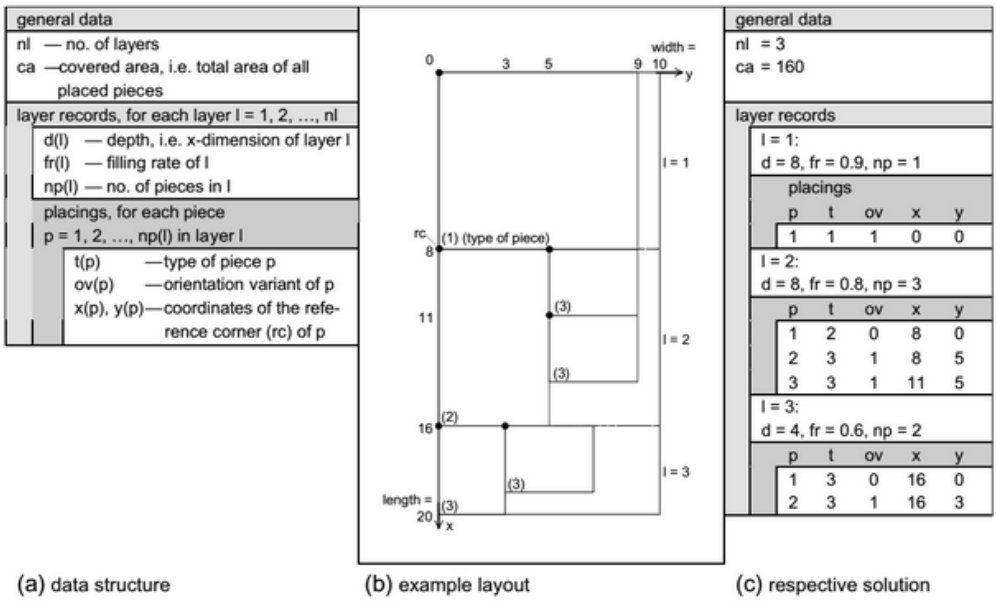


Fig. 2. Representation of layouts.

Como se puede observar, las soluciones contienen una jerarquía representada por:

- Datos generales
    - nl: Nº total de capas de la solución
    - ca: Área cubierta o utilizada. Es el espacio que ocupan las piezas de la solución.
    - Detalle de las capas, para cada capa l = 1, 2, … nl
        - d(l): Profundidad de la capa l
        - fr(l): Aprovechamiento de la capa l en tanto por uno
        - np(l): Nº de piezas que componen la capa l
        - Detalle de las piezas, para cada pieza p = 1, 2, .., np(l) de la capa l
            - t(p): Tipo de pieza
            - ov(p): Orientación
            - x(p), y(p): Coordenadas de su vértice de referencia

En nuestro caso, vamos a adaptar la información almacenada con dos objetivos principales: minimizar los cálculos del algoritmo (no calcular repetidamente valores constantes) y su fácil interpretación/representación una vez calculada. Se trata de una notación explícita en la que viendo directamente la estructura de los datos se puede representar fácilmente la solución o realizar comparaciones con otras soluciones para determinar su relación de optimalidad.

Cada capa estará formada por una única pieza en la base que determinará su profundidad. Dicha capa se completará con un número máximo de n piezas paralelas transversalmente correspondientes al número de cuchillas adicionales de la cortadora. Ej.: Para una cortadora de 3 cuchillas móviles más las de los bordes, se tendrá un máximo de 4 piezas por capa. Al igual que el modelo original, también se permitirán varias piezas longitudinalmente siempre que sean del mismo tipo y no se supere la profundidad de la pieza que da origen a la capa.

Si tenemos en cuenta que cada capa puede llevar una configuración diferente de piezas, se interpretar el número de capas como el número máximo de movimientos que tendrán que hacer las cuchillas longitudinales para obtener todas las piezas. Nombramos lo de “máximo” porque puede haber capas que repitan su configuración y tan solo se necesitan los cortes transversales.

- Datos generales de la estructura que contiene la solución:
    - nc: Número total de capas de la solución.
    - am: Área de material utilizado (largo por ancho del material)
    - au: Área útil de material utilizado (au(c))
    - ad: Área desperdiciada de material utilizado (am - au)
    - pa: Porcentaje de aprovechamiento (au/am*100)
    - Detalle de las capas, para cada capa c = 1, 2, …, nc
        - co(c): Posición/coordenadas de la capa (minY, maxY)
        - pr(c): Profundidad (largo de la pieza que forma la base)
        - ac(c): Área de la capa (largo x anchura del material)
        - au(c): Área útil de material utilizado en la capa (au(p))
        - ad(c): Área desperdiciada de material utilizado en la capa (ac(c) - au(c))
        - pa(c): Porcentaje de aprovechamiento de la capa (au(c) / ac(c))
        - np(c): Número de piezas que componen la capa c

- Detalle de las piezas, para cada pieza p = 1, 2, ..., np(c) de la capa c
  - t(p): Tipo de pieza
  - r(p): Rotación de la pieza (Si/No)
  - minx(p), miny(p): Coordenadas de su vértice de referencia
  - maxx(p), maxy(p): Coordenadas de su vértice opuesto

Además del modelo de la solución, necesitaremos representar algunos parámetros mas no incluidos en la solución, pero que condicionarán la generación de estas. Estos parámetros se consideran parámetros del problema y son:

- ab: Ancho de la bobina de material
- cu: Número de cuchillas de la máquina
- Detalle de los pedidos, para cada pedido p
  - id(p): Identificador del pedido
  - an(p): ancho de la pieza del pedido
  - la(p): largo de la pieza del pedido
  - ca(p): cantidad de piezas del pedido
  - ro(p): Indicador de si se permite rotar las piezas de este pedido

3.2. Algoritmo a implementar

Al igual que con el modelo de la solución, en este TFG partimos de la definición que propone a Bortfeldt en su artículo, que no es ni más ni menos que una búsqueda genética estándar basada en tres pasos:

1. Crear la población inicial de soluciones
2. Crear la siguiente generación aplicando los operadores
3. Repetir el paso 2 cuántas veces se requiera.

3.2.1. Definición del algoritmo original

A continuación se muestra un recorte del artículo referente al algoritmo planteado en él. Recordemos de nuevo que en el planteamiento original este algoritmo está pensado para solucionar un problema de cobertura basado en una superficie finita y la intención de aprovecharla al máximo posible.

```
procedure cl_genetic_search(in: container length lC, container width wC, piece set P, out: best solution sbest)
set generation counter g := 0;
generate npop solutions for the start generation g;
for g := 1 to ngen do
        reproduce the best nrep solutions from generation g – 1 for generation g;
        while generation g contains fewer than npop solutions do
                if randomly chosen operator is crossover then
                        select parent solutions ps1, ps2 in generation g – 1;
                        generate offspring os by crossover for generation g;
                else {randomly chosen operator is mutation}
                        select parent solution ps in generation g – 1;
                        generate offspring os by standard mutation for generation g;
                endif;
        endwhile;
        for i := 1 to nmerge do
                select parent solution ps in generation g – 1;
                generate offspring os by merger mutation;
                if os is better than the worst solution ws in generation g then
                        replace ws by os;
                endif;
        endfor;
endfor;
end.
```

Fig. 3. Overall procedure of the GA for the CLP.

El algoritmo se define como un procedimiento que recibe las medidas del contenedor y el conjunto de piezas a colocar, y devuelve la mejor solución que haya encontrado. Para ello, se generan una población inicial (sin especificar como) con tantas soluciones como el tamaño de la población especificado (npop). A continuación se van creando las diferentes generaciones hasta llegar al límite marcado (ngen). En cada generación, se reproducen las nrep mejores soluciones de la generación anterior y a continuación se completa la población aplicando los operadores de cruce o mutación aleatoriamente. Una vez completada la población de la nueva generación, se aplica un número nmerge de veces el operador mutación combinada cuyo resultado se compara con la peor solución de la población y si lo mejora, la sustituye.

- **Operador cruce (crossover):** Se seleccionan dos soluciones "padre" de la generación anterior y se combinan por orden de aprovechamiento y se seleccionan las mejores hasta que todas las piezas están colocadas o no queden más capas por añadir. Si una capa, incluye un tipo de pieza cuya cantidad ya ha sido cubierta, se descarta y se pasa a la siguiente capa.

- **Operador mutación (mutation):** Se seleccionan dos soluciones "padre" de la generación anterior y se copian hasta un máximo del 50% de capas aleatorias. Una vez cubierto este porcentaje se completa la solución como si de un cruce se tratara, seleccionando las mejores capas restantes hasta que se complete la solución.

- **Operador mutación combinada (merger mutation):** Se selecciona una única solución "padre" y se copian todas las capas menos dos aleatoriamente.

3.2.2. Adaptación del algoritmo original a nuestro problema

A diferencia del planteamiento del anterior algoritmo, en este TFG se pretende ofrecer una solución a un problema ligeramente distinto. Que es: dadas un conjunto de piezas, intentar que ocupen el menor espacio posible fijando únicamente una de sus dimensiones. Por lo tanto debemos adaptar el algoritmo de referencia a las características de nuestro problema concreto. Principalmente en cuanto a la generación de las soluciones ya que no pueden quedar incompletas con piezas por incorporar ya que nuestro espacio es "infinito", por tanto se tienen que redefinir los operadores para que tengan en cuenta esta característica.

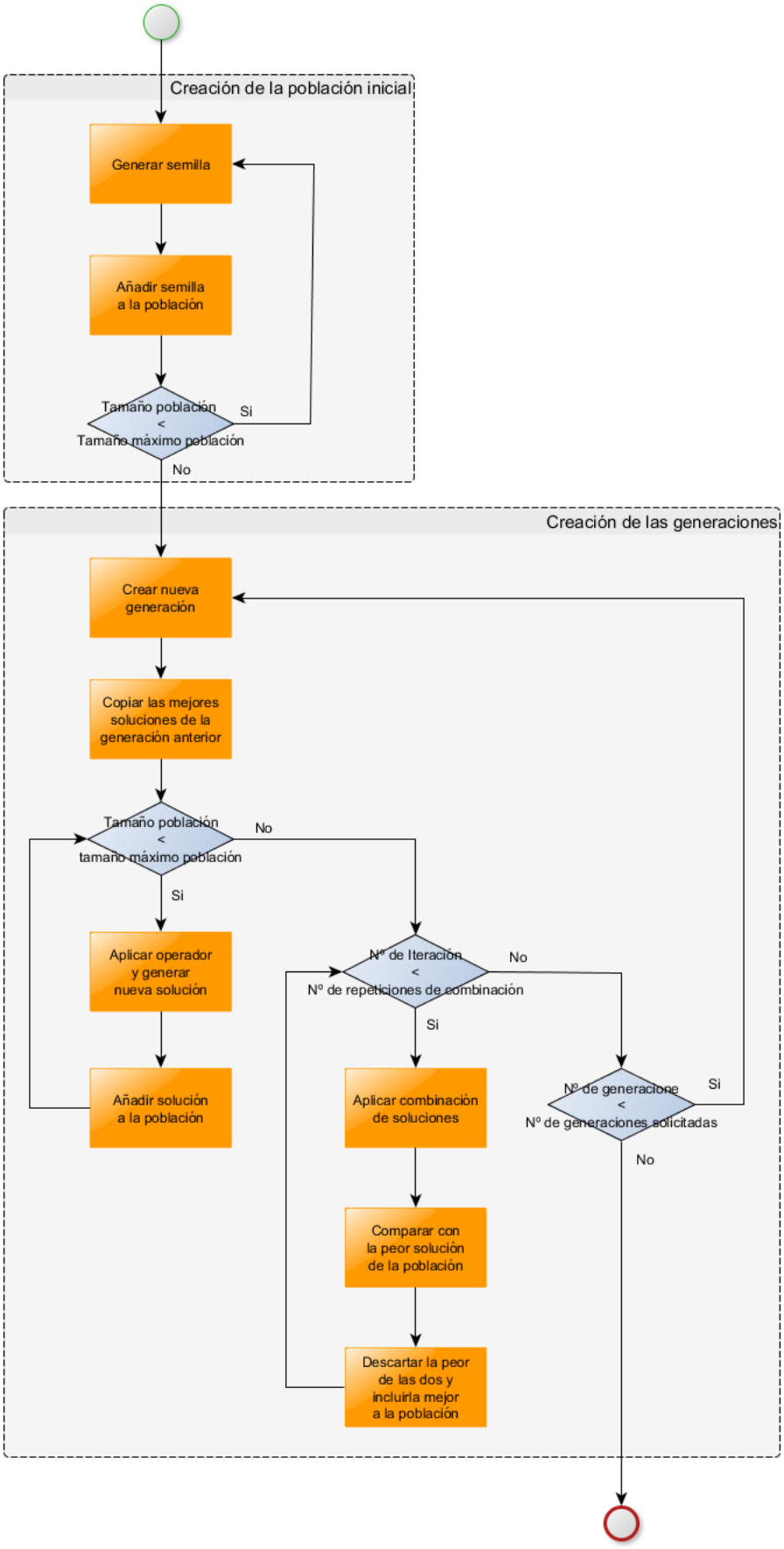
En los siguientes puntos presentamos la adaptación definitiva del algoritmo propuesto.

Parámetros del problema. (sin modificaciones respecto al modelo original):

- ab: Ancho de la bobina de material
- cu: Número de cuchillas de la máquina
- Detalle de los pedidos, para cada pedido p
  - id(p): Identificador del pedido
  - an(p): ancho de la pieza del pedido
  - la(p): largo de la pieza del pedido
  - ca(p): cantidad de piezas del pedido
  - ro(p): Indicador de si se permite rotar las piezas de este pedido

Procedimiento de búsqueda de solución:

- Parámetros de entrada:
  pp: Parámetros del problema
  pa: Parámetros del algoritmo
- Parámetros de salida:
  ls: Listado de soluciones
- Algoritmo
  Creación de la población inicial:
  Mientras soluciones < tamaño máximo de la población
  Generar semilla
  Para cada generación
  Reproducir las *n* mejores soluciones de la generación anterior
  Mientras soluciones < tamaño máximo de la población
  Seleccionar aleatoriamente (ponderado) el operador
  Repetir el número de repeticiones de combinación
  Se ejecuta la combinación de soluciones
  Si el resultado es mejor que la peor solución de esta generación, se reemplaza



<

Diagrama de flujo del algoritmo

Por tratarse de un algoritmo genético, vamos a heredar los operadores básicos de cruce y mutación, además, como se trata de una adaptacion academica para comprobar la posibilidad de utilizar el planteamiento del algoritmo original en este tipo de problemas, creemos conveniente añadir algunos operadores más, lo cual permitirá comparar en la fase de pruebas su viabilidad o aporte real en el proceso de búsqueda de la solución óptima.

En nuestro caso se podrían utilizar los siguientes operadores:

- **Semilla:** Genera una nueva distribución desde cero. Este operador se utiliza principalmente para generar la población inicial, pero utilizado en la fase de evolución de las generaciones sirve para paliar la endogamia que se va generando y que es propia de los algoritmos genéticos. En otras palabras, si el algoritmo se estanca en un óptimo local, aporta vías de exploración fuera de él.

- **Cruce (crossover):** Con una definición similar al operador original, se seleccionan dos soluciones "padre" de la generación anterior y se combinan por orden de aprovechamiento, a continuación se seleccionan las mejores capas hasta que todas las piezas están colocadas o no queden más capas por añadir. Si una capa, incluye un tipo de pieza cuya cantidad ya ha sido cubierta, se descarta y se pasa a la siguiente capa. Se crea una nueva solución con las lapas seleccionadas. Se añade en este punto, la obligación de contar con una solución con todas las piezas añadidas. Para ello, se completa la solución con el mismo algoritmo utilizado por el operador semilla.

- **Mutación (mutation):** En este operador, se selecciona un porcentaje (definido por parámetro) de las mejores capas de dos soluciones fuente de manera similar al operador cruce, pero limitando el número de capas. Se selecciona un porcentaje aleatorio del resto de capas de las distribuciones fuente y se descartan las capas no seleccionadas hasta el momento. Para completar la solución, se añaden las piezas no incluidas aún con el mismo criterio que el operador semilla.

- **Combinación de capas:** Para añadir un componente de optimización directa en la propia evolución de las soluciones, se utilizará este operador que actúa sobre una solución origen y que intenta combinar varias capas en una cuando todas las piezas de ambas tengan cabida en una única capa respetando las restricciones del problema (cuchillas, las orientaciones de las piezas, ...)

- **Combinación de soluciones:** Este operador es el sustituto de la mutación combinada, por tanto solo se utilizará en la última fase de creación de una nueva generación. Se seleccionan aleatoriamente las capas de varias soluciones de la generación anterior hasta que todos los pedidos están cubiertos o no quedan más capas. Al igual que en el resto

de operadores, si es necesario se completa la solución con el algoritmo del operador semilla.

Procedimiento del operador semilla:

- Parámetros de entrada:
  pp: Parámetros del problema
  pa: Parámetros del algoritmo
  s: Estructura de la solución a completar. Puede estar totalmente vacía o generada parcialmente.
- Parámetros de salida:
  sg: Solución generada
- Algoritmo A
  Mientras queden piezas por incorporar a la solución
    Se selecciona una pieza al azar entre las existentes en los pedidos
    Si hay capas en las que quepa
      Se selecciona una de las capas al azar y se incorpora la pieza
    Si no hay capas en las que quepa
      Se crea una nueva capa con la pieza
- Algoritmo B
  Mientras queden piezas por incorporar a la solución
    Se selecciona una pieza al azar entre las existentes en los pedidos
    Se decide al azar si se creara una capa nueva o se incorporara a una existente
      Si se incorpora a una capa existente
        Si hay capas en las que quepa
          Se selecciona una de las capas al azar y se incorpora la pieza
        Si no hay capas en las que quepa
          Se crea una capa nueva con la pieza
      Si se crea una capa nueva
        Se crea una capa nueva con la pieza

Procedimiento del operador cruce:

- Parámetros de entrada:
  pp: Parámetros del problema
  pa: Parámetros del algoritmo
  so: Soluciones de la generación anterior.
- Parámetros de salida:
  sg: Solución generada
- Algoritmo
  Se unen las capas de ambas soluciones en una sola estructura y se ordenan por factor de aprovechamiento
  Mientras queden capas en la lista fuente
    Se selecciona la primera capa de la lista
      Si todas sus piezas están disponibles en el listado de piezas de los pedidos
        Se agrega la capa a la solución
        Se elimina la capa de la lista fuente

Se eliminan las piezas que la conforman de los pedidos
Si alguna de las piezas que contiene la capa no está disponible en el listado de piezas de los pedidos
Se elimina la capa de la lista
Si quedan piezas en los pedidos
Llamada al operador semilla

***Procedimiento del operador** mutación:*

- Parámetros de entrada:
  pp: Parámetros del problema
  pa: Parámetros del algoritmo
  so: Soluciones de la generación anterior.
- Parámetros de salida:
  sg: Solución generada
- Algoritmo
  Se unen las capas de ambas soluciones en una sola estructura y se ordenan por factor de aprovechamiento
  Mientras % de capas seleccionadas < % indicado por parámetro *%aleatorio*
  Seleccionar una nueva capa aleatoriamente
  Si todas sus piezas están disponibles en el listado de piezas de los pedidos
  Se agrega la capa a la solución
  Se elimina la capa de la lista fuente
  Se eliminan las piezas que la conforman de los pedidos
  Si alguna de las piezas que contiene la capa no está disponible en el listado de piezas de los pedidos
  Se elimina la capa de la lista
  Mientras % de capas seleccionadas < % indicado en parámetro *%mejor*
  Seleccionar la mejor capa disponible
  Si todas sus piezas están disponibles en el listado de piezas de los pedidos
  Se agrega la capa a la solución
  Se elimina la capa de la lista fuente
  Se eliminan las piezas que la conforman de los pedidos
  Si alguna de las piezas que contiene la capa no está disponible en el listado de piezas de los pedidos
  Se elimina la capa de la lista
  Mientras queden piezas en los pedidos
  Se utiliza el algoritmo semilla para completar la solución

***Procedimiento del operador** combinación de capas:*

- Parámetros de entrada:
  pp: Parámetros del problema
  pa: Parámetros del algoritmo
  so: Solución de la generación anterior.
- Parámetros de salida:
  sg: Solución generada

- Algoritmo
  Se seleccionan aleatoriamente dos capas que tengan las piezas del mismo tipo (Cada hueco entre cuchillas debe estar relleno con piezas del mismo ancho) o bien quepan todas las piezas en una sola capa (puede que una capa tenga más huecos rellenos que la otra)
  Se juntan en una sola capa y se recalculan las variables (puede que la pieza base pase a ser otra)

***Procedimiento del operador** combinación de soluciones:*

- Parámetros de entrada:
  pp: Parámetros del problema
  pa: Parámetros del algoritmo
  so: Solución de la generación anterior.
- Parámetros de salida:
  sg: Solución generada
- Algoritmo
  Se unen las capas de ambas soluciones en una sola estructura y se ordenan por factor de aprovechamiento
  Mientras queden capas en la lista fuente
  Se selecciona una capa al azar de la lista
  Si las piezas están disponibles en el listado de piezas de los pedidos
  Se agrega la capa a la solución
  Se elimina la capa de la lista
  Se eliminan las piezas que la conforman de los pedidos
  Si las piezas no están disponibles en el listado de piezas de los pedidos
  Se elimina la capa de la lista
  Mientras queden piezas en los pedidos
  Se utiliza el algoritmo semilla para completar la solución

***Procedimiento del operador** genérico:*

Puede deducirse de los anteriores pseudocódigos, que todos los operadores se pueden plantear de una única forma genérica ya que tienen muchos parámetros en común y en algunos casos se llega a reproducir parte del procedimiento o llamar a otro procedimiento directamente (Semilla). De esta manera y planteando como un único operador con los parámetros y el procedimiento unificado se puede conseguir que dando más peso a ciertos factores según el tipo de operador original, se obtengan los mismos resultados con un único algoritmo. Además, plantearlo como un único operador parametrizado aporta mucha flexibilidad y permite explorar distintas configuraciones fácilmente, lo cual permite afinar mucho mejor las capacidades del algoritmo.

A continuación se presenta un cuadro resumen que permite comparar los aspectos que tienen en común los operadores y plantear a partir de ahí el operador genérico.

Operador - Parámetros	% de capas aleatorias de soluciones fuente	% de mejores capas de soluciones fuente	% de nuevas capas	% de recombinación de capas propias
Semilla	-	-	100%	-
Cruce	-	100%	Piezas restantes	-
Mutación de una solución	n%	m%	(100 - n - m)% (Piezas restantes)	-
combinación de capas	-	-	-	p %
combinación de soluciones	100%	-	Piezas restantes	-

Como se puede observar en el cuadro, prácticamente todos los operadores tienen varios componentes y los componentes se repiten en varios operadores. Por ello se plantea un procedimiento que primero seleccione de manera aleatoria un porcentaje de capas, continúe con la selección de un porcentaje de las mejores capas que aun estén disponibles y finalmente complete la solución con el algoritmo de generación de capas nuevas. Después del proceso de construcción de la nueva solución se decidirá de manera aleatoria ponderada aplicará o no (según el parámetro) un intento de combinación de las propias capas de la solución construida

- Parámetros de entrada:
  - pp: Parámetros del problema
  - pa: Parámetros del algoritmo
  - so: Solución/es de la generación anterior.
- Parámetros de salida:
  - sg: Solución generada
- Algoritmo
  - Se unen las capas de ambas soluciones en una sola estructura y se ordenan por factor de aprovechamiento
  - Mientras % de capas seleccionadas < % indicado por parámetro *%aleatorio*
    - Seleccionar una nueva capa aleatoriamente
    - Si todas sus piezas están disponibles en el listado de piezas de los pedidos
      - Se agrega la capa a la solución
      - Se elimina la capa de la lista fuente
      - Se eliminan las piezas que la conforman de los pedidos
    - Si alguna de las piezas que contiene la capa no está disponible en el listado de piezas de los pedidos
      - Se elimina la capa de la lista
  - Mientras % de capas seleccionadas < % indicado en parámetro *%mejor*
    - Seleccionar la mejor capa disponible

Si todas sus piezas están disponibles en el listado de piezas de los pedidos
Se agrega la capa a la solución
Se elimina la capa de la lista fuente
Se eliminan las piezas que la conforman de los pedidos
Si alguna de las piezas que contiene la capa no está disponible en el listado de piezas de los pedidos
Se elimina la capa de la lista
Mientras queden piezas en los pedidos
Se utiliza el algoritmo semilla para completar la solución
Se seleccionan aleatoriamente dos capas que tengan las piezas del mismo tipo o bien quepan todas las piezas en una sola capa.
Se juntan en una sola capa y se recalculan las variables.

Parametrización del algoritmo:

Al comienzo de este punto se definen los parámetros específicos del problema que se deben tener en cuenta para calcular las soluciones. Como hemos visto según se iban desarrollando los distintos procedimientos, también se van a necesitar una serie de parámetros que no son relativos al problema sino al propio algoritmo de resolución. pasaremos por tanto a desarrollarlos en este apartado.

Parámetros específicos de la creación de generaciones

- pm: Población máxima de cada generación.
- ng: Nº de generaciones que evolucionara el procedimiento de búsqueda.
- rm: Nº de mejores soluciones que se repetirán de una generación a la siguiente
- ic: Nº de intentos de combinación de soluciones

Parámetros específicos del algoritmo de semilla

- mu: Método utilizado
- nc: Probabilidad de nueva capa si se utiliza el método B

Parámetros específicos de los operadores (para cada operador o)

- so(o): Probabilidad de que sea seleccionado en la fase de selección de operador
- ca: Porcentaje de capas aleatorias que se copiaran a la nueva solución desde las soluciones fuente
- cm: Porcentaje de mejores capas ques ee copiaran a la nueva solución desde las soluciones fuente.
- cc: Porcentaje de posibilidad de intentar la combinación de capas

3.2.3. Otras consideraciones a la implementación del algoritmo

Al tratarse de un algoritmo de optimización es importante contar con una solución competente de manera muy rápida y obtener progresivamente mejores soluciones en función del tiempo disponible. En base a esto, lo habitual es establecer un número alto de generaciones para que la calidad de las soluciones tenga una evolución reconocible. Sin embargo un número elevado de generaciones junto con la complejidad propia del algoritmo puede ocupar bastante tiempo de proceso ocupando toda la capacidad del procesador y llegue a parecer que el sistema está bloqueado. Por ello, los algoritmos antes mencionados se implementarán en forma *anytime* de manera que a partir de la primera solución, se puede detener el algoritmo en cualquier momento y siempre devolverá la mejor solución encontrada hasta ese momento. De esta manera conseguimos un algoritmo que se puede parar y reanudar en cualquier momento, llegando a tener la posibilidad de cambiar los parámetros del algoritmo en la siguiente reanudación.

4. Desarrollo de la solución

Hasta el momento se ha definido el problema, explicado su evolución histórica, encuadrado dentro de los problemas matemáticos similares y definido a nivel teórico lo que el algoritmo debería hacer. En este punto es donde se elaborará el algoritmo en un entorno de programación real y se convertirá en un proceso ejecutable en un ordenador.

4.1. Entorno y lenguaje utilizado

Para poder probar y experimentar con cualquier algoritmo es necesario implementarlo en algún lenguaje de programación. EN la actualidad existen cientos de lenguajes de programación, algunos con propósitos específicos y otros con una orientación más genérica, pero existen algunos que siempre se mantienen en cabeza en cuanto a popularidad y uso. La elección del lenguaje ha tenido en consideración

Jun 2016	Jun 2015	Change	Programming Language	Ratings	Change
1	1		Java	20.794%	+2.97%
2	2		C	12.376%	-4.41%
3	3		C++	6.199%	-1.56%
4	6	^	Python	3.900%	-0.10%
5	4	v	C#	3.786%	-1.27%
6	8	^	PHP	3.227%	+0.36%
7	9	^	JavaScript	2.583%	+0.29%
8	12	^^	Perl	2.395%	+0.64%
9	7	v	Visual Basic .NET	2.353%	-0.82%
10	16	^^	Ruby	2.336%	+0.98%
11	11		Visual Basic	2.254%	+0.41%
12	23	^^	Assembly language	2.119%	+1.36%
13	10	v	Delphi/Object Pascal	1.939%	+0.07%
14	14		Swift	1.831%	+0.39%
15	5	vv	Objective-C	1.704%	-2.64%

Los 15 lenguajes de programación más populares en 2016 según tiobe.com

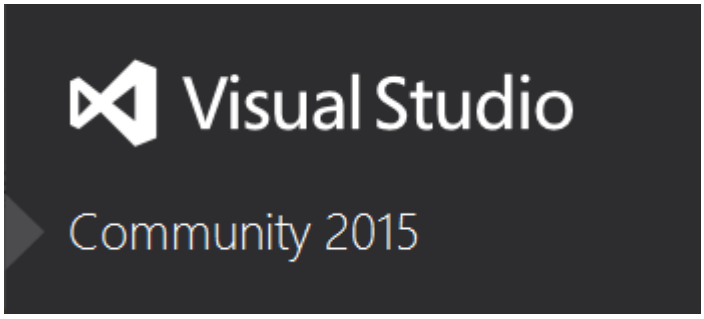
Siendo un TFG realizado por un estudiante que no se dedica profesionalmente a la programación, sino a la gestión de proyectos, se han tenido en cuenta para la elección del lenguaje aspectos como: facilidad de implementación, orientación a objetos, abundancia de documentación y principalmente familiaridad con él a través de experiencias previas. También ha influido en la decisión, contar con un entorno de desarrollo integrado en el que minimizar las configuraciones necesarias para empezar a programar. Por lo tanto **el lenguaje elegido ha sido C#** ya que es un lenguaje que cuenta con todas estas características y cuenta con un IDE de desarrollo tan intuitivo como facilidad para empezar a desarrollar con él proyectos como este.

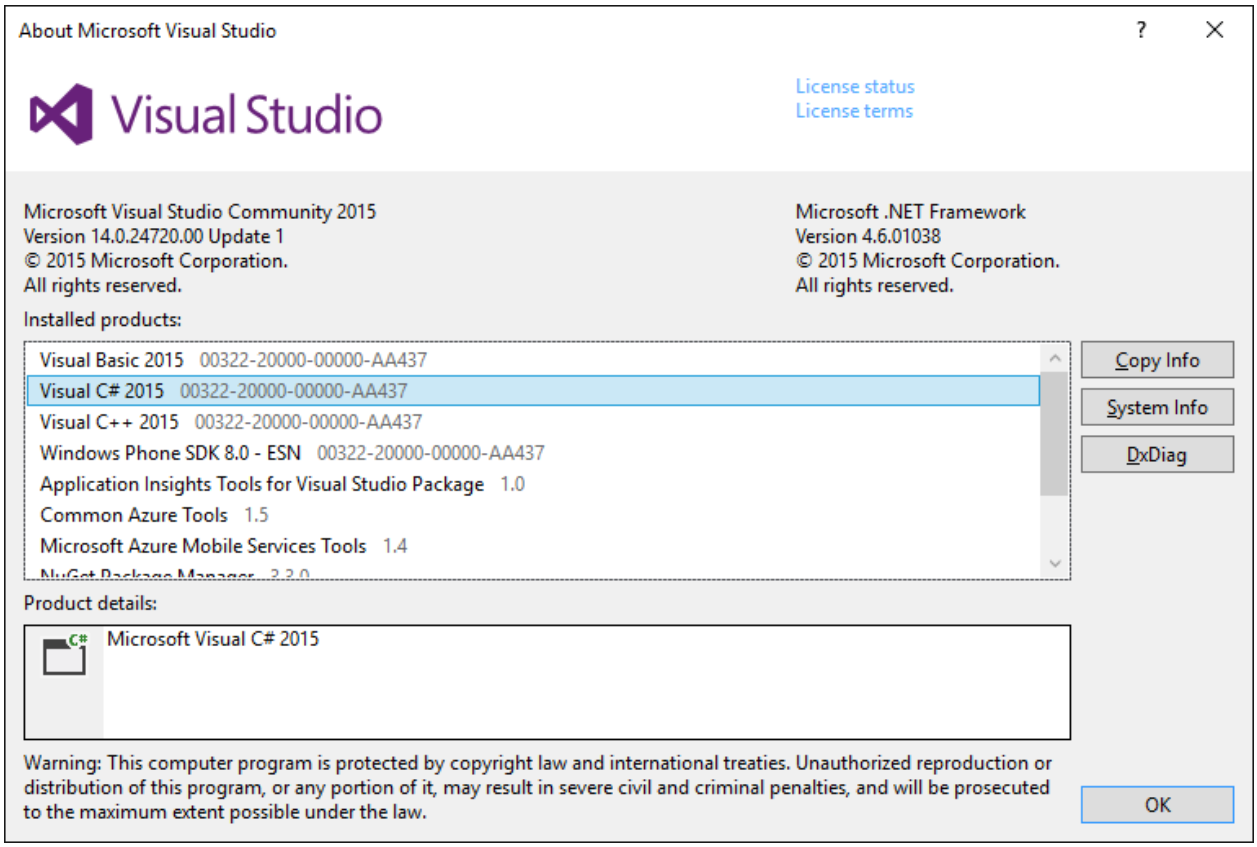
La wikipedia proporciona un buen resumen del lenguaje::

> ***C#***[1] *(pronunciado si sharp en inglés) es un lenguaje de programación orientado a objetos desarrollado y estandarizado por Microsoft como parte de su plataforma .NET, que después fue aprobado como un estándar por la ECMA (ECMA-334) e ISO (ISO/IEC 23270). C# es uno de los lenguajes de programación diseñados para la infraestructura de lenguaje común.*
>
> *Su sintaxis básica deriva de C/C++ y utiliza el modelo de objetos de la plataforma .NET, similar al de Java, aunque incluye mejoras derivadas de otros lenguajes.*
>
> *El nombre C Sharp fue inspirado por la notación musical, donde '#' (sostenido, en inglés sharp) indica que la nota (C es la nota do en inglés) es un semitono más alta, sugiriendo que C# es superior a C/C++. Además, el signo '#' se compone de cuatro signos '+' pegados.*
>
> *Aunque C# forma parte de la plataforma .NET, ésta es una API, mientras que C# es un lenguaje de programación independiente diseñado para generar programas sobre dicha plataforma. Ya existe un compilador implementado que provee el marco Mono - DotGNU, el cual genera programas para distintas plataformas como Windows, Unix, Android, iOS, Windows Phone, Mac OS y GNU/Linux.*
>
> https://es.wikipedia.org/wiki/C_Sharp

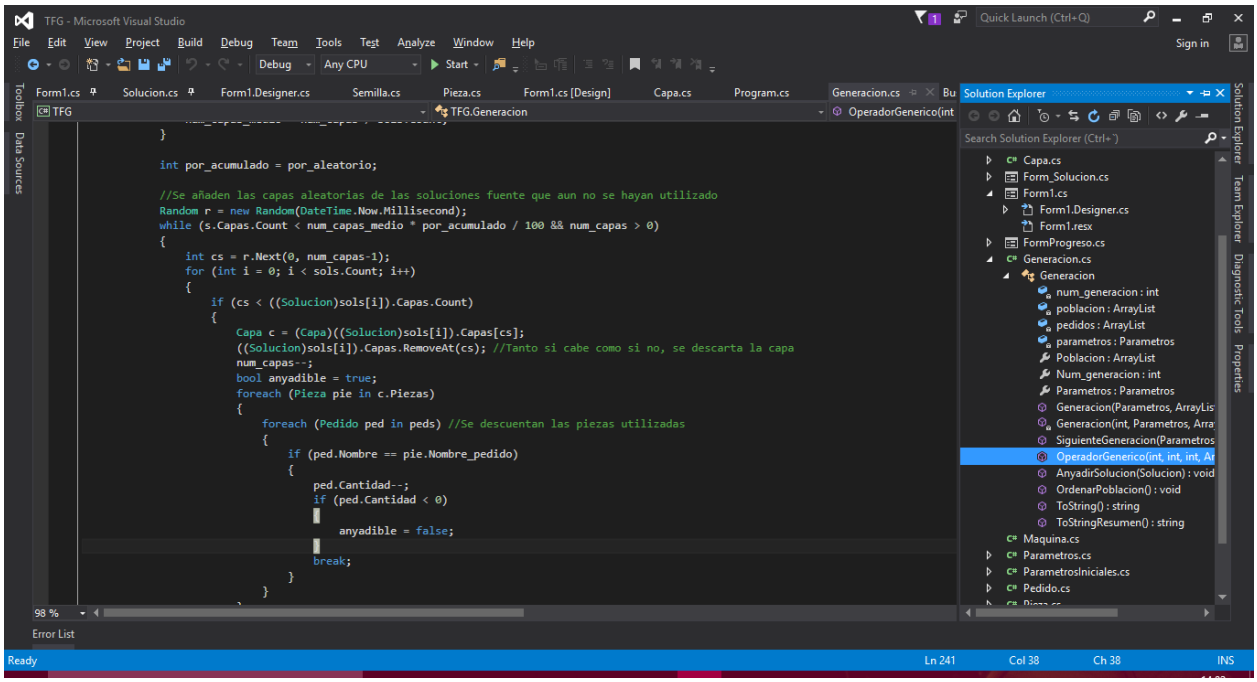
Se han descartado los lenguajes que ocupan posiciones más elevadas en la lista por desconocimiento (Python), complejidad de configuración y/o distribución (Java) y por complejidad del desarrollo en un entorno visual (c y c++).

En cuanto al entorno de desarrollo y tratándose de una aplicación en Windows, la elección es obvia, el IDE propio de los desarrolladores del lenguaje: **Visual Studio**, concretamente su última versión **Comunity 2015** que *"es gratuito para desarrolladores individuales, proyectos de código abierto, investigación académica, educación y pequeños equipos profesionales."* según su página de descarga. En la siguiente captura se puede ver la versión concreta utilizada.

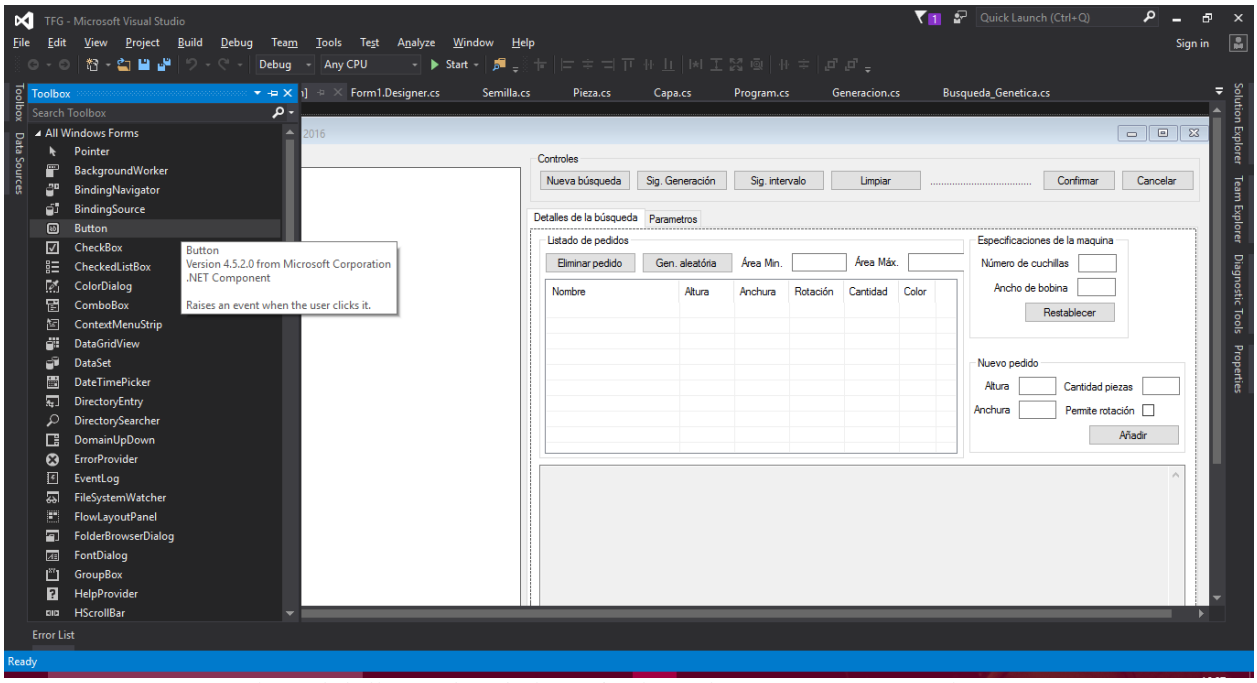




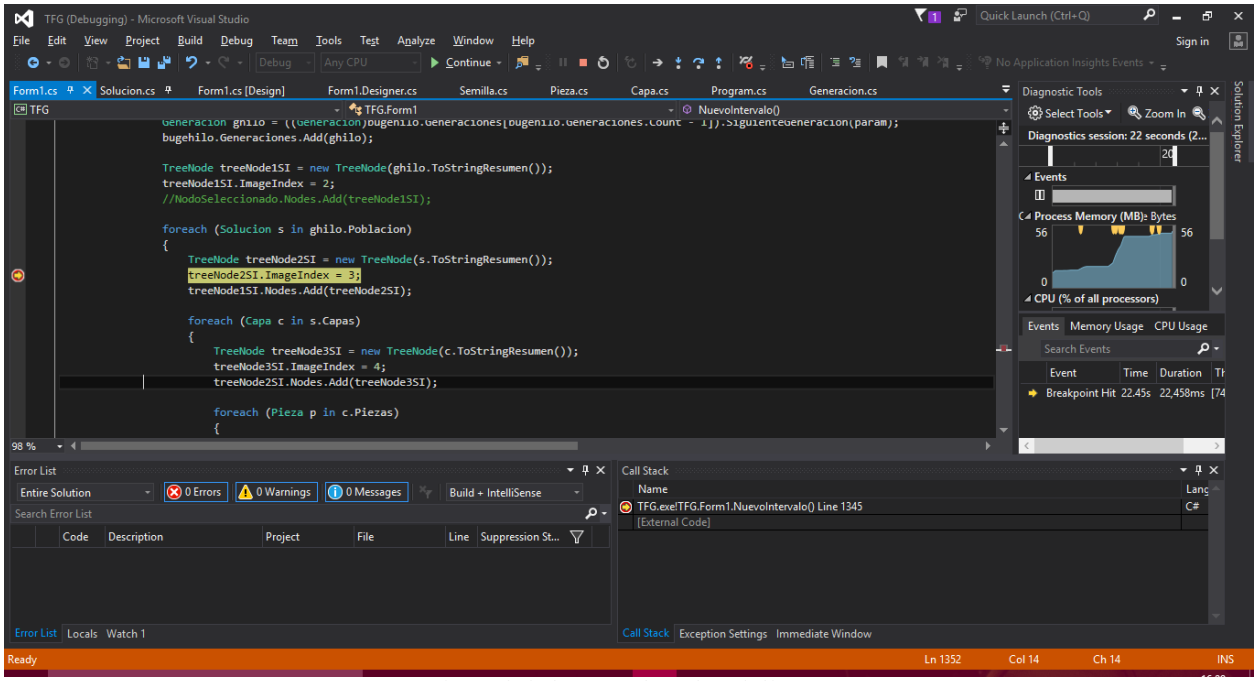
Este entorno integra todas las herramientas necesarias para un proyecto como este, de forma nativa y sin necesidad de plug-ins o complementos. Quizá para proyectos más avanzados o complejos sea necesario contar con dichas características, pero no es el caso.



Vista del código y del explorador de soluciones



Visor del diseñador de interfaz basado en arrastrar y soltar.



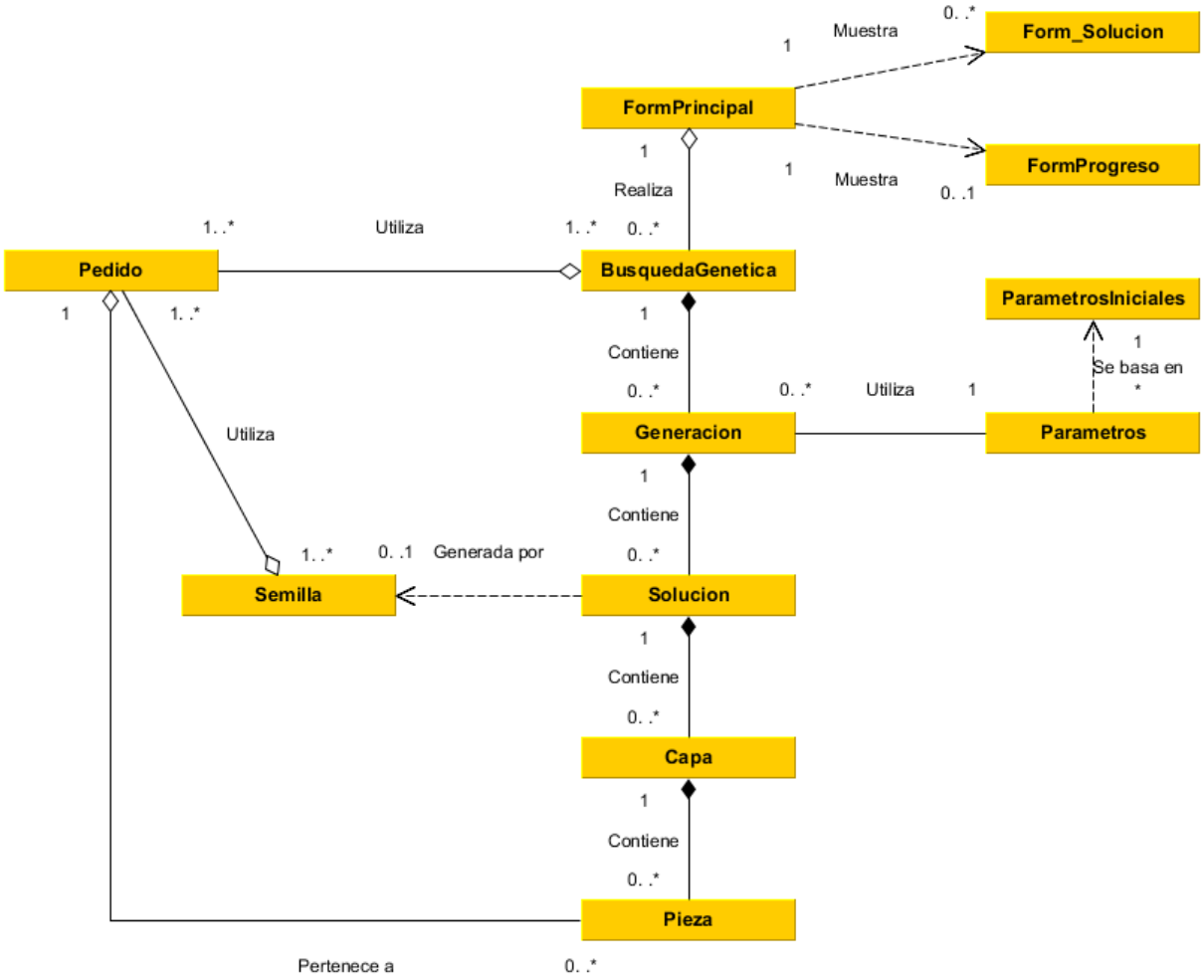
Vista del depurador de código.

4.2. Modelado e implementación de la solución

Tras el diseño conceptual del algoritmo se han identificado algunos elementos que claramente formarán el modelo de datos, como son las generaciones, las soluciones, las piezas etc. Por otro lado se deduce que vamos a necesitar una forma de representar los parámetros y además se necesitará una interfaz visual para interactuar con el algoritmo.

4.2.1. Esquema general de la solución implementada

Dado que hemos escogido realizar la implementación en un lenguaje orientado a objetos, cada uno de los elementos identificados estará representado por una clase. A continuación presentamos el diagrama de clases de UML que representa el modelo de clases de la solución.



Para explicar mejor el modelo hemos separado en tres grupos principales las clases:

1) Clases del modelo de datos

   a) BusquedaGenerica: Representa cada proceso de búsqueda que se realice en la aplicación. Contiene todas las variables y funciones necesarias para realizar el proceso de búsqueda y recuperar la información generada

   b) Generacion: Representa cada generación de un proceso de búsqueda. Contiene todas las variables y funciones necesarias para realizar el proceso de crear una nueva generación y recuperar su información

   c) Solucion: Representa una solución completa al problema que formará parte de una población de una generación. Contiene los datos que representan la solución y las funciones necesarias para calcular sus indicadores.

   d) Capa: Representa una capa concreta de solución. Contiene los datos que representan su contenido y la lógica necesaria para poder ampliar su contenido y recalcular sus indicadores.

   e) Pieza: Representa una pieza concreta de un pedido incluida en una capa de una solución. Contiene la información necesaria para representarla

   f) Pedido: Representa un conjunto de piezas con sus características y cantidad que el problema debe tener en cuenta para calcular la solución.

2) Clases de la interfaz de usuario

   a) FormPrincipal: Representa la interfaz principal del programa. Contiene todos los controles y lógica necesarios para interactuar con el modelo de datos.

   b) FormProgreso: Representa la interfaz que mostrará la información mientras el proceso de búsqueda está en marcha y permitirá cancelar el proceso en cualquier momento.

   c) Form Solucion: Representa la interfaz que mostrará una solución de manera visual y contiene los controles para interactuar con ella.

3) Clases auxiliares

   a) ParametrosIniciales: Se trata de una clase auxiliar que contiene los valores iniciales de cada uno de los parámetros cuando se inicia la aplicación. Servirá para inicializar los controles del formulario principal.

   b) Parametros: Es la representación de los parámetros concretos que se utilizará, en una búsqueda o generación determinada tomando sus valores desde el formulario principal.

   c) Semilla: Esta clase servirá para generar una solución de manera aleatoria en base al método escogido de los disponibles.

4.2.2. Detalles de las clases implementadas

Una vez explicado el modelo a nivel global, es necesario aportar los detalles más relevantes de su desarrollo. En este punto se representan en forma de figura las clases y sus componentes y en el caso de contener algún método cuya lógica sea compleja, se explica en el propio método el procedimiento que siguen.

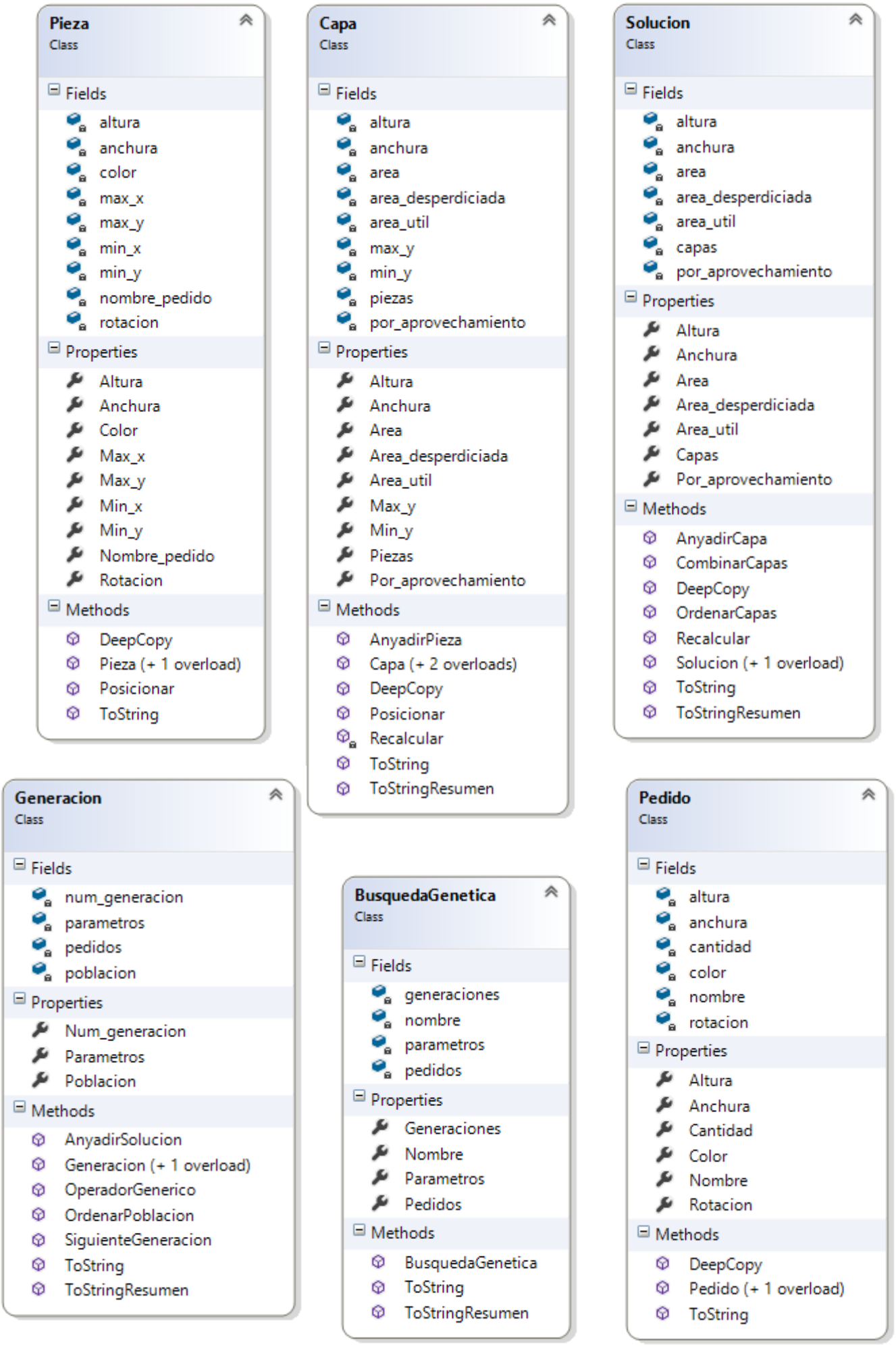
El detalle completo de cada una de las clases está disponible en el ANEXO I.

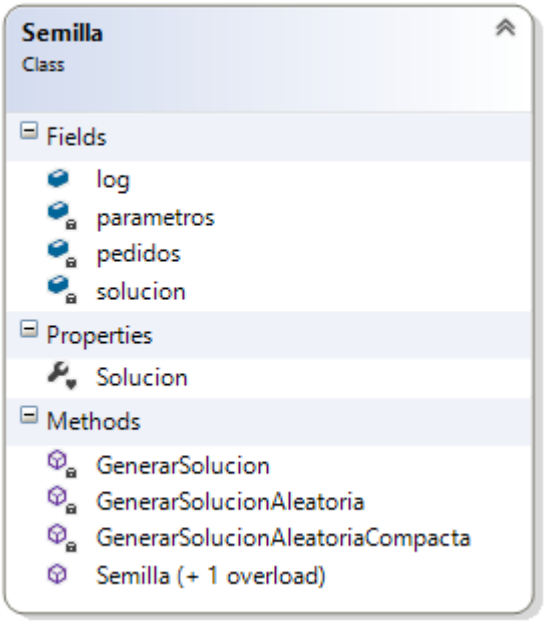
```
public class Capa
    {
        ...
        //Este método añade una pieza a la capa si se puede teniendo en
        cuenta el nº de cuchillas de la maquina. Si no puede retorna "false"
        public bool AnyadirPieza(Pieza p, int numCuchillas)
        {
                ...
                //Comprobamos que la pieza es igual o menor en altura que la
                que inició la capa o es la primera
                ...
                //Comprobamos si cabe en algún sitio
                //Comprobamos si se puede añadir al final de alguna de las
                existentes (de la primera nunca se podrá)
                ...
                //Si hay una pieza donde queramos colocar esta
                ...
                //Se comprueba que sea del mismo tipo y quepa a continuación
                (vertical)
                ...
                //Si no, comprobamos si se puede añadir al lado de la última
                (horizontal), si hay cuchillas libres
                ...
                //Si se puede añadir se añade
                ...
                //Y se devuelve el resultado de la operación
                ...
        }
        ...
    }
```

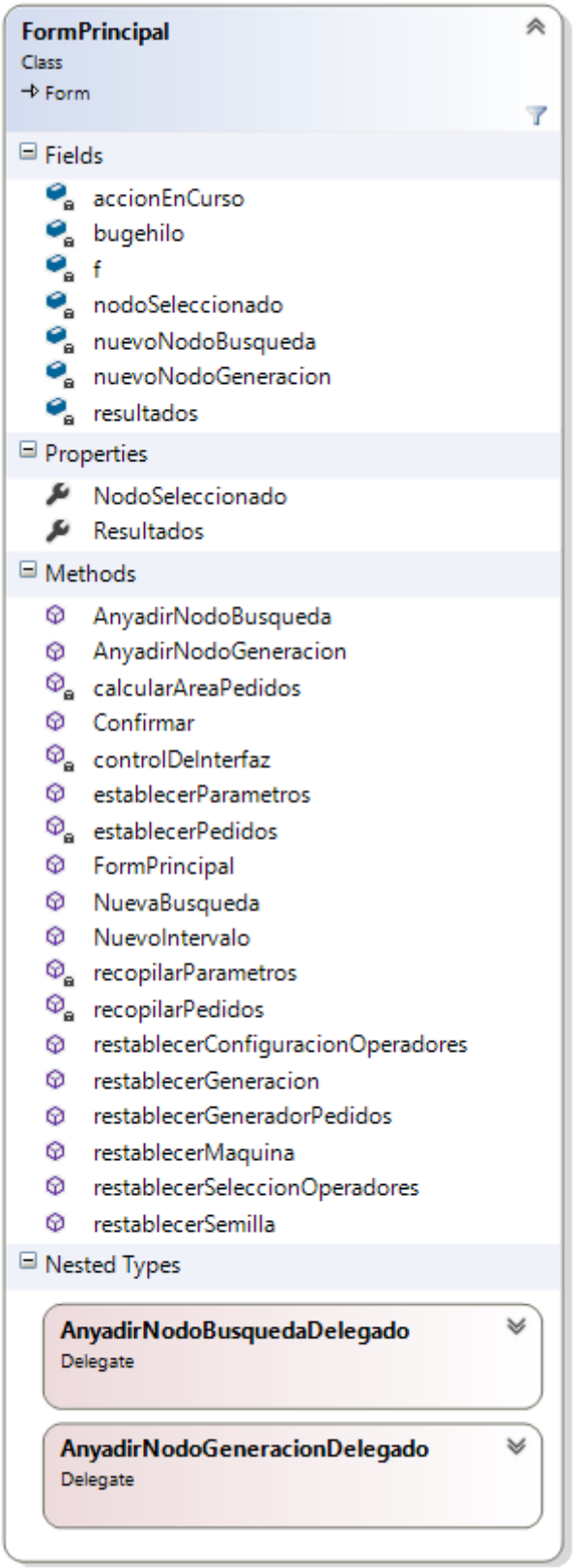
```
public class Generacion
    {
        //Este método ejecuta el algoritmo definido para evolucionar las
soluciones
        public Solucion OperadorGenerico(int por_aleatorio, int
por_mejores, int por_recombinacion, ArrayList soluciones, Parametros
parametros)
        {
                ...
                //Se crea una copia de los pedidos para poder ir descontando
                cantidades
                ...
                //Se crea una copia de las soluciones para poder modificar las
                capas que contiene
                ...
                //Se calcula el número medio de capas para poder trabajar con
                los porcentajes
                ...
```

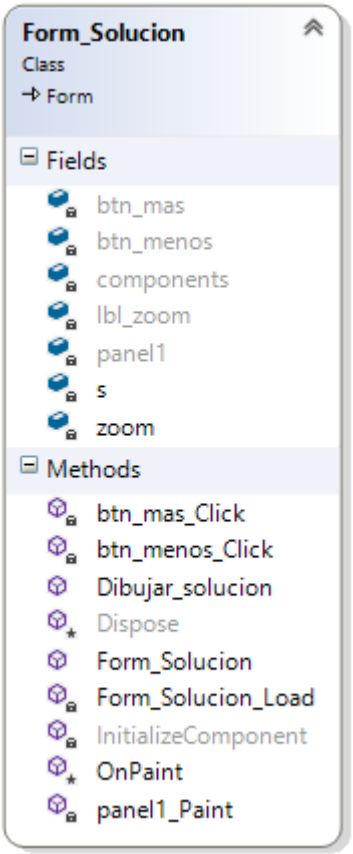
```
            //Se añaden las capas aleatorias de las soluciones fuente que
            aun no se hayan utilizado
            ...
                  //Tanto si cabe como si no, se descarta la capa
                  ...
                  //Se descuentan las piezas utilizadas
                  ...
                  //Si no se pude añadir, se vuelven a contar sus piezas
                  como pendientes
            ...
            //Se añaden las mejores capas de las soluciones fuente que aun
            no se hayan utilizado
            ...
                  //Tanto si cabe como si no, se descarta la capa
                  ...
                  //Se descuentan las piezas utilizadas
                  ...
                  //Si no se pude añadir, se vuelven a contar sus piezas
                  como pendientes
            ...
            //Se completa la solución hasta tener una solución completa
            ...
            //Se intenta recombinar dos capas al azar de la solución
            generada
            ...
           //Se devuelve la nueva solución
      }
      //Este método crea la siguiente generación en base a los parámetros
que recibe y las soluciones que contiene la actual generación
      public Generacion SiguienteGeneracion(Parametros parametros)
      {
            ...
            //Se copian las n primeras soluciones de la generación actual
            ...
            //Se ejecuta un operador de forma aleatoria hasta que se
            llegue al tamaño máximo de población
            ...
            //Se intenta la combinación de soluciones el numero indicado
            de veces
            ...
            //Se devuelve la nueva generación
            ...
      }
   }


public class Semilla
   {
      ...
     //Método que genera la solución mediante el método "Aleatoria"
      private void GenerarSolucionAleatoria()
      {
            ...
            //Elegir una pieza al azar
            ...
            //Se elige al azar si se crea una capa nueva o se intenta
            incorporar a una ya existente
            //Por estadística, las capas que se crean primero, tienen más
            probabilidad de llenarse que las últimas (han estado más veces
            en el bombo)
                  ...
                  //Si hay capas en las que quepa, se selecciona una al
                  azar y se incorpora la pieza
                  ...
```

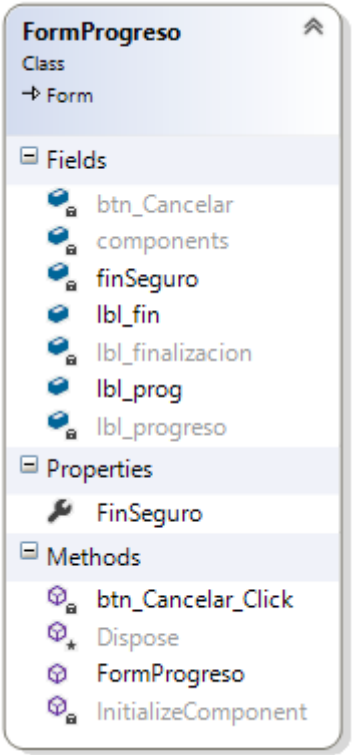
```
                    //Si no cabe, se descarta la capa y se prueba con la
                    siguiente
                    ...
                    //Si no cabe en ninguna capa o se decide crear una
                    nueva directamente, se crea una capa nueva con la nueva
                    pieza
                    ...
                    //Se resta 1 a las piezas pendientes de incorporar del
                    pedido correspondiente
                    ...
            }
        }
        //Método que genera la solución mediante el método "Aleatoria
Compacta". En este método solo se creará una nueva capa si no cabe en
ninguna de las existentes
        private void GenerarSolucionAleatoriaCompacta()
        {
             ...
             //Elegir una pieza al azar
             ...
             //Si hay capas en las que quepa, se selecciona una al azar y
             se incorpora la pieza
                    ...
                    //Si no cabe, se descarta la capa y se prueba con la
                    siguiente
                    ...
                    //Si no cabe en ninguna capa, se crea una capa nueva
                    con la nueva pieza
                    ...
                    //Se resta 1 a las piezas pendientes de incorporar del
                    pedido correspondiente
                    ...
            }
        }
        ...
    }
```


Pieza
Class
Fields
altura
anchura
color
max_x
max_y
min_x
min_y
nombre_pedido
rotacion
Properties
Altura
Anchura
Color
Max_x
Max_y
Min_x
Min_y
Nombre_pedido
Rotacion
Methods
DeepCopy
Pieza (+ 1 overload)
Posicionar
ToString
Capa
Class
Fields
altura
anchura
area
area_desperdiciada
area_util
max_y
min_y
piezas
por_aprovechamiento
Properties
Altura
Anchura
Area
Area_desperdiciada
Area_util
Max_y
Min_y
Piezas
Por_aprovechamiento
Methods
AnyadirPieza
Capa (+ 2 overloads)
DeepCopy
Posicionar
Recalcular
ToString
ToStringResumen
Solucion
Class
Fields
altura
anchura
area
area_desperdiciada
area_util
capas
por_aprovechamiento
Properties
Altura
Anchura
Area
Area_desperdiciada
Area_util
Capas
Por_aprovechamiento
Methods
AnyadirCapa
CombinarCapas
DeepCopy
OrdenarCapas
Recalcular
Solucion (+ 1 overload)
ToString
ToStringResumen
Generacion
Class
Fields
num_generacion
parametros
pedidos
poblacion
Properties
Num_generacion
Parametros
Poblacion
Methods
AnyadirSolucion
Generacion (+ 1 overload)
OperadorGenerico
OrdenarPoblacion
SiguienteGeneracion
ToString
ToStringResumen
BusquedaGenetica
Class
Fields
generaciones
nombre
parametros
pedidos
Properties
Generaciones
Nombre
Parametros
Pedidos
Methods
BusquedaGenetica
ToString
ToStringResumen
Pedido
Class
Fields
altura
anchura
cantidad
color
nombre
rotacion
Properties
Altura
Anchura
Cantidad
Color
Nombre
Rotacion
Methods
DeepCopy
Pedido (+ 1 overload)
ToString

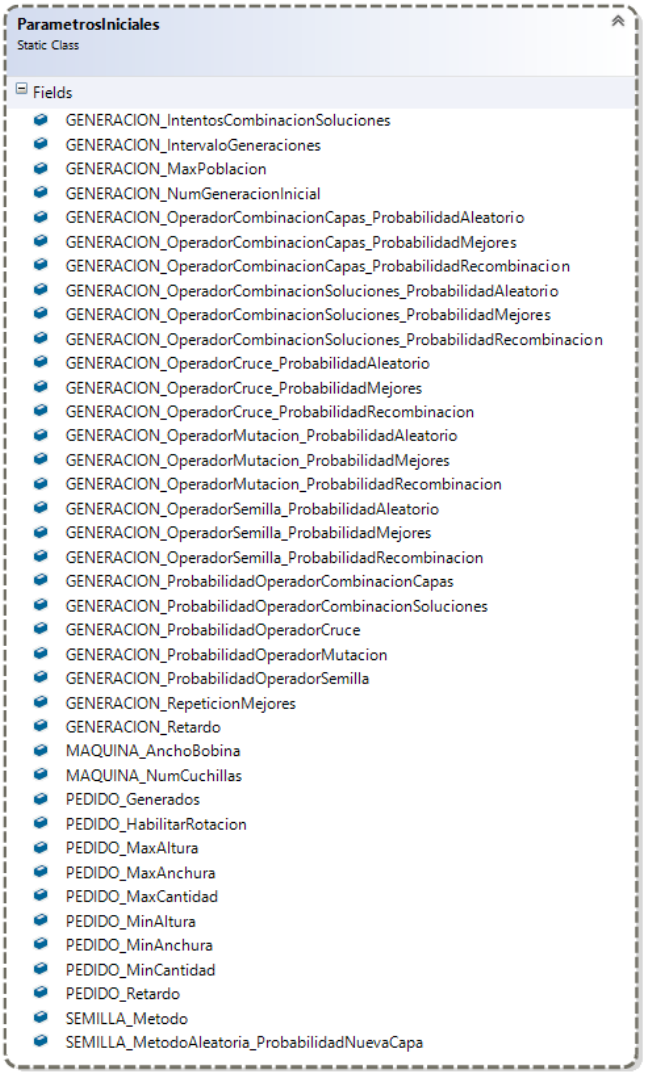


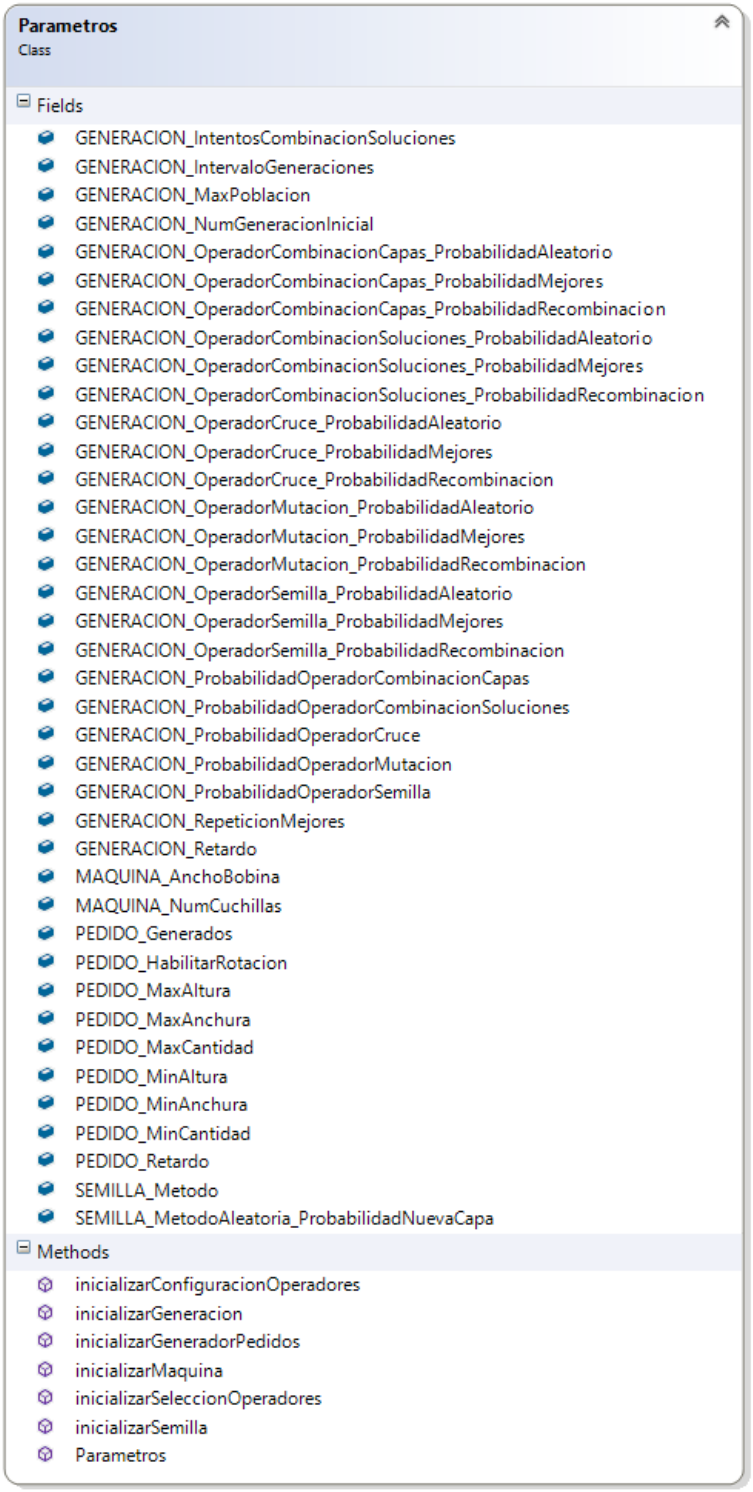






En la imagen representativa de la clase FormPrincipal, se han ocultado los ítems correspondientes a controles de la interfaz y sus métodos para no excederse en tamaño.


ParametrosIniciales
Static Class
Fields
GENERACION_IntentosCombinacionSoluciones
GENERACION_IntervaloGeneraciones
GENERACION_MaxPoblacion
GENERACION_NumGeneracionInicial
GENERACION_OperadorCombinacionCapas_ProbabilidadAleatorio
GENERACION_OperadorCombinacionCapas_ProbabilidadMejores
GENERACION_OperadorCombinacionCapas_ProbabilidadRecombinacion
GENERACION_OperadorCombinacionSoluciones_ProbabilidadAleatorio
GENERACION_OperadorCombinacionSoluciones_ProbabilidadMejores
GENERACION_OperadorCombinacionSoluciones_ProbabilidadRecombinacion
GENERACION_OperadorCruce_ProbabilidadAleatorio
GENERACION_OperadorCruce_ProbabilidadMejores
GENERACION_OperadorCruce_ProbabilidadRecombinacion
GENERACION_OperadorMutacion_ProbabilidadAleatorio
GENERACION_OperadorMutacion_ProbabilidadMejores
GENERACION_OperadorMutacion_ProbabilidadRecombinacion
GENERACION_OperadorSemilla_ProbabilidadAleatorio
GENERACION_OperadorSemilla_ProbabilidadMejores
GENERACION_OperadorSemilla_ProbabilidadRecombinacion
GENERACION_ProbabilidadOperadorCombinacionCapas
GENERACION_ProbabilidadOperadorCombinacionSoluciones
GENERACION_ProbabilidadOperadorCruce
GENERACION_ProbabilidadOperadorMutacion
GENERACION_ProbabilidadOperadorSemilla
GENERACION_RepeticionMejores
GENERACION_Retardo
MAQUINA_AnchoBobina
MAQUINA_NumCuchillas
PEDIDO_Generados
PEDIDO_HabilitarRotacion
PEDIDO_MaxAltura
PEDIDO_MaxAnchura
PEDIDO_MaxCantidad
PEDIDO_MinAltura
PEDIDO_MinAnchura
PEDIDO_MinCantidad
PEDIDO_Retardo
SEMILLA_Metodo
SEMILLA_MetodoAleatoria_ProbabilidadNuevaCapa


Parametros
Class
Fields
GENERACION_IntentosCombinacionSoluciones
GENERACION_IntervaloGeneraciones
GENERACION_MaxPoblacion
GENERACION_NumGeneracionInicial
GENERACION_OperadorCombinacionCapas_ProbabilidadAleatorio
GENERACION_OperadorCombinacionCapas_ProbabilidadMejores
GENERACION_OperadorCombinacionCapas_ProbabilidadRecombinacion
GENERACION_OperadorCombinacionSoluciones_ProbabilidadAleatorio
GENERACION_OperadorCombinacionSoluciones_ProbabilidadMejores
GENERACION_OperadorCombinacionSoluciones_ProbabilidadRecombinacion
GENERACION_OperadorCruce_ProbabilidadAleatorio
GENERACION_OperadorCruce_ProbabilidadMejores
GENERACION_OperadorCruce_ProbabilidadRecombinacion
GENERACION_OperadorMutacion_ProbabilidadAleatorio
GENERACION_OperadorMutacion_ProbabilidadMejores
GENERACION_OperadorMutacion_ProbabilidadRecombinacion
GENERACION_OperadorSemilla_ProbabilidadAleatorio
GENERACION_OperadorSemilla_ProbabilidadMejores
GENERACION_OperadorSemilla_ProbabilidadRecombinacion
GENERACION_ProbabilidadOperadorCombinacionCapas
GENERACION_ProbabilidadOperadorCombinacionSoluciones
GENERACION_ProbabilidadOperadorCruce
GENERACION_ProbabilidadOperadorMutacion
GENERACION_ProbabilidadOperadorSemilla
GENERACION_RepeticionMejores
GENERACION_Retardo
MAQUINA_AnchoBobina
MAQUINA_NumCuchillas
PEDIDO_Generados
PEDIDO_HabilitarRotacion
PEDIDO_MaxAltura
PEDIDO_MaxAnchura
PEDIDO_MaxCantidad
PEDIDO_MinAltura
PEDIDO_MinAnchura
PEDIDO_MinCantidad
PEDIDO_Retardo
SEMILLA_Metodo
SEMILLA_MetodoAleatoria_ProbabilidadNuevaCapa
Methods
inicializarConfiguracionOperadores
inicializarGeneracion
inicializarGeneradorPedidos
inicializarMaquina
inicializarSeleccionOperadores
inicializarSemilla
Parametros

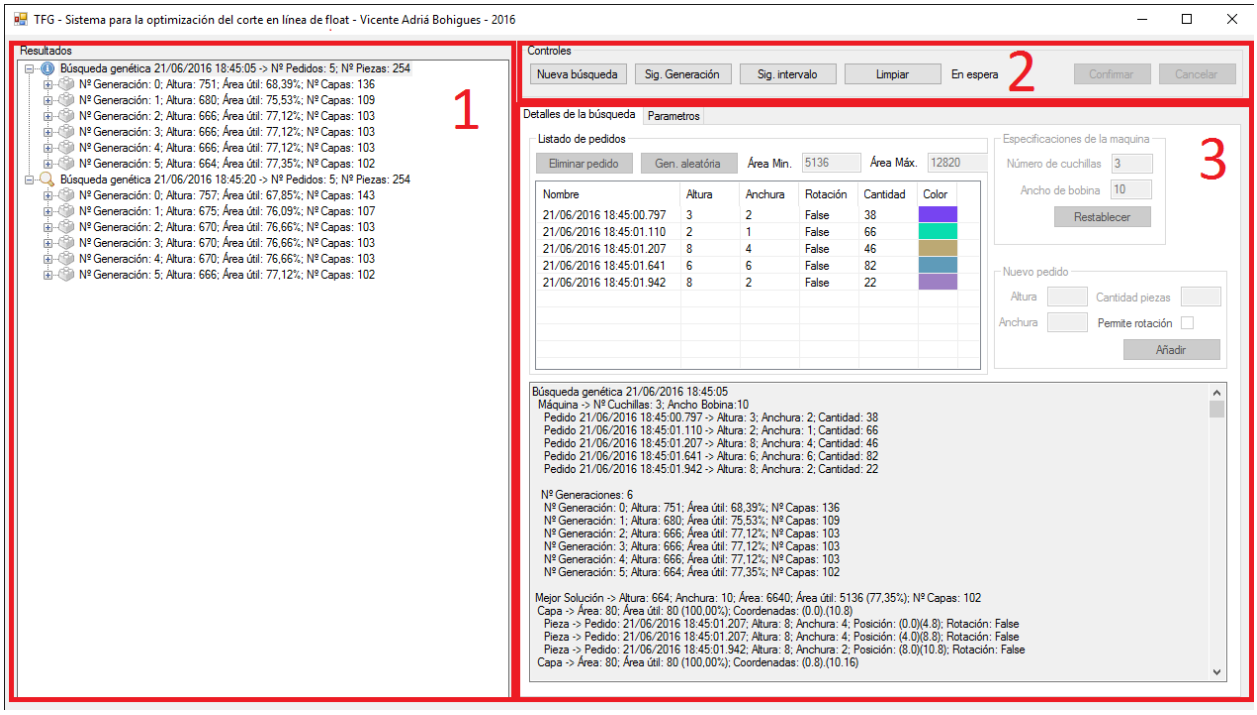
5. Interfaz de la aplicación y su funcionamiento

En este punto se describen todos los elementos que describen la interfaz de usuario y posteriormente se explica a modo de un pequeño manual las distintas funcionalidades que componen la aplicación.

5.1. Descripción de la interfaz:

La interfaz aquí descrita se ha desarrollado con unos mínimos controles de validación sobre la información introducida, sin embargo al no ser el objetivo del TFG tener una aplicación robusta a prueba de fallos en lo relativo a la introducción de información, podrían producirse algunos errores no controlados.

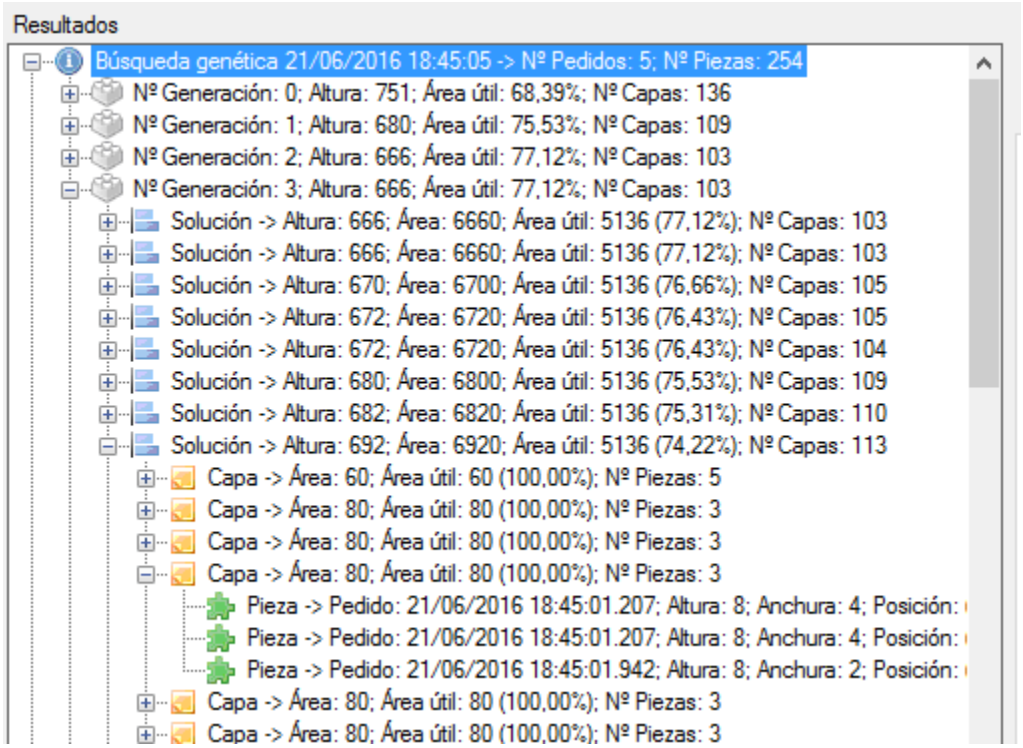
5.1.1. Pantalla principal



La pantalla principal está dividida en tres grandes bloques:

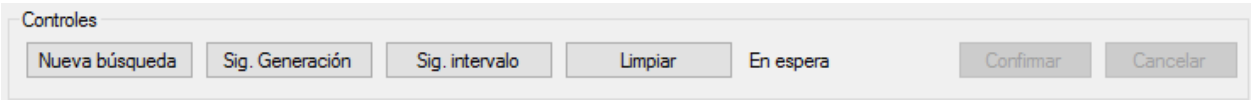
1. Área de resultados
2. Área de controles
3. Área de parámetros y detalles

En las siguientes páginas se mostrará cada una de ellas con detalle

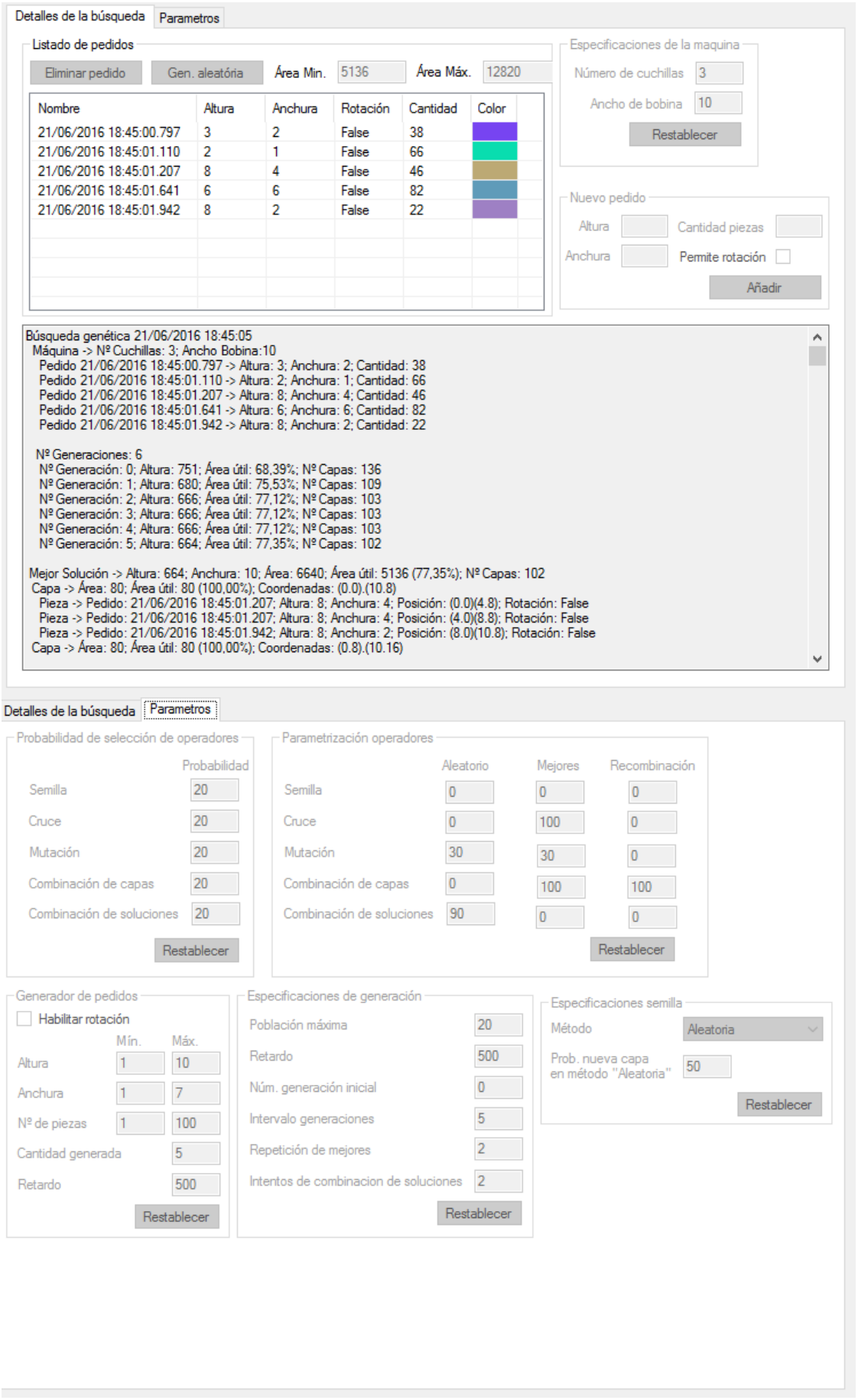


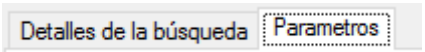
Esta es la área donde se van añadiendo los resultados de las búsquedas que se realicen en la aplicación. Se trata de un control en forma de árbol en el que cada nivel de anidamiento equivale a un nivel del modelo de datos de la solución: Búsqueda, Generación, Capa y Pieza.

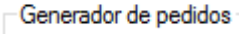
Pulsando dos veces sobre cualquier ítem, se desplegara o soltará su contenido, con la particularidad de que si se trata de un ítem solución, se mostrará una pantalla emergente en la que se representa visualmente dicha solución.



En el área de controles se muestran las funcionalidades principales de la aplicación. Se trata de una botonera separada en dos bloques por una etiqueta de tipo texto en la que el bloque de la izquierda representa las acciones a realizar, la etiqueta muestra que acción se está realizando en cada momento y el bloque de la derecha permite confirmar o cancelar una acción. Este último bloque solo se activará en el momento que se pulse en una acción principal, hasta que se confirme o cancele su ejecución. Mientras haya una acción pendiente de confirmar o cancelar el bloqueo de la izquierda permanecerá desactivado.


Detalles de la búsqueda
Parametros
Listado de pedidos
Eliminar pedido
Gen. aleatória
Área Min.
5136
Área Máx.
12820
Nombre
Altura
Anchura
Rotación
Cantidad
Color
21/06/2016 18:45:00.797 3 2 False 38
21/06/2016 18:45:01.110 2 1 False 66
21/06/2016 18:45:01.207 8 4 False 46
21/06/2016 18:45:01.641 6 6 False 82
21/06/2016 18:45:01.942 8 2 False 22
Especificaciones de la maquina
Número de cuchillas
3
Ancho de bobina
10
Restablecer
Nuevo pedido
Altura
Cantidad piezas
Anchura
Permite rotación
Añadir
Búsqueda genética 21/06/2016 18:45:05
Máquina -> Nº Cuchillas: 3; Ancho Bobina:10
Pedido 21/06/2016 18:45:00.797 -> Altura: 3; Anchura: 2; Cantidad: 38
Pedido 21/06/2016 18:45:01.110 -> Altura: 2; Anchura: 1; Cantidad: 66
Pedido 21/06/2016 18:45:01.207 -> Altura: 8; Anchura: 4; Cantidad: 46
Pedido 21/06/2016 18:45:01.641 -> Altura: 6; Anchura: 6; Cantidad: 82
Pedido 21/06/2016 18:45:01.942 -> Altura: 8; Anchura: 2; Cantidad: 22
Nº Generaciones: 6
Nº Generación: 0; Altura: 751; Área útil: 68,39%; Nº Capas: 136
Nº Generación: 1; Altura: 680; Área útil: 75,53%; Nº Capas: 109
Nº Generación: 2; Altura: 666; Área útil: 77,12%; Nº Capas: 103
Nº Generación: 3; Altura: 666; Área útil: 77,12%; Nº Capas: 103
Nº Generación: 4; Altura: 666; Área útil: 77,12%; Nº Capas: 103
Nº Generación: 5; Altura: 664; Área útil: 77,35%; Nº Capas: 102
Mejor Solución -> Altura: 664; Anchura: 10; Área: 6640; Área útil: 5136 (77,35%); Nº Capas: 102
Capa -> Área: 80; Área útil: 80 (100,00%); Coordenadas: (0.0).(10.8)
Pieza -> Pedido: 21/06/2016 18:45:01.207; Altura: 8; Anchura: 4; Posición: (0.0)(4.8); Rotación: False
Pieza -> Pedido: 21/06/2016 18:45:01.207; Altura: 8; Anchura: 4; Posición: (4.0)(8.8); Rotación: False
Pieza -> Pedido: 21/06/2016 18:45:01.942; Altura: 8; Anchura: 2; Posición: (8.0)(10.8); Rotación: False
Capa -> Área: 80; Área útil: 80 (100,00%); Coordenadas: (0.8).(10.16)
Detalles de la búsqueda
Parametros
Probabilidad de selección de operadores
Probabilidad
Semilla 20
Cruce 20
Mutación 20
Combinación de capas 20
Combinación de soluciones 20
Restablecer
Parametrización operadores
Aleatorio
Mejores
Recombinación
Semilla 0 0 0
Cruce 0 100 0
Mutación 30 30 0
Combinación de capas 0 100 100
Combinación de soluciones 90 0 0
Restablecer
Generador de pedidos
Habilitar rotación
Mín.
Máx.
Altura 1 10
Anchura 1 7
Nº de piezas 1 100
Cantidad generada 5
Retardo 500
Restablecer
Especificaciones de generación
Población máxima 20
Retardo 500
Núm. generación inicial 0
Intervalo generaciones 5
Repetición de mejores 2
Intentos de combinacion de soluciones 2
Restablecer
Especificaciones semilla
Método
Aleatoria
Prob. nueva capa en método "Aleatoria"
50
Restablecer

En la tercera área, separados en dos pestañas 

y bloques 

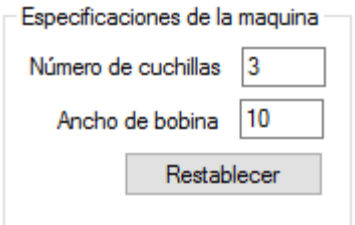
los distintos parámetros que afectan al algoritmo, algunas funcionalidades dependientes de las funcionalidades principales y un gran campo de texto en el que se muestra el detalle del nodo seleccionado del árbol de resultados.

En la pestaña de detalles de la búsqueda podemos encontrar los siguientes grupos

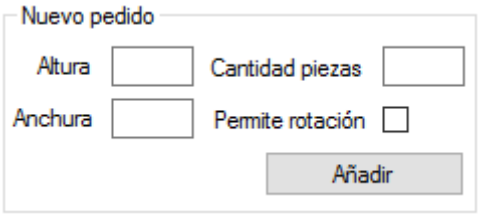
En el grupo de Listado de pedidos encontramos todos los pedidos que se hayan introducido hasta el momento para realizar la búsqueda. De cada pedido se representan sus atributos, incluyendo su color de forma visual.

En la parte superior derecha tenemos dos campos que muestran el área mínima y máxima ocupada por las piezas de los pedidos. El área mínima corresponde a la suma de las áreas de las piezas, y sería el área utilizada en el caso que encontrásemos una solución que utilice el 100% del material. El área máxima representa el área de material necesario para producir todas las piezas en el peor de los casos, es decir en el caso que que solo se produjese una pieza en cada capa y se pusiera todas ellas, una a continuación de la otra.

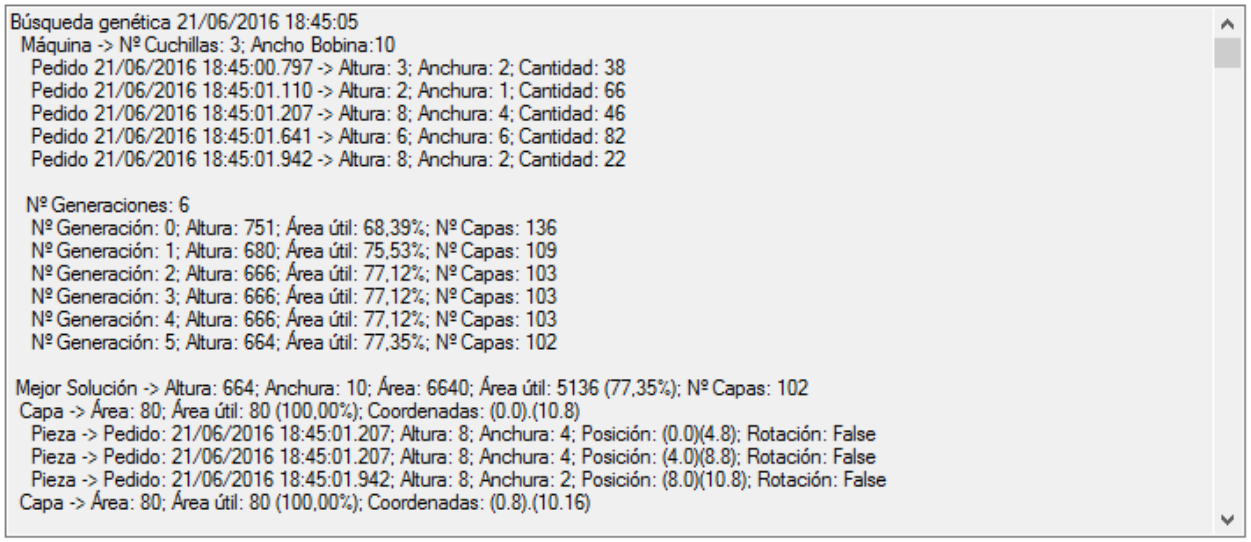
Por último en la parte superior izquierda encontramos dos botones que nos permiten, en el caso de encontrarnos en un proceso de nueva búsqueda que esté aún por confirmar, eliminar pedidos de la lista o generar un conjunto de pedidos aleatorios según las especificaciones del generador de pedidos.



En el bloque de Especificaciones de la máquina, encontramos los dos parámetros que servirán para definir la máquina sobre la que planificar soluciones para nuestros pedidos. Aquí se establecen el número de cuchillas de la máquina y el ancho del material. También se cuenta con un botón que permite restablecer sus valores a los definidos como por defecto.

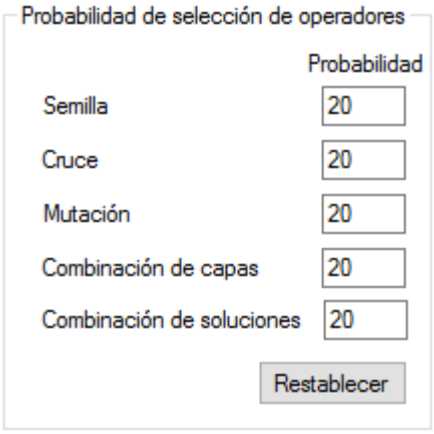


El bloque de nuevo pedido complementa a los controles del Listado de pedido de manera que cuando se pueden introducir nuevos pedidos, este bloque nos permite agregar un pedido con todos sus parámetros de forma manual.

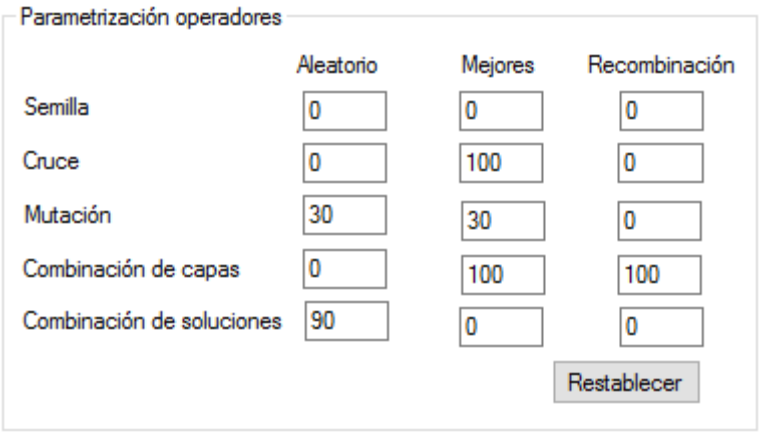


En el campo texto siempre aparecerá la información detallada del nodo que haya seleccionado en el árbol, adaptando en cada caso la información mostrada según el nivel del nodo que se seleccione.

En cuanto a la pestaña de parámetros podemos encontrar los siguientes grupos



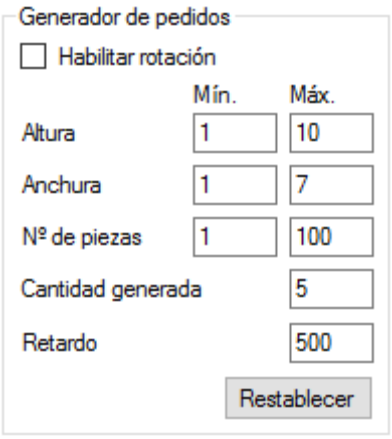
El grupo Probabilidad de selección de operadores permite especificar con qué probabilidad se va a seleccionar un operador u otro cuando el proceso de búsqueda deba elegir entre operadores. Hay que tener en cuenta que entre todos los campos se debe sumar un valor del 100%. En caso contrario se pueden producir resultados inesperados en cuanto a probabilidades se refiere.



En el bloque de parametrización de operadores se muestra la configuración por defecto del operador genérico para que funcione como si de los operadores concretos se tratara. Sin embargo se ofrece la posibilidad al usuario de establecer su propia parametrización para probar diferentes configuraciones y explorar los resultados del algoritmo.

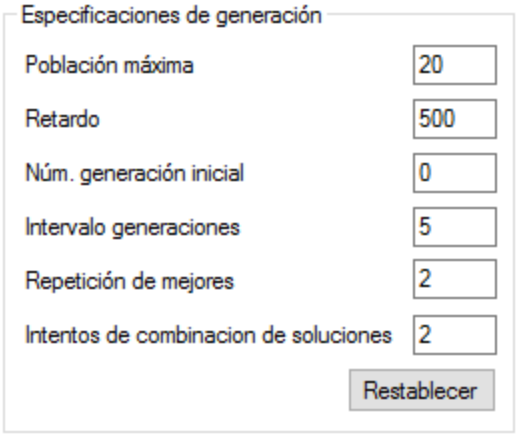
En este bloque encontramos los porcentajes de capas aleatorias y mejores que copiaran los operadores de las soluciones de la generación anterior. Si ambos valores no llegan al 100% se completara la solución generada con el algoritmo semilla.

La última columna del bloque representa la probabilidad de que al final de la ejecución del operador se haga un intento de recombinación de capas.

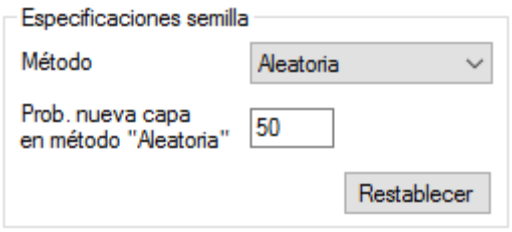


Este bloque corresponde a la parametrización del Generador de pedidos, que no es más que los parámetros que se tendrán en cuenta cuando se pulse en el botón de generación aleatoria de pedidos.

En este bloque se pueden establecer si los pedidos admitirán rotación o no, los valores mínimos y máximos de altura, anchura y cantidad de piezas del pedido, la cantidad de pedidos que se generan cada vez que se pulse el botón, y se puede indicar un retardo (en milisegundos) entre la generación de un periodo y otro ya que el generador depende de una variable aleatoria inicializada con el reloj del sistema.

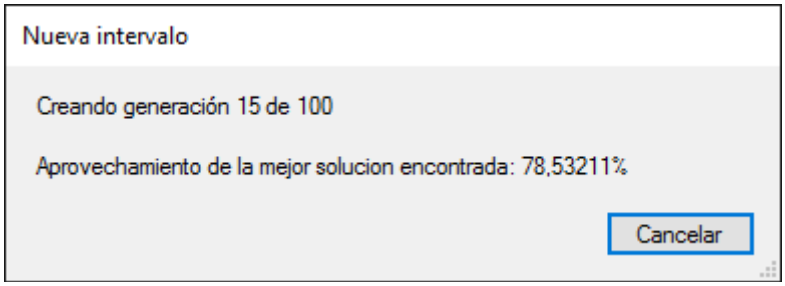


El bloque de Especificaciones de generación, permite definir las variables que parametrizan la creación de nuevas generaciones. En concreto se puede definir la población de cada generación, el retardo entre soluciones generadas por el algoritmo semilla, el número de generación inicial, cuántas generaciones se crearán cada vez que se pulse el botón Intervalo de generaciones, cuántas de las mejores soluciones pasarán a la siguiente generación y cuántos intentos de combinación de soluciones se realizarán antes de dar por creada una nueva generación.



En el bloque de Especificaciones de la semilla se puede establecer el método de generación de soluciones aleatorias y en el caso del método “aleatoria” con qué probabilidad se creará una nueva capa.

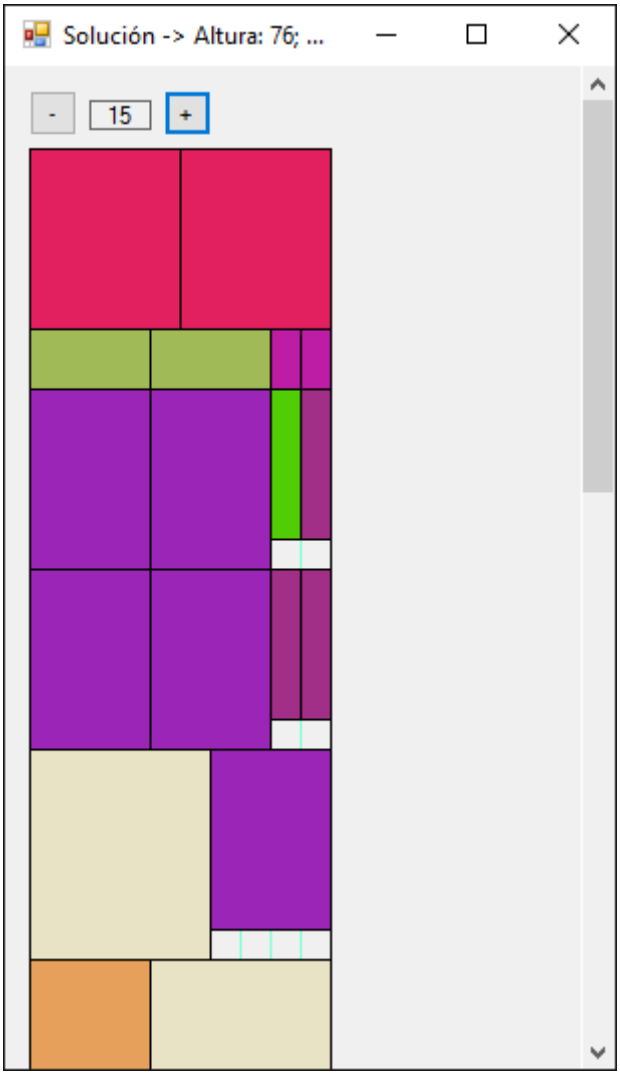
5.1.2. Pantalla de progreso



En esta pantalla se muestra como título, la acción que se está realizando y en el cuerpo se muestra el progreso de la solución, tanto la generación por la que va el proceso del intervalo definido como la calidad de la mejor solución encontrada hasta el momento.

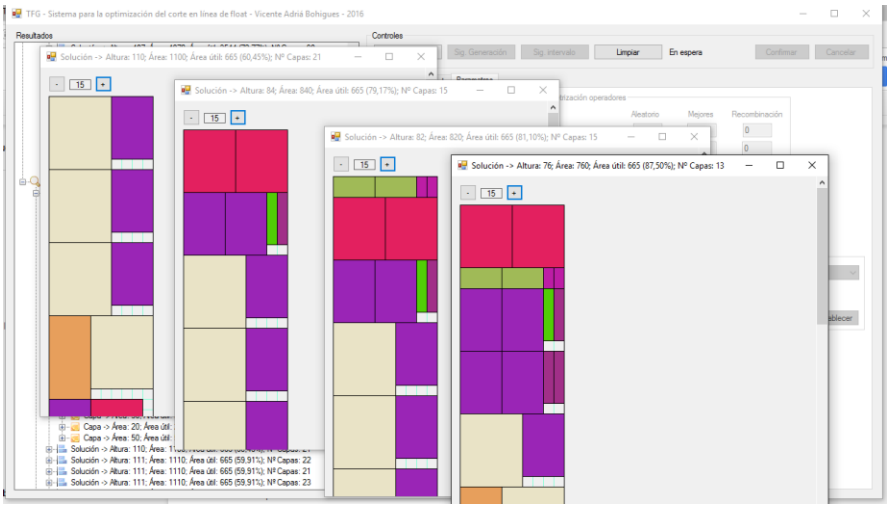
Esta pantalla cuenta con un botón cancelar para parar el proceso en cualquier momento, bien porque tarde demasiado el proceso o porque la calidad encontrada sea suficiente o no evolucione. El proceso siempre terminará con una generación completa para no dejar resultados a medias.

5.1.3. Pantalla de solución



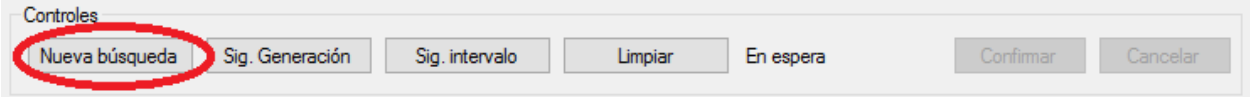
En esta pantalla se muestra la solución seleccionada de manera visual con los colores establecidos para cada pedido. Esta ventana se puede redimensionar para ver mayor cantidad de la solución y también ofrece controles para ampliar o reducir el zoom con el que se representa la solución.

Por último, mencionar que se pueden tener varias ventanas de solución abiertas para poder realizar comparaciones entre ella.



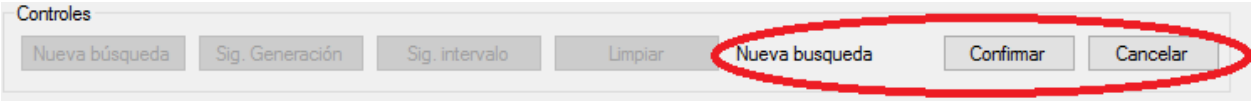
5.2. Funcionalidades de la aplicación:

5.2.1. Nueva búsqueda

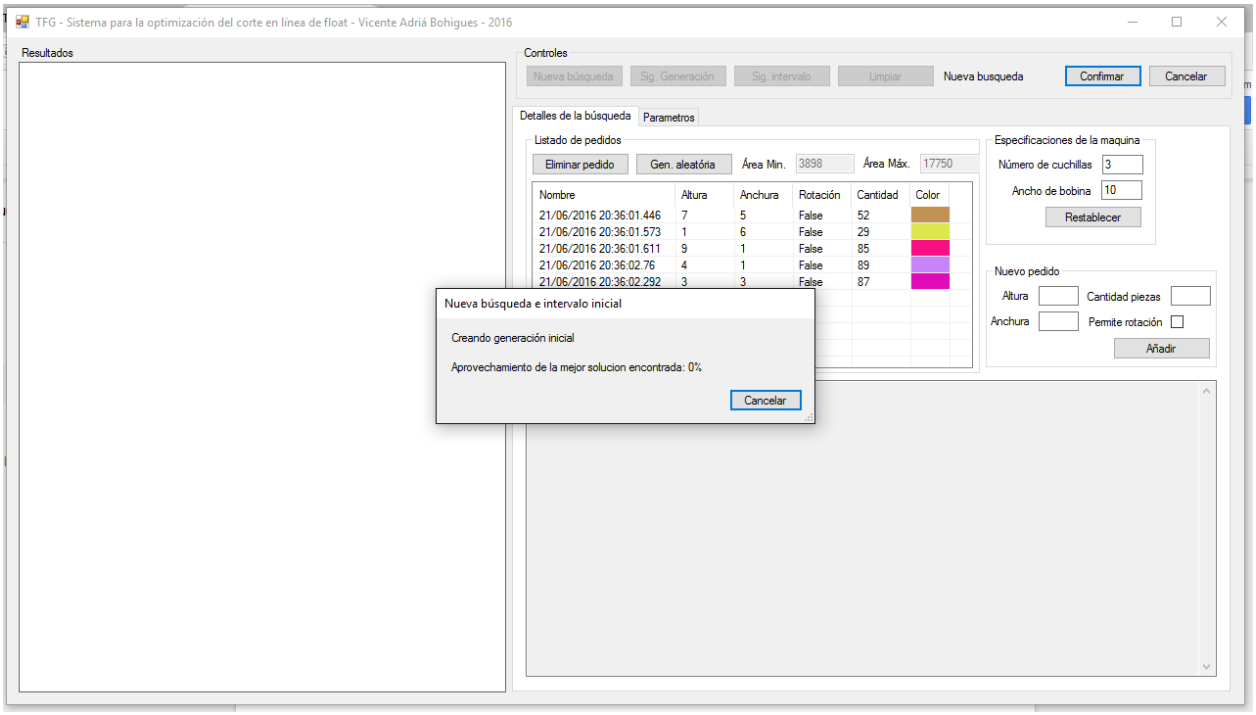


La principal funcionalidad de la aplicación es realizar búsquedas con unos determinados parámetros para satisfacer un conjunto de pedidos con la menor cantidad de material posible. Para iniciar una nueva búsqueda basta con pulsar el botón correspondiente. En ese momento se habilitarán todos los controles necesarios para gestionar los pedidos y parámetros del problema.

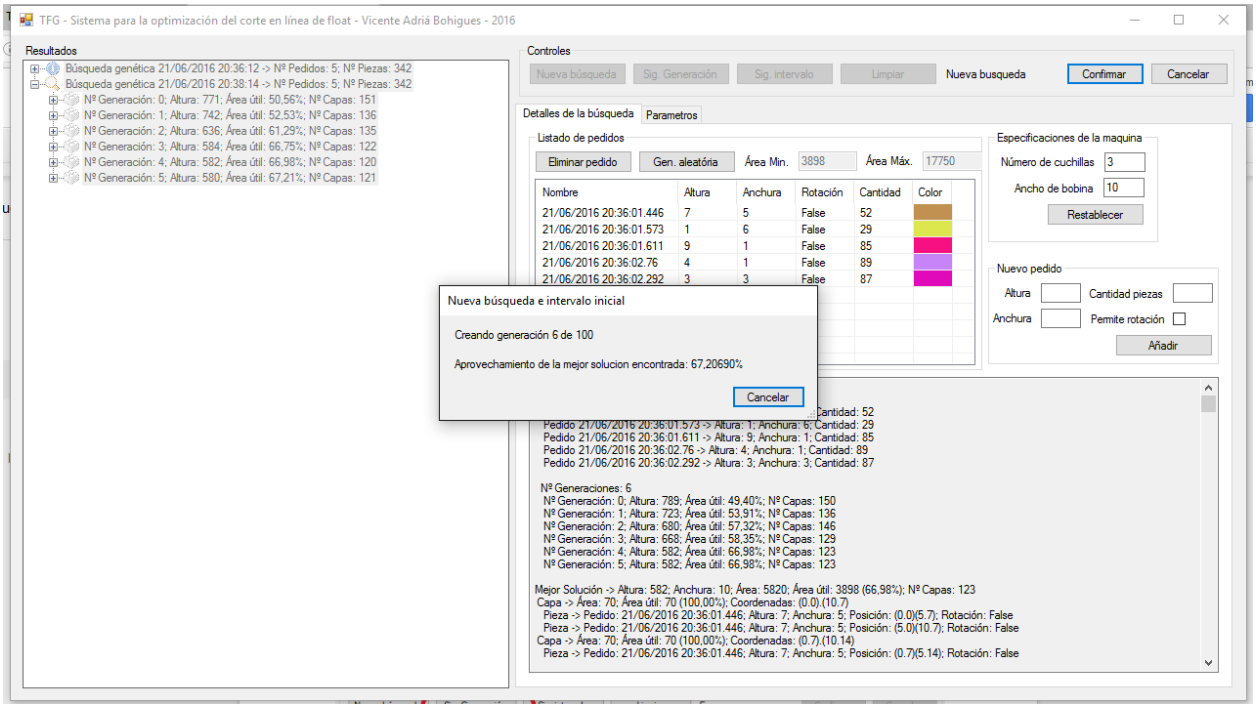
Una vez definidos los pedidos y los parámetros tal y como queremos para proceder a la búsqueda, basta con pulsar en el botón confirmar para iniciar el proceso o en el botón cancelar para volver al estado de espera.

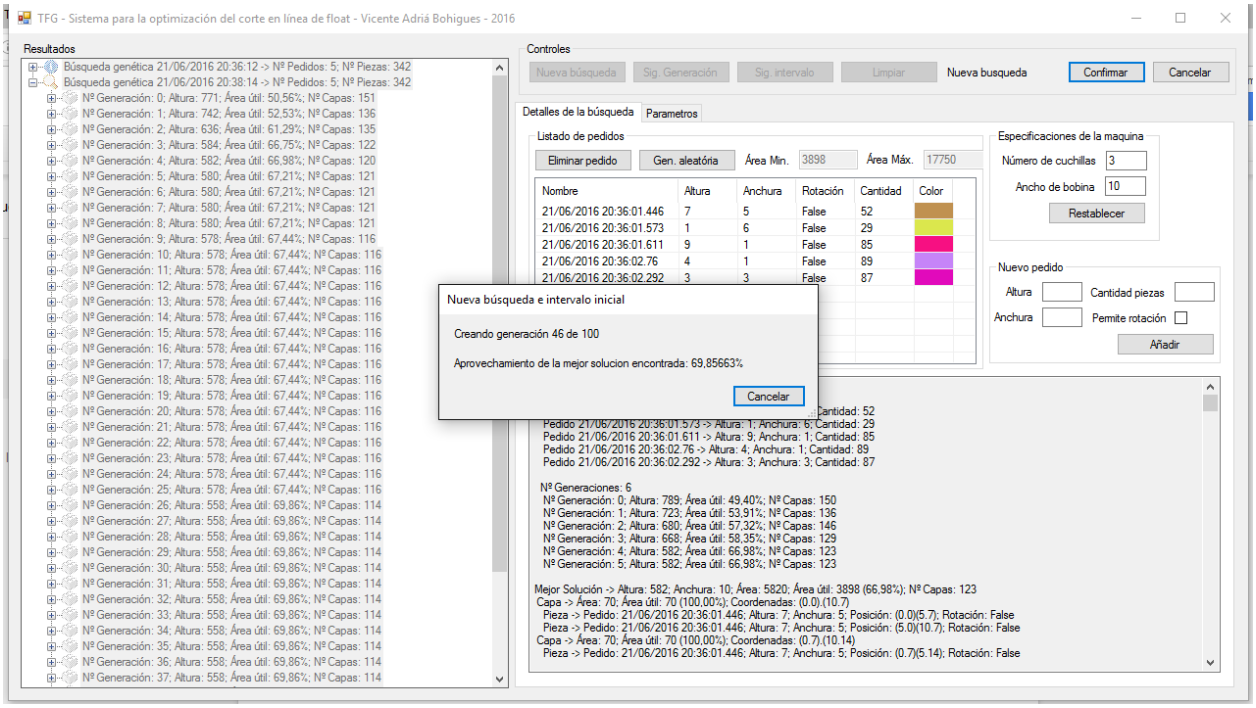


Si iniciamos el proceso de búsqueda se mostrará la ventana de progreso para indicarnos cómo va el proceso. En ese momento podremos esperar a que se complete el proceso o abortarlo cuando decidamos.

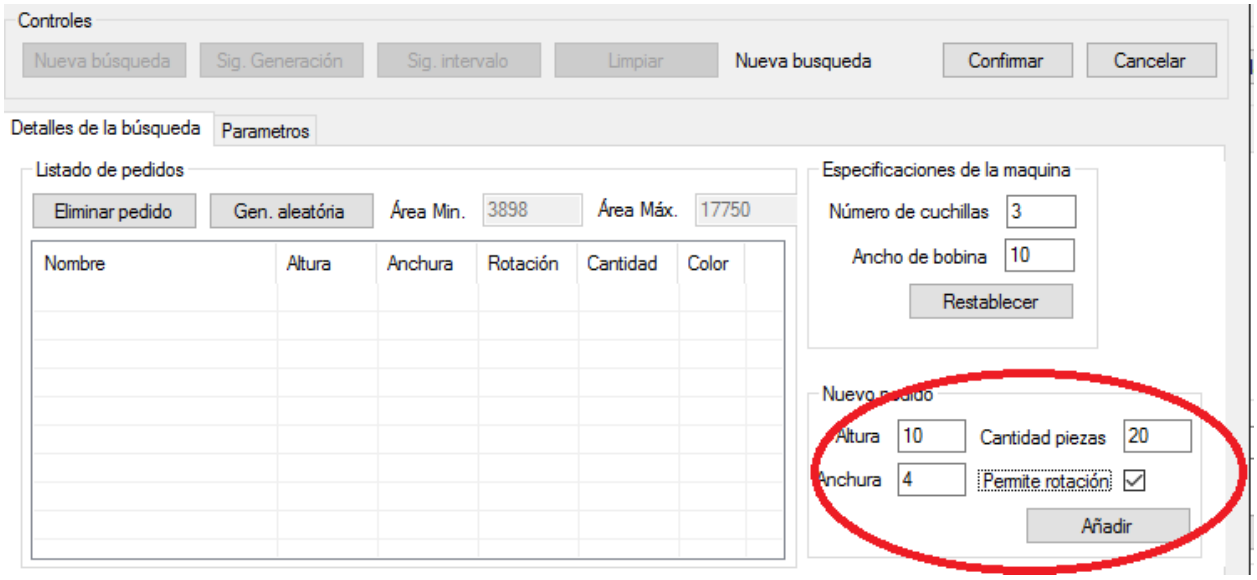


Conforme avance el proceso veremos que se va rellenando el árbol de resultados en tiempo real

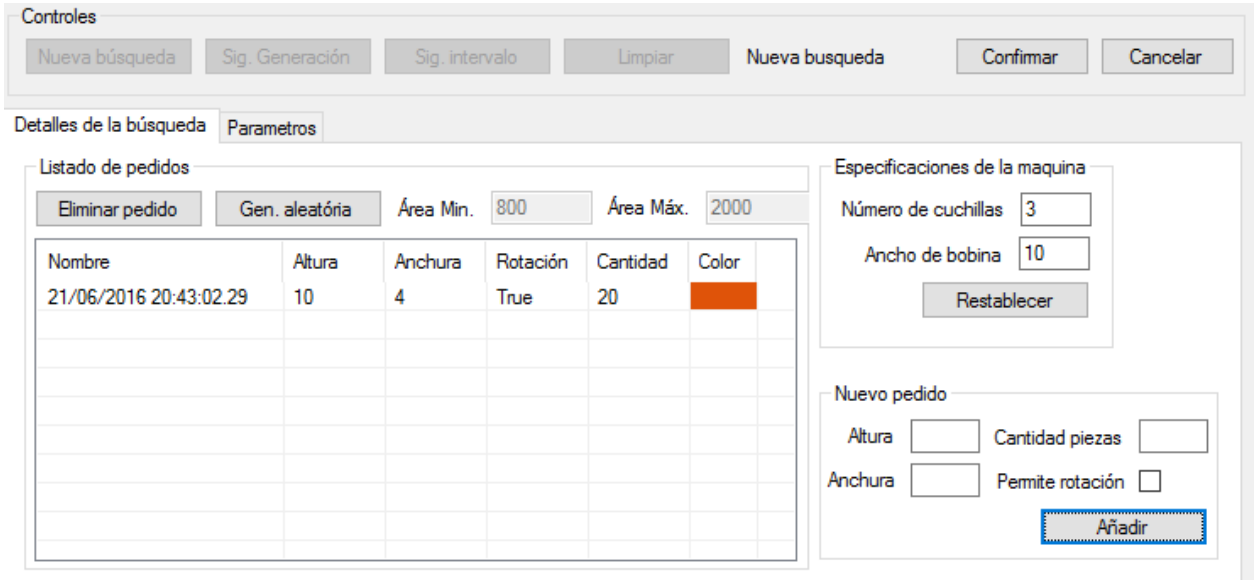

TFG - Sistema para la optimización del corte en línea de float - Vicente Adriá Bohigues - 2016
Resultados
Búsqueda genética 21/06/2016 20:36:12 -> Nº Pedidos: 5; Nº Piezas: 342
Búsqueda genética 21/06/2016 20:38:14 -> Nº Pedidos: 5; Nº Piezas: 342
Nº Generación: 0; Altura: 771; Área útil: 50,56%; Nº Capas: 151
Nº Generación: 1; Altura: 742; Área útil: 52,53%; Nº Capas: 136
Nº Generación: 2; Altura: 636; Área útil: 61,29%; Nº Capas: 135
Nº Generación: 3; Altura: 584; Área útil: 66,75%; Nº Capas: 122
Nº Generación: 4; Altura: 582; Área útil: 66,98%; Nº Capas: 120
Nº Generación: 5; Altura: 580; Área útil: 67,21%; Nº Capas: 121
Controles
Nueva búsqueda
Sig. Generación
Sig. intervalo
Limpiar
Nueva busqueda
Confirmar
Cancelar
Detalles de la búsqueda
Parametros
Listado de pedidos
Eliminar pedido
Gen. aleatória
Área Min.
3898
Área Máx.
17750
Nombre
Altura
Anchura
Rotación
Cantidad
Color
21/06/2016 20:36:01.446 7 5 False 52
21/06/2016 20:36:01.573 1 6 False 29
21/06/2016 20:36:01.611 9 1 False 85
21/06/2016 20:36:02.76 4 1 False 89
21/06/2016 20:36:02.292 3 3 False 87
Especificaciones de la maquina
Número de cuchillas 3
Ancho de bobina 10
Restablecer
Nuevo pedido
Altura
Cantidad piezas
Anchura
Permite rotación
Añadir
Nueva búsqueda e intervalo inicial
Creando generación 6 de 100
Aprovechamiento de la mejor solucion encontrada: 67,20690%
Cancelar
Pedido 21/06/2016 20:36:01.573 -> Altura: 1; Anchura: 6; Cantidad: 29
Pedido 21/06/2016 20:36:01.611 -> Altura: 9; Anchura: 1; Cantidad: 85
Pedido 21/06/2016 20:36:02.76 -> Altura: 4; Anchura: 1; Cantidad: 89
Pedido 21/06/2016 20:36:02.292 -> Altura: 3; Anchura: 3; Cantidad: 87
Nº Generaciones: 6
Nº Generación: 0; Altura: 789; Área útil: 49,40%; Nº Capas: 150
Nº Generación: 1; Altura: 723; Área útil: 53,91%; Nº Capas: 136
Nº Generación: 2; Altura: 680; Área útil: 57,32%; Nº Capas: 146
Nº Generación: 3; Altura: 668; Área útil: 58,35%; Nº Capas: 129
Nº Generación: 4; Altura: 582; Área útil: 66,98%; Nº Capas: 123
Nº Generación: 5; Altura: 582; Área útil: 66,98%; Nº Capas: 123
Mejor Solución -> Altura: 582; Anchura: 10; Área: 5820; Área útil: 3898 (66,98%); Nº Capas: 123
Capa -> Área: 70; Área útil: 70 (100,00%); Coordenadas: (0.0).(10.7)
Pieza -> Pedido: 21/06/2016 20:36:01.446; Altura: 7; Anchura: 5; Posición: (0.0)(5.7); Rotación: False
Pieza -> Pedido: 21/06/2016 20:36:01.446; Altura: 7; Anchura: 5; Posición: (5.0)(10.7); Rotación: False
Capa -> Área: 70; Área útil: 70 (100,00%); Coordenadas: (0.7).(10.14)
Pieza -> Pedido: 21/06/2016 20:36:01.446; Altura: 7; Anchura: 5; Posición: (0.7)(5.14); Rotación: False


TFG - Sistema para la optimización del corte en línea de float - Vicente Adriá Bohigues - 2016
Resultados
Búsqueda genética 21/06/2016 20:36:12 -> Nº Pedidos: 5; Nº Piezas: 342
Búsqueda genética 21/06/2016 20:38:14 -> Nº Pedidos: 5; Nº Piezas: 342
Nº Generación: 0; Altura: 771; Área útil: 50,56%; Nº Capas: 151
Nº Generación: 1; Altura: 742; Área útil: 52,53%; Nº Capas: 136
Nº Generación: 2; Altura: 636; Área útil: 61,29%; Nº Capas: 135
Nº Generación: 3; Altura: 584; Área útil: 66,75%; Nº Capas: 122
Nº Generación: 4; Altura: 582; Área útil: 66,98%; Nº Capas: 120
Nº Generación: 5; Altura: 580; Área útil: 67,21%; Nº Capas: 121
Nº Generación: 6; Altura: 580; Área útil: 67,21%; Nº Capas: 121
Nº Generación: 7; Altura: 580; Área útil: 67,21%; Nº Capas: 121
Nº Generación: 8; Altura: 580; Área útil: 67,21%; Nº Capas: 121
Nº Generación: 9; Altura: 578; Área útil: 67,44%; Nº Capas: 116
Nº Generación: 10; Altura: 578; Área útil: 67,44%; Nº Capas: 116
Nº Generación: 11; Altura: 578; Área útil: 67,44%; Nº Capas: 116
Nº Generación: 12; Altura: 578; Área útil: 67,44%; Nº Capas: 116
Nº Generación: 13; Altura: 578; Área útil: 67,44%; Nº Capas: 116
Nº Generación: 14; Altura: 578; Área útil: 67,44%; Nº Capas: 116
Nº Generación: 15; Altura: 578; Área útil: 67,44%; Nº Capas: 116
Nº Generación: 16; Altura: 578; Área útil: 67,44%; Nº Capas: 116
Nº Generación: 17; Altura: 578; Área útil: 67,44%; Nº Capas: 116
Nº Generación: 18; Altura: 578; Área útil: 67,44%; Nº Capas: 116
Nº Generación: 19; Altura: 578; Área útil: 67,44%; Nº Capas: 116
Nº Generación: 20; Altura: 578; Área útil: 67,44%; Nº Capas: 116
Nº Generación: 21; Altura: 578; Área útil: 67,44%; Nº Capas: 116
Nº Generación: 22; Altura: 578; Área útil: 67,44%; Nº Capas: 116
Nº Generación: 23; Altura: 578; Área útil: 67,44%; Nº Capas: 116
Nº Generación: 24; Altura: 578; Área útil: 67,44%; Nº Capas: 116
Nº Generación: 25; Altura: 578; Área útil: 67,44%; Nº Capas: 116
Nº Generación: 26; Altura: 558; Área útil: 69,86%; Nº Capas: 114
Nº Generación: 27; Altura: 558; Área útil: 69,86%; Nº Capas: 114
Nº Generación: 28; Altura: 558; Área útil: 69,86%; Nº Capas: 114
Nº Generación: 29; Altura: 558; Área útil: 69,86%; Nº Capas: 114
Nº Generación: 30; Altura: 558; Área útil: 69,86%; Nº Capas: 114
Nº Generación: 31; Altura: 558; Área útil: 69,86%; Nº Capas: 114
Nº Generación: 32; Altura: 558; Área útil: 69,86%; Nº Capas: 114
Nº Generación: 33; Altura: 558; Área útil: 69,86%; Nº Capas: 114
Nº Generación: 34; Altura: 558; Área útil: 69,86%; Nº Capas: 114
Nº Generación: 35; Altura: 558; Área útil: 69,86%; Nº Capas: 114
Nº Generación: 36; Altura: 558; Área útil: 69,86%; Nº Capas: 114
Nº Generación: 37; Altura: 558; Área útil: 69,86%; Nº Capas: 114
Controles
Nueva búsqueda
Sig. Generación
Sig. intervalo
Limpiar
Nueva busqueda
Confirmar
Cancelar
Detalles de la búsqueda
Parametros
Listado de pedidos
Eliminar pedido
Gen. aleatória
Área Min.
3898
Área Máx.
17750
Nombre
Altura
Anchura
Rotación
Cantidad
Color
21/06/2016 20:36:01.446 7 5 False 52
21/06/2016 20:36:01.573 1 6 False 29
21/06/2016 20:36:01.611 9 1 False 85
21/06/2016 20:36:02.76 4 1 False 89
21/06/2016 20:36:02.292 3 3 False 87
Especificaciones de la maquina
Número de cuchillas 3
Ancho de bobina 10
Restablecer
Nuevo pedido
Altura
Cantidad piezas
Anchura
Permite rotación
Añadir
Nueva búsqueda e intervalo inicial
Creando generación 46 de 100
Aprovechamiento de la mejor solucion encontrada: 69,85663%
Cancelar
Pedido 21/06/2016 20:36:01.573 -> Altura: 1; Anchura: 6; Cantidad: 29
Pedido 21/06/2016 20:36:01.611 -> Altura: 9; Anchura: 1; Cantidad: 85
Pedido 21/06/2016 20:36:02.76 -> Altura: 4; Anchura: 1; Cantidad: 89
Pedido 21/06/2016 20:36:02.292 -> Altura: 3; Anchura: 3; Cantidad: 87
Nº Generaciones: 6
Nº Generación: 0; Altura: 789; Área útil: 49,40%; Nº Capas: 150
Nº Generación: 1; Altura: 723; Área útil: 53,91%; Nº Capas: 136
Nº Generación: 2; Altura: 680; Área útil: 57,32%; Nº Capas: 146
Nº Generación: 3; Altura: 668; Área útil: 58,35%; Nº Capas: 129
Nº Generación: 4; Altura: 582; Área útil: 66,98%; Nº Capas: 123
Nº Generación: 5; Altura: 582; Área útil: 66,98%; Nº Capas: 123
Mejor Solución -> Altura: 582; Anchura: 10; Área: 5820; Área útil: 3898 (66,98%); Nº Capas: 123
Capa -> Área: 70; Área útil: 70 (100,00%); Coordenadas: (0.0).(10.7)
Pieza -> Pedido: 21/06/2016 20:36:01.446; Altura: 7; Anchura: 5; Posición: (0.0)(5.7); Rotación: False
Pieza -> Pedido: 21/06/2016 20:36:01.446; Altura: 7; Anchura: 5; Posición: (5.0)(10.7); Rotación: False
Capa -> Área: 70; Área útil: 70 (100,00%); Coordenadas: (0.7).(10.14)
Pieza -> Pedido: 21/06/2016 20:36:01.446; Altura: 7; Anchura: 5; Posición: (0.7)(5.14); Rotación: False

5.2.2. Añadir pedidos de manera manual

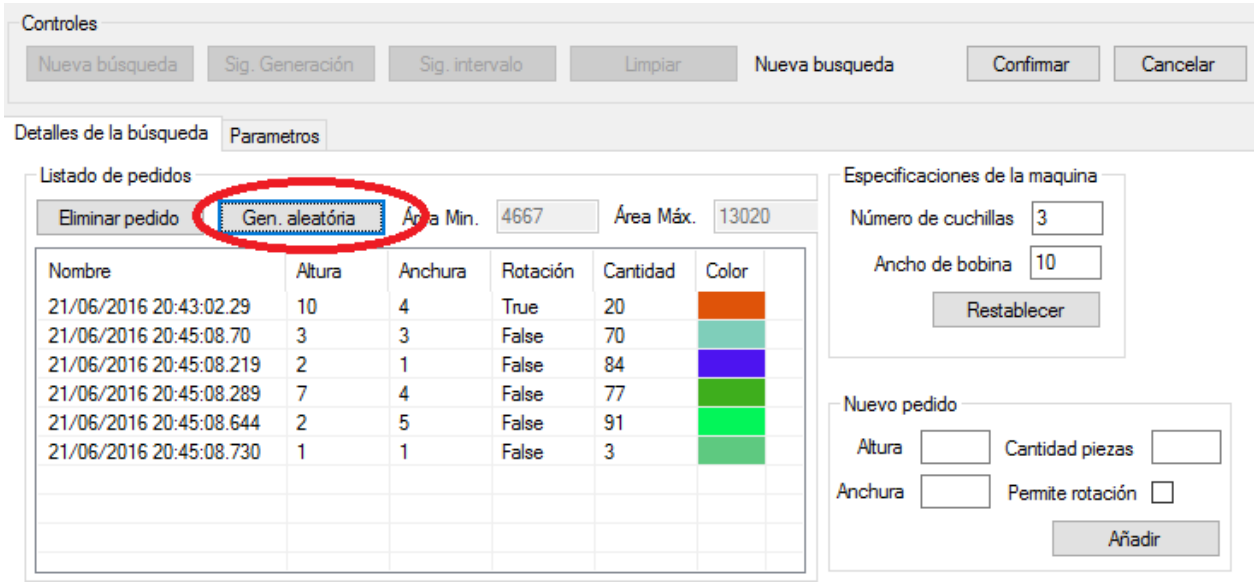


Mientras estemos en proceso de definir una nueva búsqueda, se podrán añadir tantos pedidos como se quiera, en este caso de forma manual. A continuación vemos el resultado de pulsar el botón añadir.



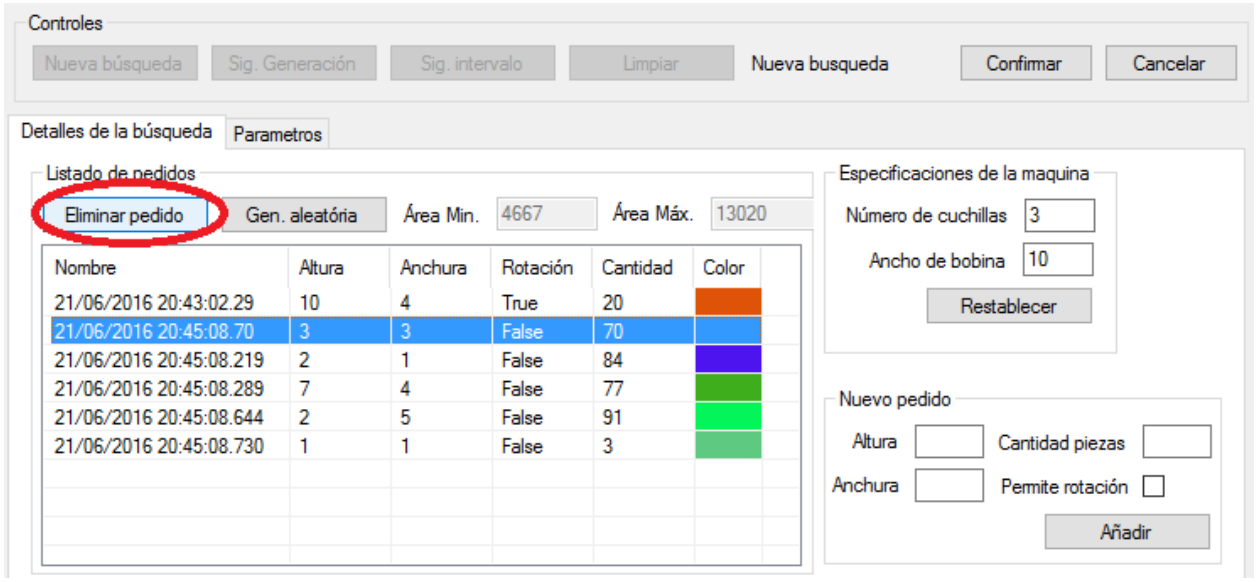
5.2.3. Añadir pedidos de manera aleatoria

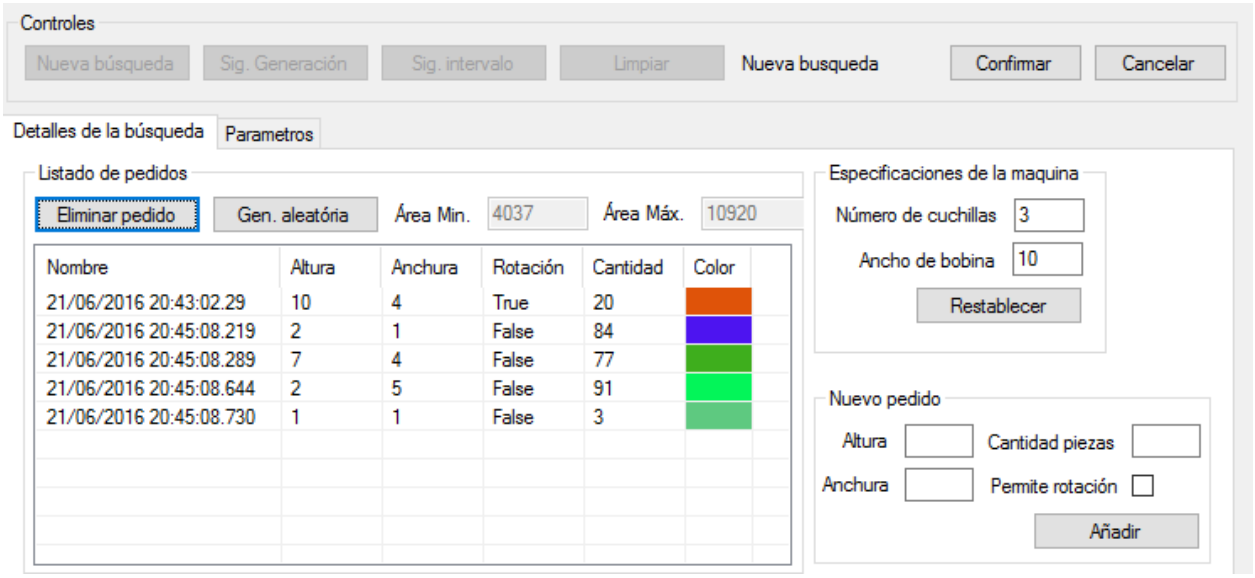
Si el lugar de añadir los pedidos de forma manual, se prefiere que se creen aleatoriamente en base a los parámetros especificados, basta con pulsar el botón de Generación aleatoria para que se carguen los nuevos pedidos directamente



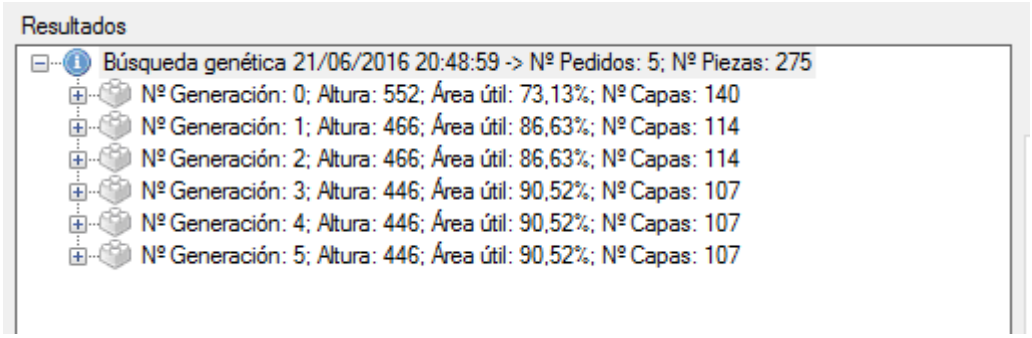
5.2.4. Eliminar pedidos

Si alguno de los pedidos de la lista no satisface nuestras necesidades, ya sea generado manual o aleatoriamente, basta con seleccionarlo y pulsar el botón Eliminar pedido para borrarlo de la lista.



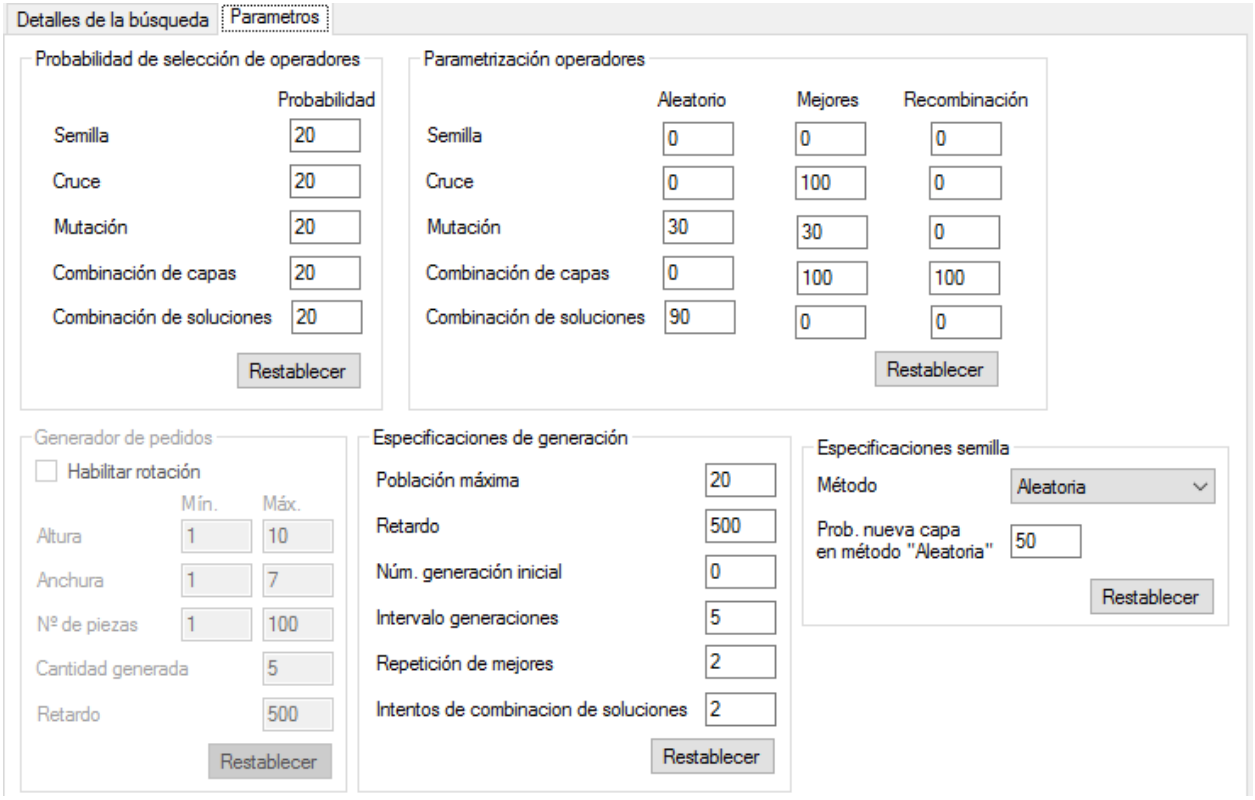


5.2.5. Nueva generación

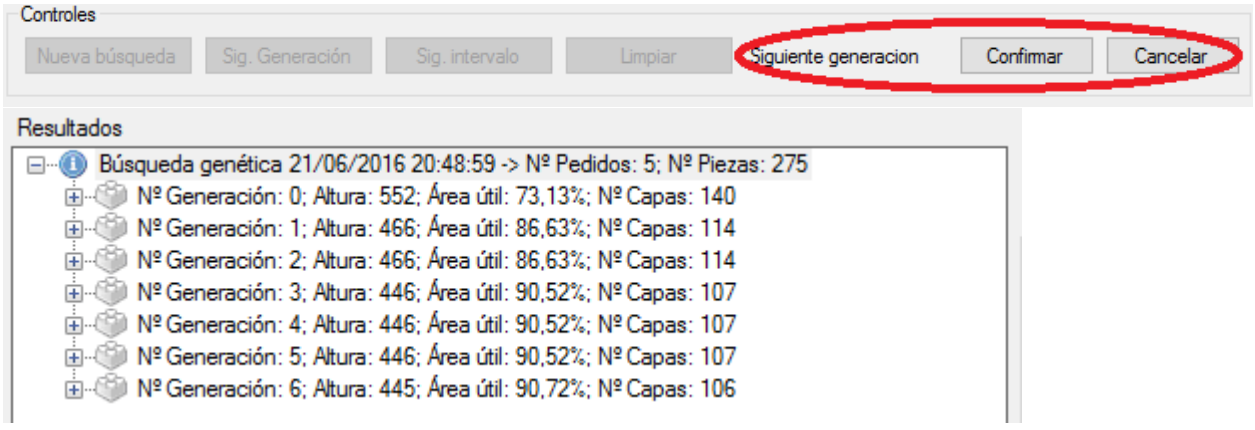


Si se quiere calcular una nueva generación de una búsqueda existente, se debe seleccionar la búsqueda en cuestión y pulsar el botón siguiente generación.

Tras pulsar el botón se habilitan los campos relativos a parámetros del algoritmo para que los modifiquemos si así lo queremos.

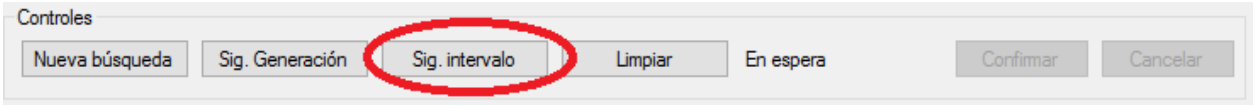


Si confirmamos la acción, veremos que un momento después aparecerá la nueva generación en el árbol de resultados. En el ejemplo, mejorando la solución de la generación anterior.

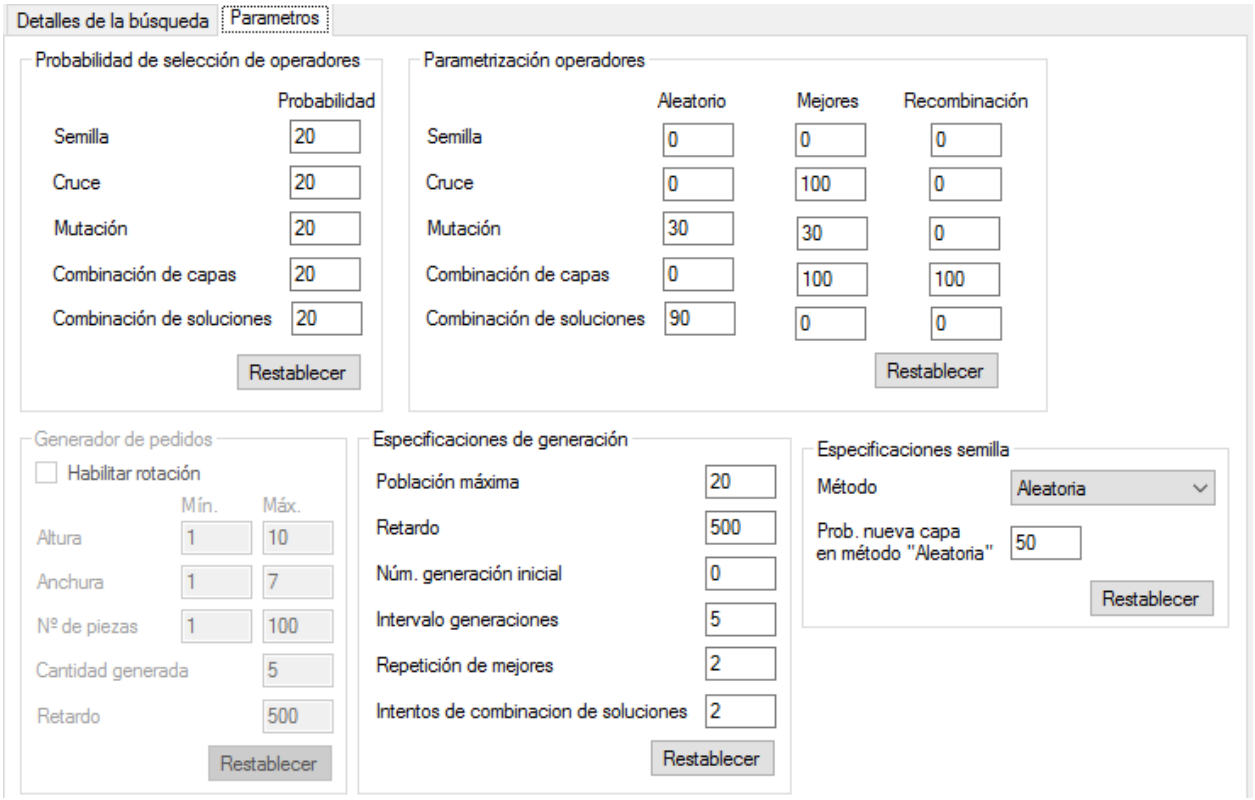


5.2.6. Nuevo intervalo de generaciones

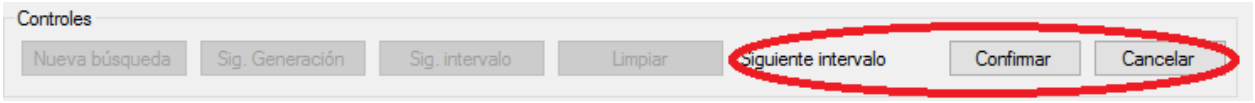
Si en lugar de una única generación, queremos calcular varias generaciones en el mismo proceso, tenemos la opción disponible a través del botón siguiente intervalo.

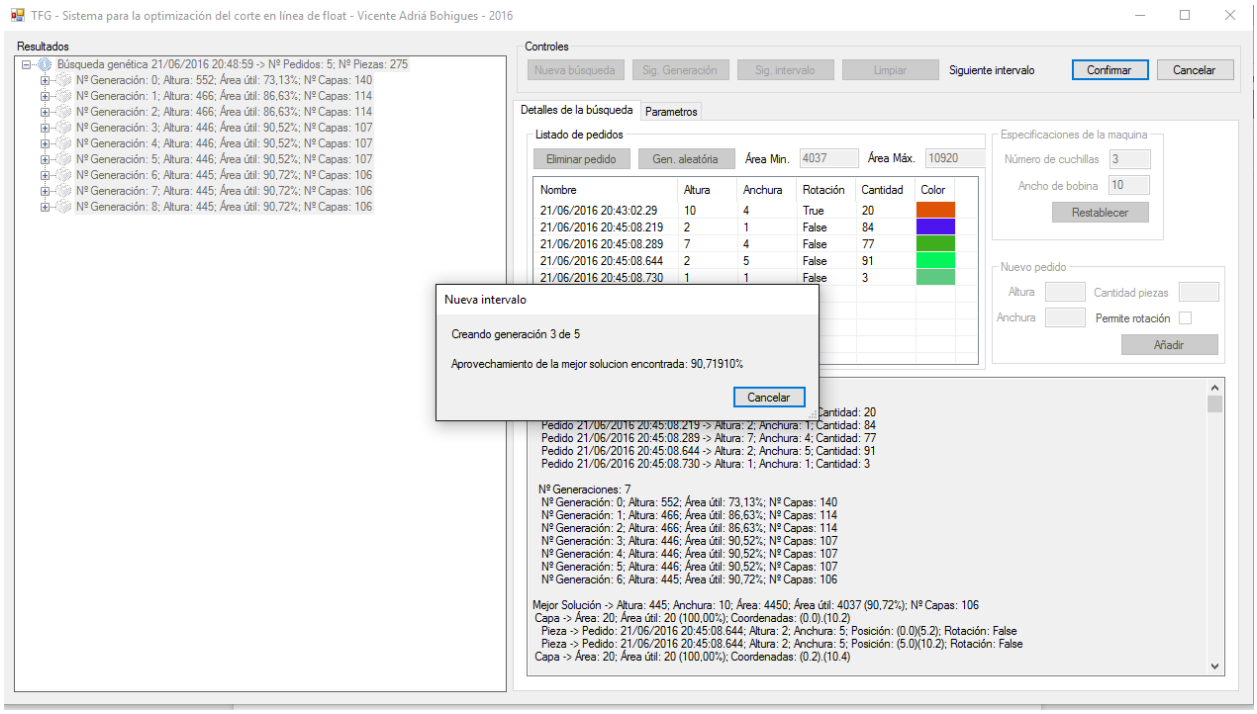


Tras pulsar el botón se habilitan los campos relativos a parámetros del algoritmo para que los modifiquemos si así lo queremos.

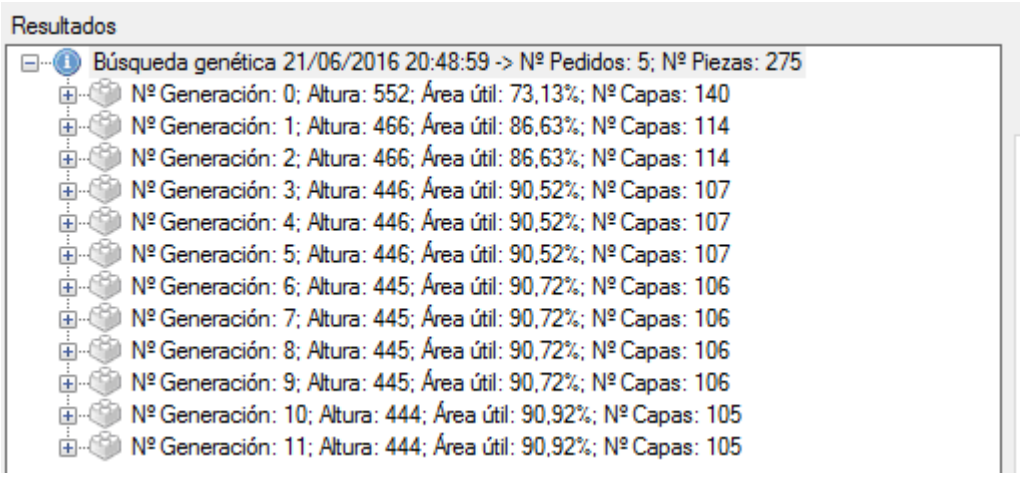


Si confirmamos la acción, aparecerá la ventana de progreso y se iran añadiendo las generaciones al árbol de resultados conforme se generen..

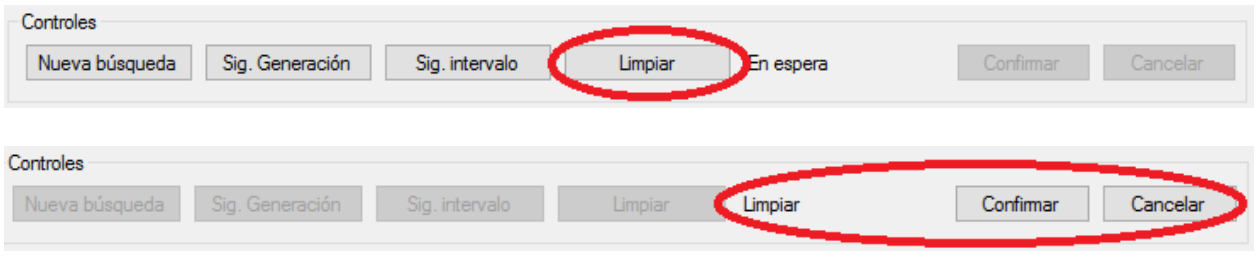




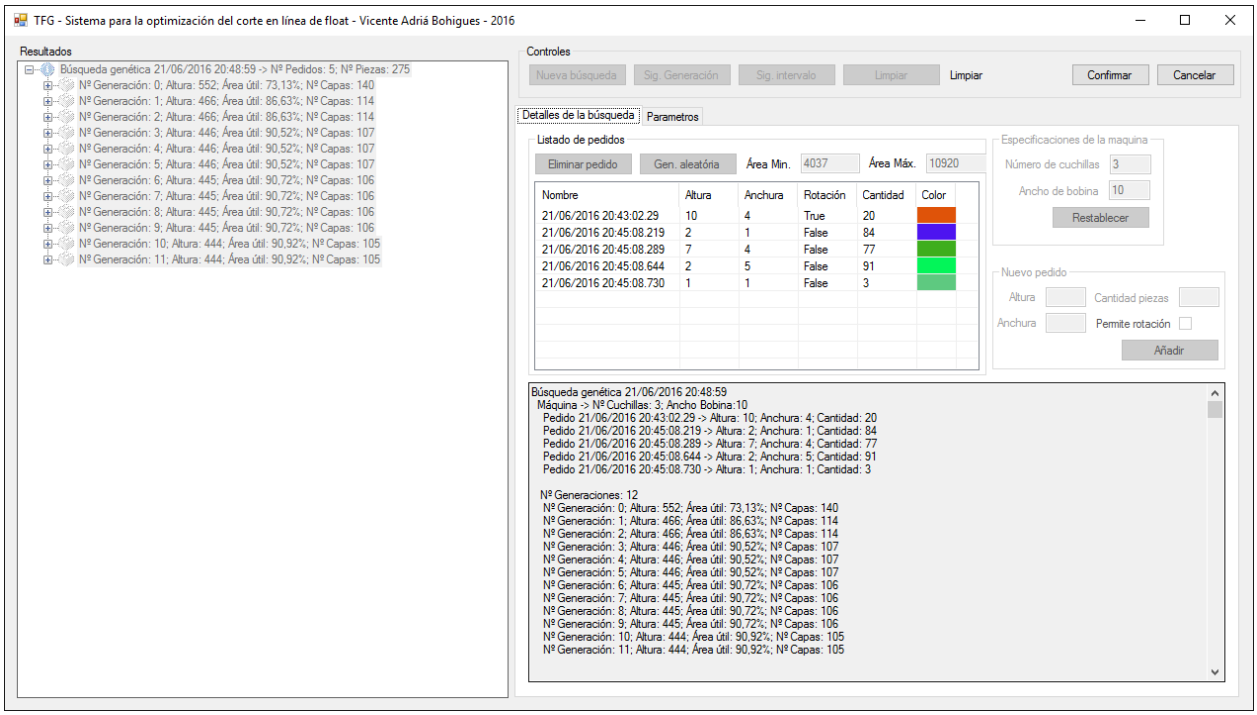
Cuando termine el proceso, en el árbol de resultados estarán todas las nuevas generaciones calculadas. En el caso del ejemplo, mejorando ligeramente la solucion.

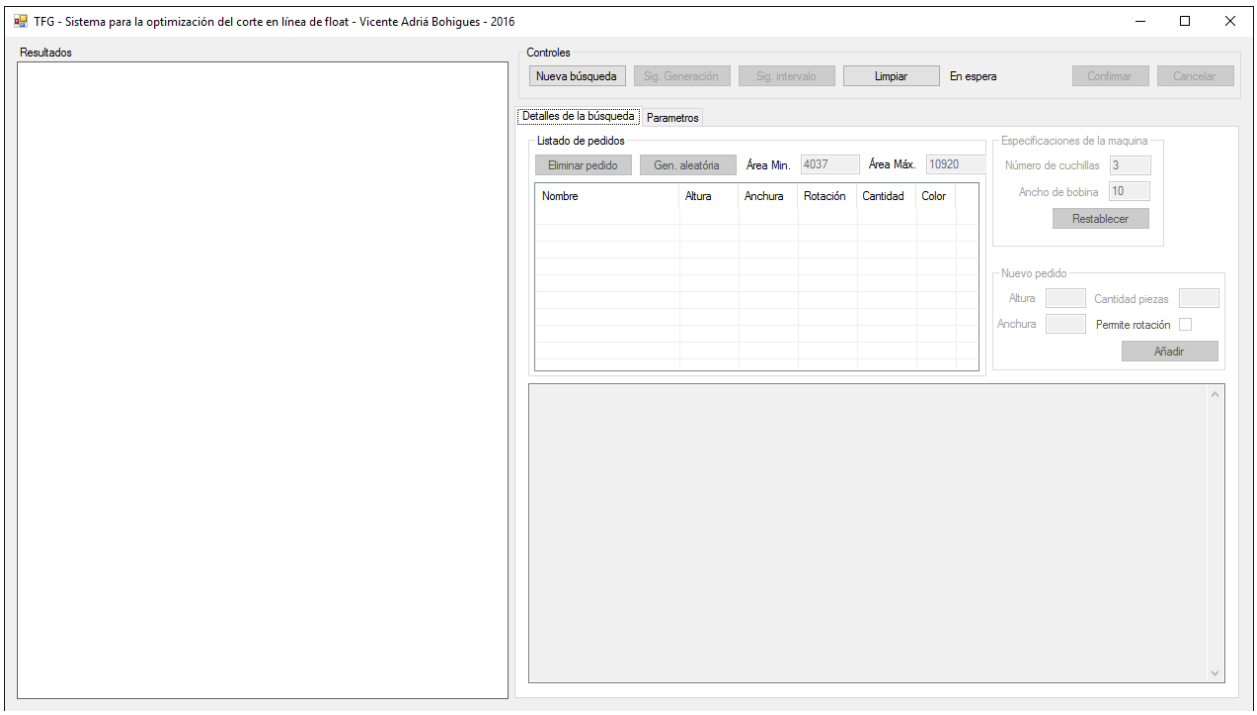


5.2.7. Limpieza de resultados anteriores



Si se pulsa en el botón limpiar y se confirma la acción, se borraran todos los datos generados durante la ejecución de la aplicación y se restablecerá los valores por defecto.



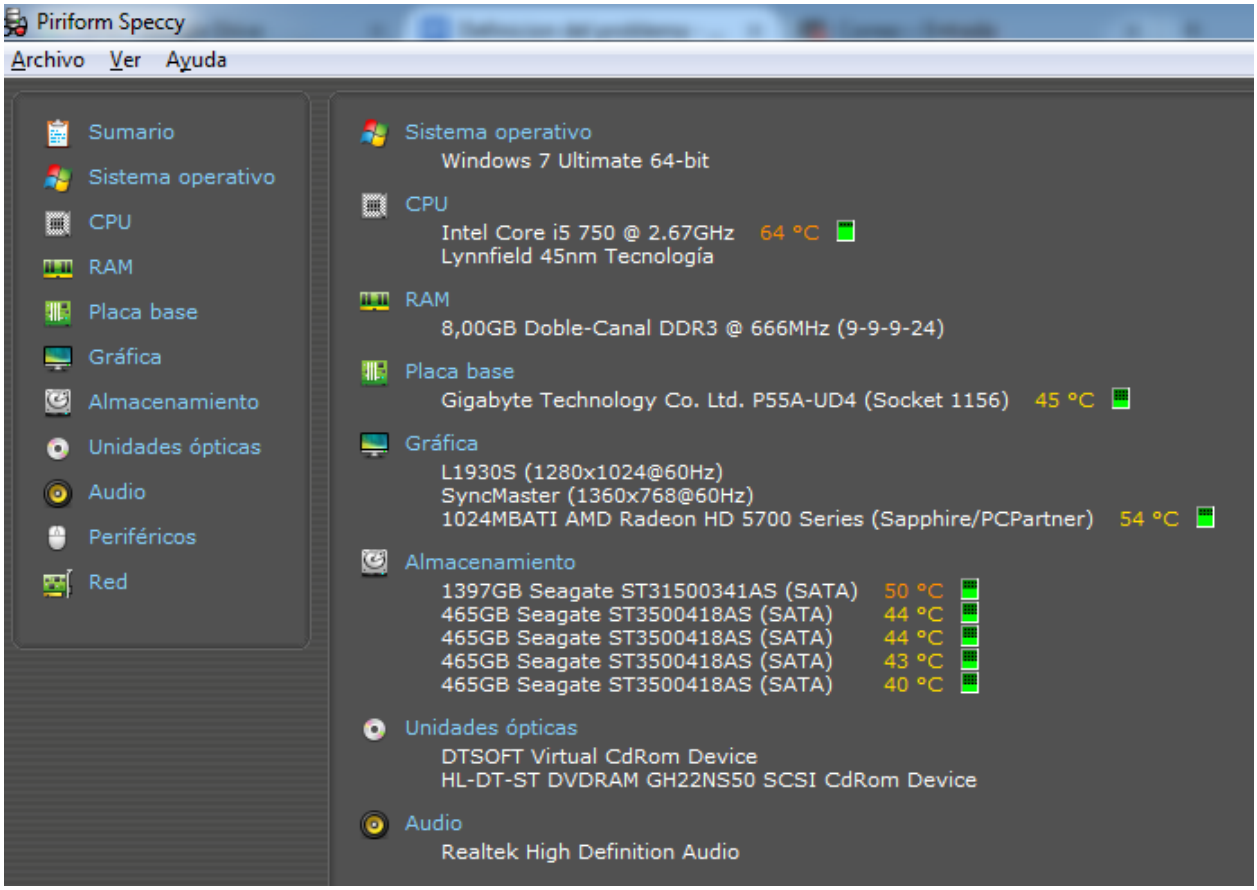


6. Pruebas realizadas

6.1. Entorno de pruebas

Las pruebas del algoritmo desarrollado se ejecutarán sobre un pc con las siguientes características (solo se nombran las más relevantes para nuestra aplicación):

Sistema operativo: Windows 7 Ultimate 64 bits
Procesador: Intel Core i5 750
Memoria Ram: 8 GB

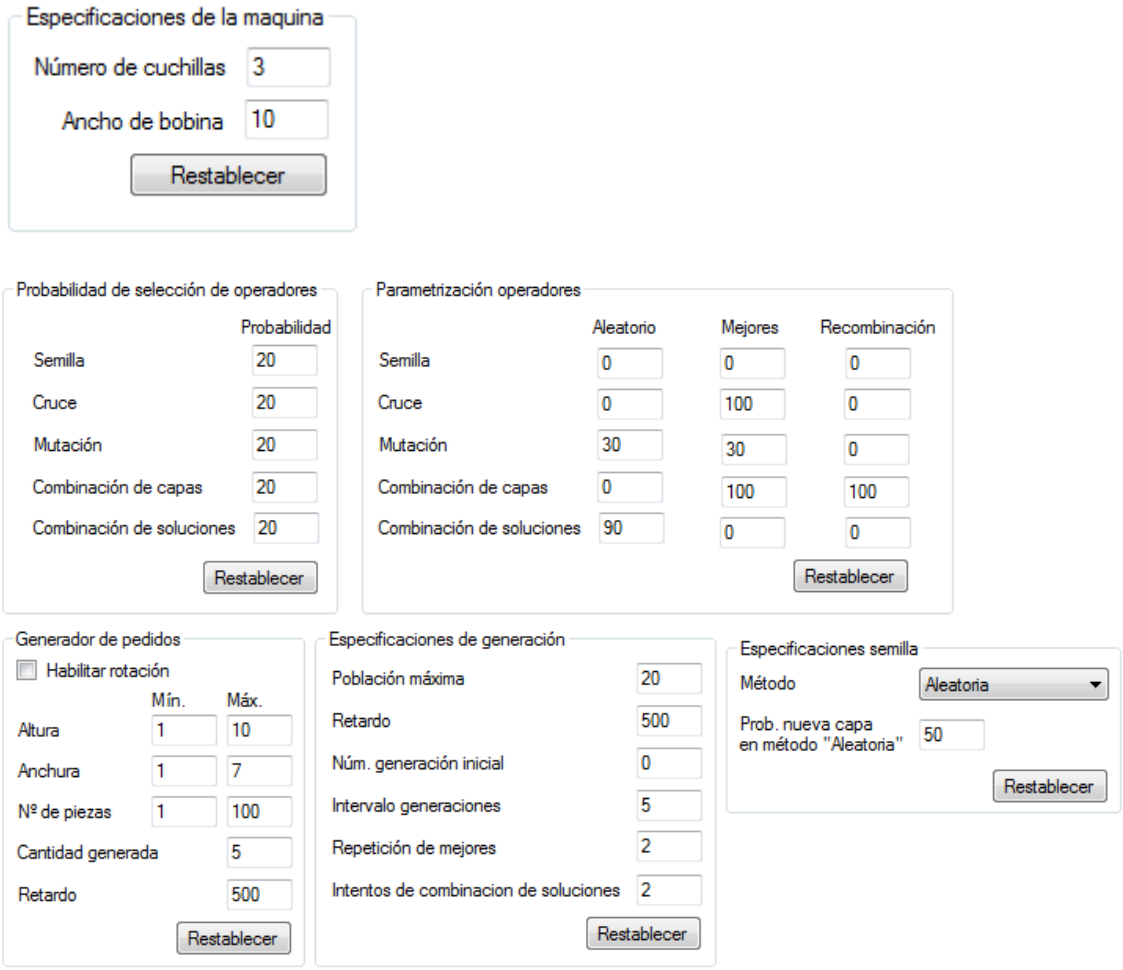


Características extraídas con el software Speccy de Piriform en su versión *free*

6.2. Modelado y ejecución de las pruebas.

En este punto vamos a evaluar el rendimiento del algoritmo adaptado. Para ello se van a definir un conjunto de problemas base con distintos nº de pedidos y cantidad de sus piezas. En cada una de las pruebas se variará en ellos el aspecto concreto que se pretenda evaluar.

Para todas las pruebas se definen los parámetros base como se ve en la imagen. En cada prueba se indicaran aquellos que se varían para adaptarlos a la prueba concreta.



6.2.1. Comparativa de los métodos de generación

En esta prueba se pretende comparar los distintos métodos que hay disponibles para el algoritmo semilla. Por un lado el método aleatorio, para el cual se puede indicar un porcentaje de oportunidad de generar una nueva capa y por otro, el método aleatorio compacto en el que siempre se intenta poner la pieza en una capa ya existente

y como último recurso generar una capa nueva. Este segundo método sería el equivalente a parametrizar al 0% el método aleatorio.

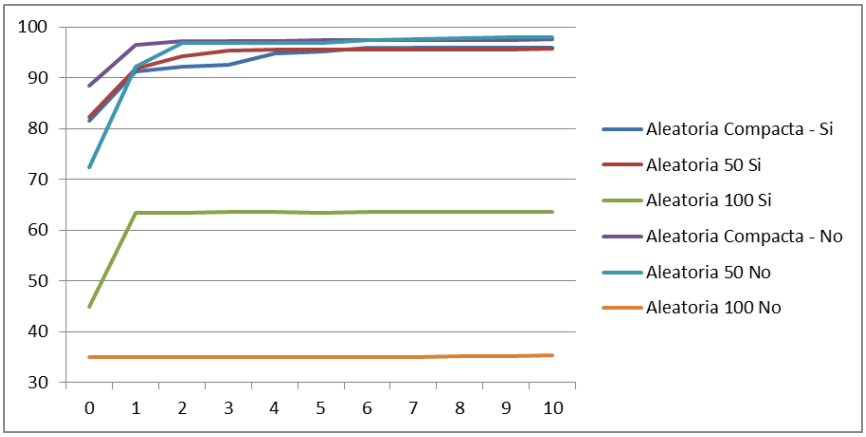
Para la prueba se han generado los pedidos y se han realizado 6 ejecuciones con las variantes que aparecen en las siguientes tablas:

Pedidos			
Altura	Anchura	Rotación	Cantidad
7	4	-	586
6	4	-	155
6	1	-	239
3	6	-	850
6	1	-	378
8	5	-	131
7	3	-	600
6	4	-	720
5	3	-	290
6	3	-	49

Ejecuciones			
Ejecución	Método	% Aleatorio	Rotación
1	Aleatoria Compacta	-	Si
2	Aleatoria	50	Si
3	Aleatoria	100	Si
4	Aleatoria Compacta	-	No
5	Aleatoria	50	No
6	Aleatoria	100	No

A continuación se presentan los resultados

Mejor solución (%)		
	Generación	
Ejecución	0	10
1	81,56	95,94
2	82,33	95,68
3	44,84	63,68
4	88,4	97,52
5	72,33	97,96
6	34,98	35,34



Se puede deducir de los resultados que cuanto más compacta se genere la semilla, mejor van a ser los resultados. Ya que en las ejecuciones con u 100% de probabilidad de nueva capa, la calidad de las soluciones no es capaz de remontar el mal comienzo.

En los casos de generación del 100% de capas nuevas, se ha observado que el algoritmo tiende a generar capas lo más anchas posibles, rotando las piezas para ello si tiene la opción disponible. Con esta configuración de capas, se dificulta la operación de añadir nuevas piezas a las capas existentes ya que siempre son anchas y cortas.

6.2.2. Variabilidad del algoritmo.

En este caso se va a comprobar, para una misma parametrización del algoritmo, la variabilidad de sus resultados. Al tratarse de un algoritmo genético es posible que las soluciones se estanquen en óptimos locales y no se continúe refinando la solución.

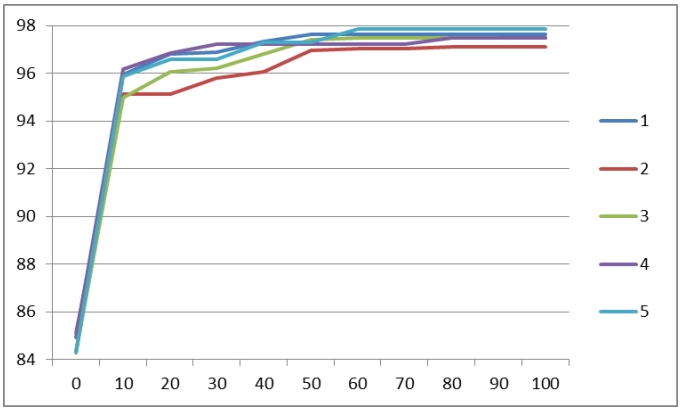
Para la prueba se han generado 20 pedidos con la configuración de la tabla siguiente y se ha ejecutado el algoritmo 5 veces con los parámetros estándar.

Pedidos			
Altura	Anchura	Rotación	Cantidad
5	3	S	13
6	3	S	17
4	6	S	58
1	5	S	30
3	6	N	96
3	2	N	14
5	6	S	51
8	1	N	23
8	3	N	22
6	3	N	97

Pedidos			
Altura	Anchura	Rotación	Cantidad
5	5	S	71
7	3	S	28
8	2	N	23
7	6	S	50
5	2	N	11
9	6	S	43
4	2	S	56
9	5	S	94
5	4	S	59
4	1	S	1

Se han obtenido los siguientes resultados.

Mejor solución (%)		
	Generación	
Ejecución	0	100
1	84,28	97,62
2	85,1	97,12
3	84,34	97,48
4	84,93	97,48
5	84,34	97,84



Como vemos, existe cierta variabilidad, pero los resultados tienden a converger a partir de generaciones avanzadas. En su aplicación a un caso real, bastaría con hacer varias ejecuciones y aumentar el número de generaciones para obtener resultados uniformes.

6.2.3. Comparativa de combinación de capas

En esta prueba, vamos a comprobar si el componente “combinación de capas” del algoritmo influye significativamente en la calidad de los resultados. Para ello se han creado tres conjuntos de pedidos y se ha ejecutado el algoritmo con dicho componente al 0% y al 100% en cada uno de los operadores

A continuación se muestran los pedidos y la configuración de las ejecuciones

Pedidos A			
Altura	Anchura	Rotación	Cant.
4	2	N	57
5	6	S	56
7	4	S	3
3	6	N	96
9	4	S	83
1	1	S	23
4	5	N	41
4	2	S	31
3	2	N	68
7	2	N	92
6	3	N	66
3	1	N	27
6	2	S	79
1	3	S	7
4	2	N	25
3	4	S	85
2	1	S	8
4	6	S	10
2	3	S	32
2	3	S	51

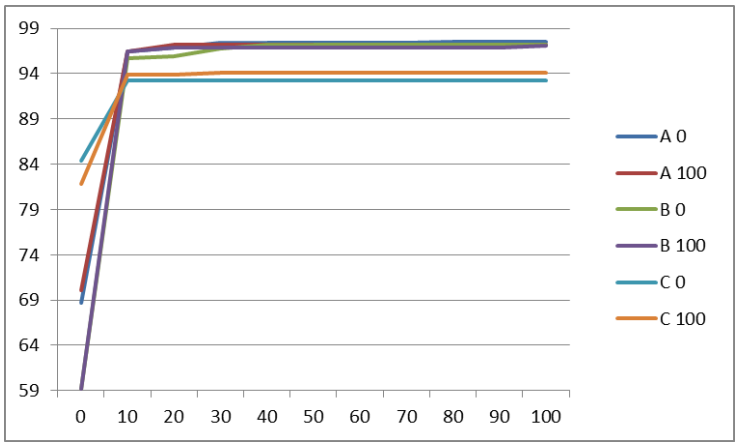
Pedidos B			
Altura	Anchura	Rotación	Cant.
9	2	N	84
7	2	N	11
7	2	S	34
6	1	S	38
6	1	N	74
6	3	S	17
1	1	S	23
8	6	S	31
9	4	S	50
8	6	N	13
1	5	S	49
5	2	N	63
9	2	N	90
2	2	S	1
6	6	S	33
3	1	S	64
8	1	S	25
1	3	S	94
2	1	N	89
7	3	S	78

Pedidos C			
Altura	Anchura	Rotación	Cant.
2	3	S	50
9	2	N	3
2	5	S	68
2	6	N	90
7	2	N	91
9	4	N	76
8	6	S	79
5	3	S	67
3	5	S	29
7	5	N	26
4	5	S	8
7	6	N	68
6	1	S	32
5	6	N	21
3	3	S	86
9	2	N	34
7	5	S	77
4	2	N	50
9	1	N	42
9	6	N	44

Ejecuciones		
Ejecución	Pedidos	Recombinación
1	A	0
2	A	100
3	B	0
4	B	100
5	C	0
6	C	100

A continuación presentamos los resultados.

Mejor solución (%)		
	Generación	
Ejecución	0	100
1	68,72	97,53
2	70,1	97,25
3	59,22	97,15
4	59,2	97,08
5	84,4	93,25
6	81,87	94,12



Se observa que no hay una diferencia significativa en los resultados entre aplicar este componente o no ya que la diferencia puede venir determinada por la propia variabilidad del algoritmo. La mejora/empeoramiento no es homogéneo en todos los casos, empeorando la solución en dos de ellos y mejorándola en el tercer conjunto de pedidos. Podemos determinar que no es un componente influyente.

6.2.4. Anulación de un operador

En este caso vamos a comprobar cuán importante es para la calidad de las soluciones el aporte que realiza cada operador. Para ello se ha preparado un conjunto de 20 pedidos y se ha ejecutado el algoritmo una vez por operador, más una de referencia con la parametrización estándar. En cada ejecución se ha puesto a 0 la probabilidad de que se seleccione un operador concreto.

A continuación se presentan los pedidos y la parametrización de cada ejecución.

Pedidos			
Altura	Anchura	Rotación	Cantidad
4	6	N	66
9	1	N	85
4	3	N	98
8	5	S	7
4	5	N	35
9	5	N	18
8	1	S	80
2	2	N	26
9	1	S	78
6	3	S	73

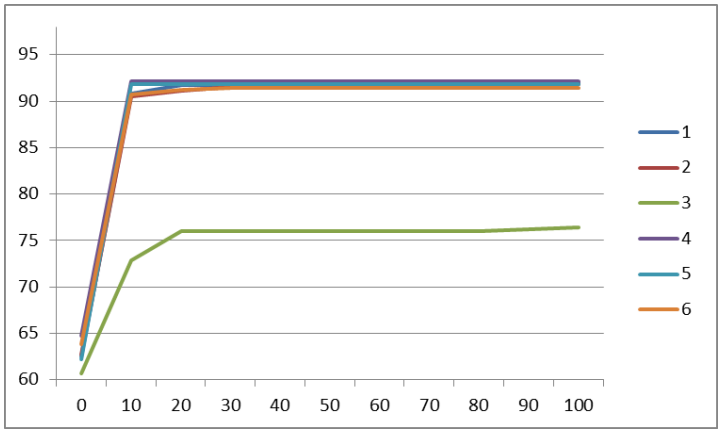
Ejecuciones	
Ejecución	Operadores
1	Referencia
2	Sin semilla
3	Sin cruce

3	2	N	19
6	3	S	71
6	2	N	99
2	1	N	89
5	5	N	27
6	2	N	49
3	1	N	39
5	5	N	76
5	1	N	1
4	4	N	42

4	Sin mutación
5	Sin combinación de capas
6	Sin combinación de soluciones

A continuación presentamos los resultados.

Mejor solución (%)		
	Generación	
Ejecución	0	100
1	62,51	91,98
2	62,68	91,86
3	60,64	76,4
4	64,7	92,09
5	62,25	91,86
6	63,86	91,41



Se puede observar en los resultados que el único operador realmente influyente en la calidad de las soluciones es el operador cruce, el encargado en realidad de seleccionar las mejores capas de dos soluciones y crear una nueva solución con lo mejor de las dos. En cuanto al resto de operadores, no parece tener un efecto real el hecho de que no se seleccionen para evolucionar las soluciones.

6.2.5. Evolución con un solo operador

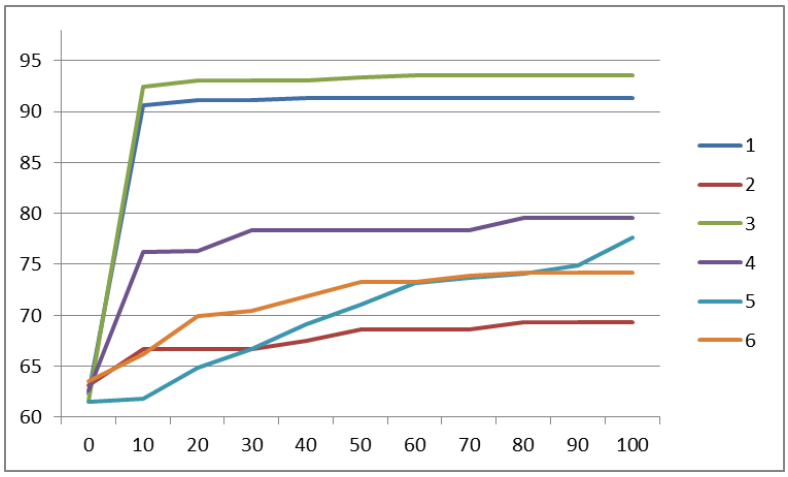
A diferencia de la prueba anterior, hemos querido comprobar que aporta en realidad cada operador por si mismo, para ello se han seleccionado los mismos pedidos que la prueba anterior, pero en este caso se han realizado ejecuciones con un único operador cada una más una ejecución de referencia con los parámetros por defecto.

Pedidos			
Altura	Anchura	Rotación	Cantidad
4	6	N	66
9	1	N	85
4	3	N	98
8	5	S	7
4	5	N	35
9	5	N	18
8	1	S	80
2	2	N	26
9	1	S	78
6	3	S	73
3	2	N	19
6	3	S	71
6	2	N	99
2	1	N	89
5	5	N	27
6	2	N	49
3	1	N	39
5	5	N	76
5	1	N	1
4	4	N	42

Ejecuciones	
Ejecución	Operadores
1	Referencia
2	Solo semilla
3	Solo cruce
4	Solo mutación
5	Solo combinación de capas
6	Solo combinación de soluciones

A continuación presentamos las soluciones.

Mejor solución (%)		
	Generación	
Ejecución	0	100
1	62,33	91,3
2	63,1	69,27
3	61,6	93,55
4	62,64	79,52
5	61,47	77,6
6	63,48	74,21



En este caso, se observa que las mejores ejecuciones son la de referencia, y superando a esta, la ejecución correspondiente al operador cruce. Se observa también que el operador cruce hace mejorar las soluciones en un intervalo muy pequeño de generaciones pero sin embargo se estanca en determinado punto, muy probablemente debido a la endogamia que genera este operador. Por otro lado, el resto de operadores aportan una mejora más lenta y a su vez introducen el componente aleatorio que va aportando información nueva en cada generación lo que permite salir de óptimos locales llegado el caso.

6.2.6. Cantidad de piezas vs. Nº de pedidos idénticos

Se ha pretendido en este caso, comprobar la coherencia del algoritmo al comparar las soluciones de varios conjuntos de pedidos. En los cuales, partiendo de 3 pedidos con muchas piezas, se han "partido" en varios pedidos idénticos pero con un número menor de piezas.

Pedidos A			
Altura	Anchura	Rotación	Cant.
3	3	N	941
7	2	N	845
3	6	N	216
4	4	N	968
4	2	N	990

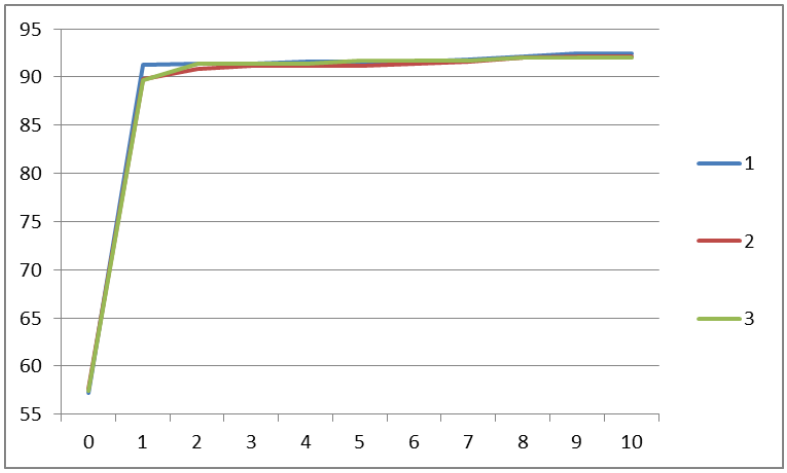
Pedidos B			
Altura	Anchura	Rotación	Cant.
3	3	N	471
3	3	N	470
7	2	N	423
7	2	N	422
3	6	N	108
3	6	N	108
4	4	N	484
4	4	N	484
4	2	N	495
4	2	N	495

Pedidos C			
Altura	Anchura	Rotación	Cant.
3	3	N	314
3	3	N	313
3	3	N	313
7	2	N	282
7	2	N	281
7	2	N	281
3	6	N	72
3	6	N	72
3	6	N	72
4	4	N	323
4	4	N	322
4	4	N	322
4	2	N	330
4	2	N	330
4	2	N	330

Ejecuciones	
Ejecución	Conjunto de pedidos
1	A
2	B
3	C

A continuación presentamos los resultados.

Mejor solución (%)		
	Generación	
Ejecución	0	10
1	57,23	92,44
2	57,64	92,08
3	57,41	92,07



Podemos comprobar que al algoritmo no le influye si hay muchas piezas de un pedido o muchos pedidos de una pieza. El algoritmo solo tiene en cuenta el numero total de piezas de cada tipo (alto*ancho)

6.2.7. Variabilidad en la generación inicial

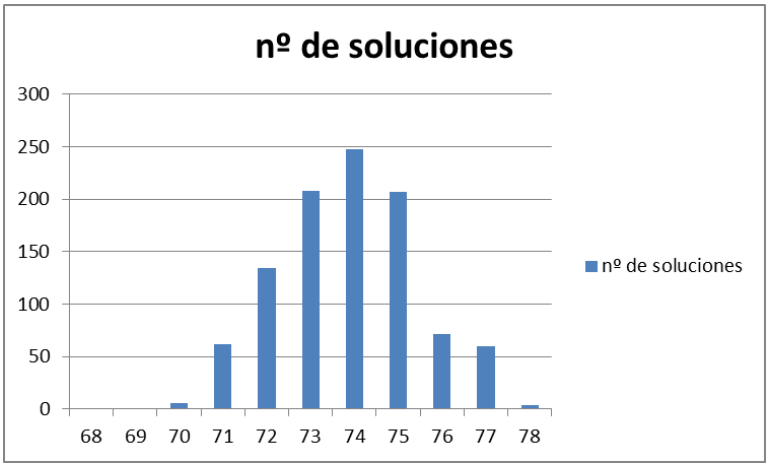
En este caso, se ha generado, para un conjunto de 25 pedidos una generación inicial con una población de 1000 soluciones generados por el algoritmo semillas. En esta prueba se pretende comprobar la distribución de la aleatoriedad relacionada con la calidad de las soluciones de la generación 0.

Pedidos			
Altura	Anchura	Rotación	Cantidad
9	5	N	26
8	5	N	56
5	3	N	37
6	4	N	41
7	4	N	2
4	3	N	6
7	1	N	61
8	6	N	56
7	4	N	83
5	6	N	44
3	5	N	29
9	1	N	85
9	6	N	98

Pedidos			
Altura	Anchura	Rotación	Cantidad
7	3	N	78
3	5	N	72
7	2	N	12
3	4	N	5
4	5	N	90
5	2	N	80
3	4	N	38
8	6	N	29
7	5	N	76
2	4	N	19
8	3	N	70
6	4	N	66

A continuación presentamos los resultados

% de aprovechamiento	Nº de soluciones
68	1
69	0
70	6
71	62
72	134
73	208
74	247
75	207
76	71
77	60
78	4
	1000



Según se observa, la distribución de las soluciones se comporta como una distribución probabilística normal. Son los resultados esperables cuando estamos basando el algoritmo semilla en decisiones aleatorias. En la mayoría de los casos se obtendrá una solución promedio, pero se producen casos aislados de muy buenas o muy malas soluciones respecto a la media.

6.2.8. Tamaño de población

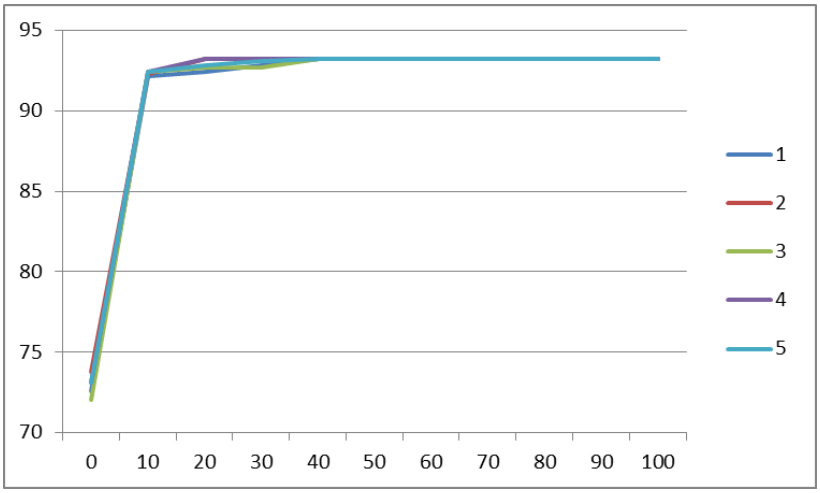
En esta punto se ha puesto a prueba si el tamaño de la población influye en la calidad de los resultados o en la rapidez de su obtención. Para ello se han generado 10 pedidos y se han realizado varias ejecuciones, incrementando en cada una de ellas el tamaño de la población.

Pedidos			
Altura	Anchura	Rotación	Cantidad
9	4	N	20
1	2	N	15
4	6	N	33
5	5	N	53
1	1	N	28
8	4	N	8
5	2	N	87
7	6	N	44
3	2	N	38
4	3	N	24

Ejecuciones	
Ejecución	Población
1	20
2	40
3	60
4	80
5	100

A continuación presentamos los resultados.

Mejor solución (%)		
	Generación	
Ejecución	0	100
1	72,56	93,21
2	73,82	93,21
3	72,07	93,21
4	73,14	93,21
5	73,14	93,21



En este caso, parece que a partir de un tamaño mínimo de población como pueden ser 20 individuos, no influye ni en la calidad ni en la rapidez de obtención de los resultados. Como se puede ver en el gráfico, todas las ejecuciones prácticamente se solapan entre sí.

6.2.9. Nº de intentos de recombinación

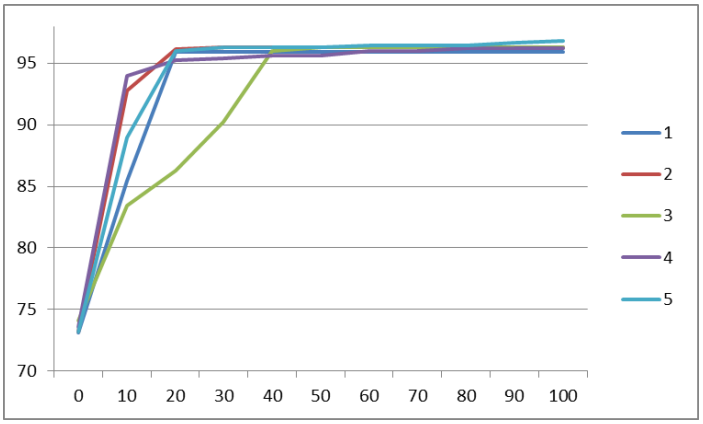
En este punto se ha puesto a prueba el número de intentos de recombinación que se ejecuta al final de cada generación. Para ello se han generado 20 pedidos y se ha parametrizado este factor, incrementándolo en cada caso.

Pedidos			
Altura	Anchura	Rotación	Cantidad
1	5	N	56
2	6	N	35
5	1	N	69
1	5	N	63
3	5	N	30
4	5	N	72
3	5	N	11
2	2	N	33
7	3	N	34
1	4	N	95
2	4	N	93
8	4	N	45
6	5	N	41
6	2	N	49
7	3	N	9
7	2	N	61
3	4	N	54
4	5	N	40
1	6	N	31
4	4	N	49

Ejecuciones	
Ejecución	Población
1	2
2	5
3	10
4	20
5	50

A continuación presentamos los resultados.

Mejor solución (%)		
	Generación	
Ejecución	0	100
1	73,09	95,91
2	73,22	96,29
3	74,12	96,29
4	73,58	96,22
5	73,22	96,83



Como vemos en los resultados, salvo un caso, que puede haberse producido debido a la variabilidad intrínseca del algoritmo, el nº de intentos de recombinación no es un factor influyente en la calidad de las soluciones. Además, a partir de generaciones avanzadas su efecto es nulo salvo la reducción de la distancia en puntos porcentuales entre la mejor y la peor solución de cada generación. Un exceso de intentos de recombinación favorece la endogamia y el estancamiento en óptimos locales.

6.2.10. Optimización de la parametrización

Para finalizar este apartado, en el último punto correspondiente a las pruebas hemos querido poner a examen las conclusiones obtenidas de los puntos anteriores. Para ello, se han generado 20 pedidos y se han realizado 2 ejecuciones con los parámetros estándar y 2 ejecuciones más con los parámetros modificados según nuestro criterio, para los cuales, se debería obtener una mejor solución.

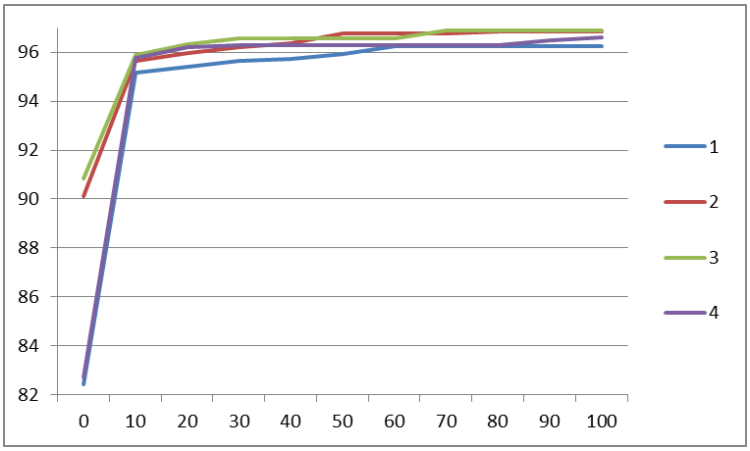
Pedidos			
Altura	Anchura	Rotación	Cantidad
7	5	N	27
2	4	N	68
1	1	S	72
3	1	N	39
5	5	N	76
3	4	N	61
8	3	S	65
9	5	S	76
8	5	S	7
9	6	S	85
6	4	S	9

Ejecuciones				
Ejecución	Operadores	Población	Intentos	Método
1	20,20,20,20,20	20	2	aleatoria
2	5,70,15,5,5	30	10	compacta
3	5,70,15,5,5	30	10	compacta
4	20,20,20,20,20	20	2	aleatoria

9	4	N	51
4	6	S	64
8	2	S	72
4	3	N	1
1	2	N	71
3	2	S	38
5	6	N	75
4	3	N	18
1	2	N	88

A continuación presentamos los resultados.

Mejor solución (%)		
	Generación	
Ejecución	0	100
1	82,45	96,24
2	90,11	96,84
3	90,85	96,88
4	82,7	96,62



Observamos en los resultados que, si bien no son diferencias significativas, siempre se ha obtenido una mejor solución con los parámetros “optimizados” que con los parámetros por defecto. Esto no hace sino poner de manifestó que una correcta parametrización del algoritmo puede aumentar la calidad de las soluciones obtenidas.

7. Posibles ampliaciones y conclusiones

7.1. Posibles ampliaciones

Durante la elaboración del TFG han ido surgiendo cuestiones que promueven el planteamiento de que pasaría si se tuviese en cuenta x característica. Por extensión, no caben en un TFG todas las posibles variantes, pero se considera convenientes dejarlas plasmadas por si en un futuro se desea retomar y ampliar el trabajo realizado.

Ampliación del modelo representado y adaptación del algoritmo a piezas base múltiples

Si atendemos a que cada capa puede representar un movimiento de alguna de las cuchillas, se podría ampliar la cantidad de piezas base que conforman la profundidad de la capa siempre que todas las piezas puedan alinearse en cuanto a anchura a las ya existentes en la capa (posible merging de capas). Este último punto complica la representación, pero considero que es adecuado para el caso ya que si tenemos varios pedidos con muchas repeticiones para cada pedido, es posible agruparlos para minimizar los desplazamientos y algunos cálculos.

Modificación a los datos de cada capa

1) Profundidad (largo del conjunto de piezas que conforman la base)
2) Matriz de piezas (representación de i,j piezas, donde i indica el número de pieza consecutiva y j indica el espacio entre cuchillas que ocupa)
   a) Se podría definir como un Array de Arrays por ser independientes las dos dimensiones ya que cada pieza en j puede repetirse un número diferente de i veces
   b) Si se establece la restricción de que en cada j sólo puede haber un tipo de pieza se puede sustituir el array por un multiplicador
   c) Si no se establece la restricción anterior, se puede añadir el multiplicador para simplificar el array.

Datos de la pieza

1) Posición de la pieza (minX, maxX, minY, maxY)
2) Largo de la pieza
3) Ancho de la pieza
4) Si consideramos la opción de múltiples piezas base, añadiríamos a la estructura
   a) Nº de repeticiones
   b) Posición de la pieza pasaría a representar el conjunto de piezas en lugar de una pieza individual

Número máximo de pedidos abiertos

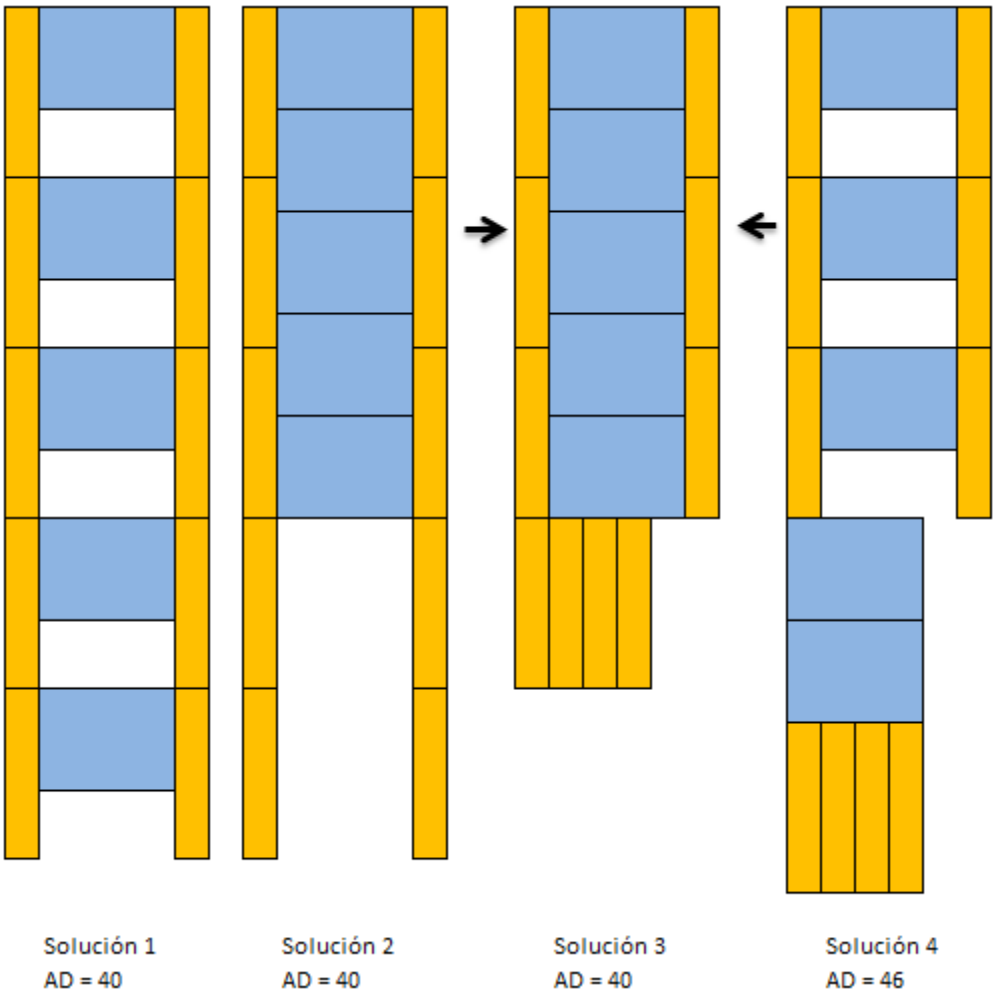
Normalmente las líneas de producción no suelen tener un espacio de almacenamiento grande al final de la línea. Tener muchos pedidos en marcha al mismo tiempo puede complicar las operaciones de manipulado del material. Por ello sería interesante comprobar cómo afectaría al algoritmo la adición a las restricciones planteadas con anterioridad de una nueva restricción relativa al número máximo de pedidos abiertos a la vez. En la industria siempre se pondera la relación coste de operación/pérdida de material, de manera que la solución más óptima en cuanto a material desperdiciado puede ser más costosa económicamente que otra con mayor desperdicio.

Gestión de calidades y defectos en la materia prima

Una de las opciones para adaptar el algoritmo a otros tipos de material que sean susceptibles a tener defectos en el propio material será añadir un registro de las impurezas o defectos en el material para realizar una gestión de calidades antes de ubicar finalmente una pieza sobre la materia prima. En realidad bastaría añadir a las condiciones donde se comprueba si se puede incluir una pieza o no en una capa, la comprobación de los defectos en el espacio que ocuparía la pieza, si se mantiene el criterio de calidad mínimo para dicho pedido.

Optimización post-solución

Junto con la ampliación del modelo para capas multi-pieza, se podría considerar la posibilidad de añadir un proceso de optimización post-solución sobre la mejor solución que nos haya devuelto el algoritmo genético, ya que se han observado ciertas configuraciones repetitivas en los que el algoritmo no acaba de afinar. En la figura siguiente se muestra un ejemplo de dichas configuraciones.



Una posible post-optimización sería la siguiente:

- Ordenar las capas según su aprovechamiento.
- Ordenar las piezas de cada capa en función de su tamaño
- Para todas las capas y su siguiente,
    - si comparten los anchos de piezas
    - juntar capas con n piezas principales (añadir la posibilidad de capas multipieza)
    - Si se han juntado capas, ver si se puede separar en algún punto donde haya corte limpio y decrece el aprovechamiento a partir de ese punto (ver ejemplo)
- Repetir desde el principio si se ha tenido algún éxito
- Intentar combinar todas las parejas de capas aunque no sean correlativas
- Si se ha tenido algún éxito, volver al principio

Este algoritmo puede tener un coste bastante elevado, pero recordemos que se ejecuta sobre una solución bastante optimizada, con lo que se espera que el número de reinicios no sea muy elevado.

Persistencia de entrada/salida

Habitualmente los programas con un estado de madurez elevado cuentan con algún medio para proporcionar persistencia de los datos utilizados en el programa, ya sea a través de bases de datos o en forma de ficheros. Por tratarse de una prueba de concepto y su escaso valor añadido en el ámbito que nos ocupa, no se ha implementado esta característica durante el desarrollo del TFG. Sin embargo una buena forma de ampliar sus capacidades, principalmente en cuanto a almacenamiento de las soluciones, sería la implementación del soporte a ficheros de entrada para poder tener los pedidos almacenados previamente y a ficheros de salida para poder volver a representar una búsqueda o solución en cualquier momento.

Dado que la información tratada en este problema es jerárquica y bien estructurada, tanto los pedidos como los resultados del proceso de búsqueda, un formato candidato serían los ficheros XML, estándares en el ámbito de intercambio de información y soportados en multitud de proyectos existentes.

Para la representación gráfica se podrían utilizar ficheros .SVG, una especificación para describir gráficos vectoriales bidimensionales en formato XML y directamente representables en multitud de aplicaciones de tratamiento de imagen.

7.2. Conclusiones

A lo largo de este TFG se han abordado dos casos reales en los que los algoritmos de optimización juegan un papel fundamental para el rendimiento económico de sus operaciones. Tanto en el corte de vidrio continuo como en el corte de bobinas de cartón para obtener laminas más pequeñas y manejables, una buena optimización supone un ahorro sustancial en costes de operación y materia prima desechada.

En este TFG se ha propuesto la idea de adaptar un algoritmo característico de los *strip packing problems* de Andreas Bortfeldt pensado para el llenado de contenedores, al problema aquí planteado en el que no hay límite de espacio pero si un costo de merma de material. Para ello se ha propuesto una notación ligeramente distinta, pensada principalmente para facilitar su representatividad y se han propuesto un conjunto de operadores deducidos del algoritmo original y que pensamos que aportarían calidad a la solución.

La solución se ha construido en base al lenguaje mejor conocido por el alumno en un entorno de desarrollo también familiar para él. Se ha diseñado según los parámetros de la programación orientada a objetos de manera que cada elemento representa un concepto real o un ítem material.

Tras las pruebas realizadas, hemos comprobado cuales de las propuestas han resultado mejores para el problema y que parámetros resultan irrelevantes para la obtención de una solución de calidad.

En general, creemos que es un buen algoritmo que puede adaptarse perfectamente a un entorno real. Se ha observado que en gran parte de las primeras capas se obtiene un aprovechamiento del 100%. En una situación real, permite que se vayan procesando las piezas de las primeras capas mientras que con toda probabilidad, antes de llegar a las capas “malas”, habrán llegado nuevos pedidos y se recalculará con las piezas restantes y los nuevos pedidos.

Se ha observado también que se obtienen resultados razonablemente buenos en un corto espacio de tiempo y numero de generaciones. El tiempo de respuesta es razonable en su contexto de aplicación, aunque es mejorable con una implementación revisada, y por ello se ha descartado realizar mediciones de tiempo en las pruebas ya que al tratarse de una primera versión del algoritmo, los tiempos medidos no serían representativos. Llevado al mundo real, esto no sería un problema ya que las planificaciones se hacen con suficiente antelación, y en todo caso, el tiempo de computo siempre es mucho menor que el de una planificación manual

Durante el desarrollo de este TFG también han surgido multitud de opciones y posibilidades que podrían mejorar tanto el algoritmo como el programa implementado pero que por motivos de extensión o no formar parte del problema principal a tratar, se han planteado como posibles aplicaciones.

Por lo tanto, para finalizar estas conclusiones, se podría decir que se ha propuesto una adaptación a un algoritmo existente que cumple suficientemente con las necesidades planteadas en la premisa del problema y que perfectamente se podría implantar en un entorno de producción real, aportando beneficios respecto a la situación anterior

8. Referencias

1. Andreas Bortfeldt - European Journal of Operational Research 172 (2006) 814–837 - A genetic algorithm for the two-dimensional strip packing problem with rectangular pieces.
2. Paul E. Sweeney and Elizabeth Ridenour Paternoster - The Journal of the Operational Research Society, Vol. 43, No. 7 (Jul., 1992), pp. 691-706 - Cutting and Packing Problems: A Categorized, Application-Orientated Research Bibliography
3. Shinji IMAHORI, Mutsunori YAGIURA and Hiroshi NAGAMOCHI - METR 2006–19 March 2006 - practical Algorithms for Two-dimensional Packing
4. María Cristina Riff, Xavier Bonnaire, Bertrand Neveu - Engineering Applications of Artificial Intelligence 2008 - A revision of recent approaches for two-dimensional strip-packing problems
5. Andrea Lodi, Silvano Martello, Michele Monaci - European Journal of Operational Research141 (2002) 241–252 - Two-dimensional packing problems: A survey
6. Miguel sanchez Garcia - Optimizacion combinatoria - http://www.sinewton.org/numeros/numeros/43-44/Articulo22.pd
7. Rafael Martí - Procedimientos Metaheur´ısticos en Optimizaci´on Combinatoria - http://www.uv.es/rmarti/paper/docs/heur1.pdf
8. Pilar Moreno Díaz, Gabriel Huecas Fernández-Toribio, Jesús Sánchez Allende, Almudena García Manso -TECNOLOGÍA y DESARROLLO, Revista de Ciencia, Tecnología y Medio Ambiente VOLUMEN V . AÑO 2007 - METAHEURÍSTICAS DE OPTIMIZACIÓN COMBINATORIA: USO DE SIMULATED ANNEALING PARA UN PROBLEMA DE CALENDARIZACIÓN - http://www.uax.es/publicacion/metaheuristicas-de-optimizacion-combinatoria-uso-de-simulated-annealing.pdf
9. Carlos P Gracia Calandín - Tesis Doctoral: Métodos y Algoritmos para resolver problemas de Corte unidimensional en entornos realistas. Aplicación a una empresa del Sector Siderúrgico.- https://riunet.upv.es/bitstream/handle/10251/7530/tesisUPV3250.pdf
10. Elena Mediavilla - Master de Computación. II MODELOS y HERRAMIENTAS. II.3 UML: Modelado estructura - http://www.ctr.unican.es/asignaturas/mc_oo/doc/m_estructural.pdf
11. Fernando Berzal - Diagramas de clases UML - http://elvex.ugr.es/decsai/java/pdf/3C-Relaciones.pdf
12. http://www.cristalyvidrio.com/vidrio-flotado

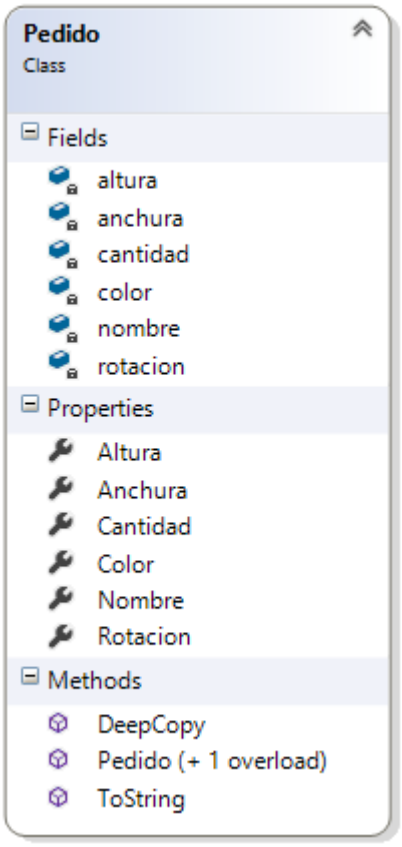
ANEXO I

Detalles de las clases implementadas

Debido a su extensión se detalla en este anexo la estructura de cada clase, tanto sus variables como funciones. Se ha omitido el código propiamente ya que esta disponible en los fuentes del proyecto y no aporta mucho valor en el formato de la memoria. Sin embargo, al igual que en el punto ***4.2.2 – Detalles de las clases implementadas***, para los métodos que tienen una lógica compleja, se explica en el propio método, a modo de comentario, el procedimiento que siguen.

Se ha decidido representar las clases de menor dependencia de otras a mayor, de manera que resulte más fácil la comprensión del modelo y cuando una clase contenga instancias o utilice funcionalidad de otra clase, esta última se haya explicado con anterioridad.

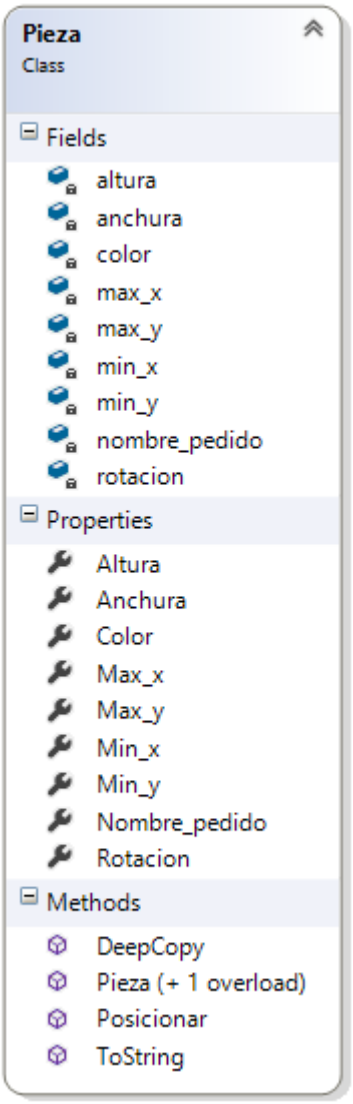
Clase Pedido:




```
public class Pedido

    {
        //Variables privadas
        private int altura;
        //Altura de las   piezas del pedido
        private int anchura;
        //Anchura de las piezas del pedido
        private int cantidad;
        //Cantidad de piezas del pedido
        private string nombre;
        //Nombre del pedido. Se genera
        automáticamente en el momento de
        creación
        private bool rotacion;
        //Indicará si se permite rotar las
        piezas del pedido o no
        private Color color;
        //Color con el que se representaran
        visualmente las piezas del pedido
        //Propiedades públicas que encapsulan
        las variables privadas {get; set}
        public int Altura {...}
        public int Anchura {...}
        public int Cantidad {...}
        public string Nombre {...}
        public bool Rotacion {...}
        public Color Color {...}
    //Este método realiza una copia exacta de la
    capa en otro espacio de memoria
        public Pedido DeepCopy() {...}
        //Constructor que crea un pedido con las especificaciones indicadas
        public Pedido(int Altura, int Anchura, int Cantidad, string Nombre,
bool Rotacion, Color color) {...}
```

```
        //Constructor que genera un pedido aleatorio cumpliendo las
        especificaciones indicadas
        public Pedido(int MinAltura, int MaxAltura, int MinAnchura, int
MaxAnchura, int MinCantidad, int MaxCantidad, bool Rotacion) {...}
        //Este método devuelve una representación en formato texto de la
        clase
        //Ej: "Pedido 21/06/2016 14:25:41.345 -> Altura: 2; Anchura: 4;
        Cantidad: 43"
        public override string ToString() {...}
    }
```

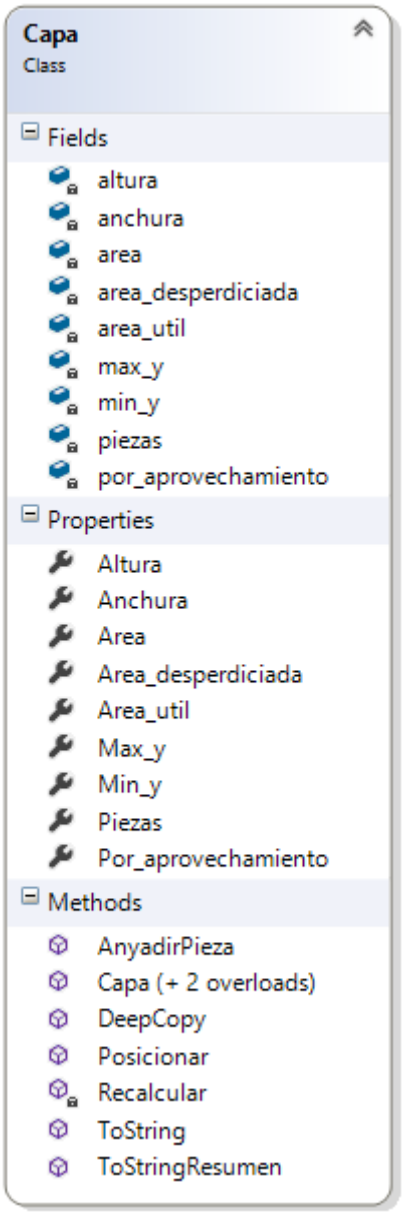
Clase Pieza:




```
public class Pieza
{
        //Variables privadas
        private int altura; //Altura de la pieza
        private int anchura;
        //Anchura de la pieza
        private string nombre_pedido;
        //Nombre del pedido al cual pertenece la
        pieza
        private int min_x;
        //Posición sobre el eje horizontal del
        vértice superior izquierdo
        private int max_x;
        //Posición sobre el eje
        horizontal del vértice inferior derecho
        private int min_y;
        //Posición sobre el eje vertical del
        vértice superior izquierdo
        private int max_y;
        //Posición sobre el eje vertical del
        vértice inferior derecho
        private bool rotación;
        //Variable que indica si la pieza esta
        rotada respecto a la especificación del
        pedido
        private Color color;
        //Color de la pieza cuando se represente
        visualmente especificada por el pedido

        //Propiedades públicas que encapsulan las
        variables privadas {get; set}
        public int Altura {...}
        public int Anchura {...}
        public string Nombre_pedido {...}
        public int Min_x {...}
        public int Max_x {...}
        public int Min_y {...}
        public int Max_y {...}
        public bool Rotacion {...}
        public Color Color {...}
    //Este método crea una copia exacta de la instancia pero en otro
    espacio de memoria
    public Pieza DeepCopy() {...}

    //Este método es el constructor de la clase, crea una pieza a partir
    de un pedido y asigna una rotación aleatoria si el pedido lo permite
    public Pieza(Pedido p) {...}
    //Se define un constructor privado que solo se usará para la copia
    profundidad, para ello recibe todas las variables de la clase
```

```
        private Pieza(int altura, int anchura, string nombre_pedido, int
min_x, int max_x, int min_y, int max_y, bool rotacion, Color color) {...}
        //Este método asigna la posición indicada a la pieza y recalcula el
        resto de variables de posición
        public void Posicionar(int x, int y) {...}
        //Este método devuelve una representación en formato texto de la
        clase
        //Ej: "Pieza -> Pedido: 21/06/2016 11:33:26.220; Altura: 6; Anchura:
        3; Posición: (0.0)(3.6); Rotación: False"
        public override string ToString() {...}

}
```

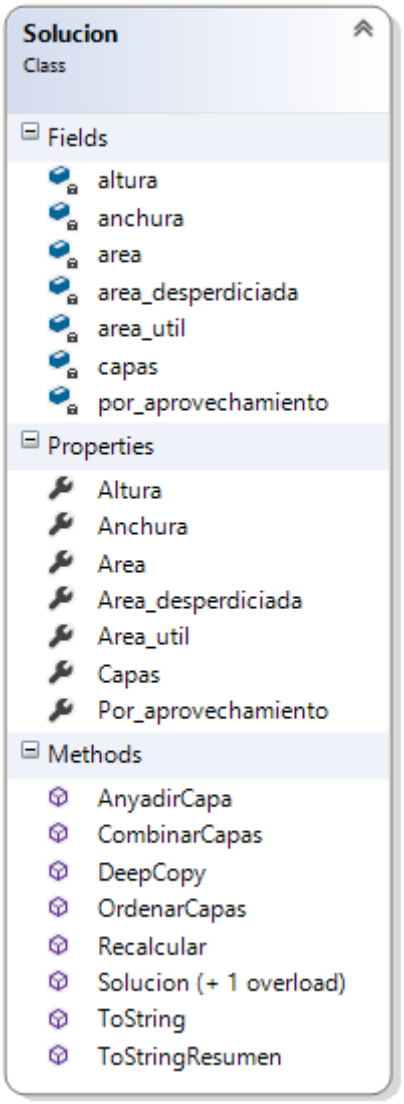
Clase Capa:




```
public class Capa
    {
        //Variables privadas
        private int anchura; //Anchura de la capa
        private int min_y; //Posición sobre el eje
        vertical del borde superior de la capa
        private int max_y; //Posición sobre el eje
        vertical del borde inferior de la capa
        private int altura; //Distancia entre los
        bordes superior e inferior
        private int area; //Área de superficie
        ocupada por la capa
        private int area_util; //Área de superficie
        ocupada por las piezas que contiene la capa
        private int area_desperdiciada; //Área de
        superficie no ocupada por las piezas que
        contiene la capa
        private double por_aprovechamiento;
        //Porcentaje de aprovechamiento de la
        superficie de la capa
        private ArrayList piezas; //Listado de las
        piezas que contiene la capa
        //Propiedades públicas que encapsulan las
        variables privadas {get; set}
        public int Min_y {...}
        public int Max_y {...}
        public int Altura {...}
        public int Area {...}
        public int Area_util {...}
        public int Area_desperdiciada {...}
        public double Por_aprovechamiento {...}
        public ArrayList Piezas {...}
        public int Anchura {...}
        //Este método añade una pieza a la capa si
        se puede teniendo en cuenta el nº de
        cuchillas de la maquina. Si no puede
        retorna "false"
        public bool AnyadirPieza(Pieza p, int numCuchillas)
        {
                ...
                //Comprobamos que la pieza es igual o menor en altura que la
                que inició la capa o es la primera
                ...
                //Comprobamos si cabe en algún sitio
```

```
            //Comprobamos si se puede añadir al final de alguna de las
            existentes (de la primera nunca se podrá)
            ...
                  //Si hay una pieza donde queramos colocar esta
                  ...
                   //Se comprueba que sea del mismo tipo y quepa a
                  continuación (vertical)
                   ...
            //Si no, comprobamos si se puede añadir al lado de la última
            (horizontal), si hay cuchillas libres
            ...
            //Si se puede añadir se añade
            ...
            //Y se devuelve el resultado de la operación
            ...
      }
      //Este método es el constructor de la capa, crea una capa a partir
      de una posición inicial y una anchura del material
      public Capa(int y, int anchura) {...}
      //Este método es una sobrecarga del constructor principal que crea
      una capa con una pieza inicial {...}
      //Este es un constructor privado que solo se usará para la copia
      profundidad
      private Capa(int anchura, int min_y, int max_y, int altura, int
area, int area_util, int area_desperdiciada, double por_aprovechamiento,
ArrayList piezas) {...}
      //Este método realiza una copia exacta de la capa en otro espacio de
      memoria
      public Capa DeepCopy(){...}
      //Este método cambia la posición de la capa al punto indicado y
      actualiza las variables relacionadas con la posición
      public void Posicionar(int y) {...}
      //Este método recalcula los indicadores que dependen del contenido
      de la capa
      private void Recalcular()
      {
            ...
            //Se recalcula el área útil en función de las piezas que
            contenga
            ...
            //Se recalcula la altura en función de las piezas que
            contiene
            ...
            //Se recalculan las áreas y el aprovechamiento
            ...
            }
      //Este método devuelve una representación en formato texto de la
      clase
      //Ej: "Capa -> Área: 10; Área útil: 10 (100,00%); Coordenadas:
      (0.0).(10.1)
      //      Pieza -> Pedido: 21/06/2016 12:46:11.661; Altura: 1;
      Anchura: 5; Posición: (0.0)(5.1); Rotación: False
      //      Pieza -> Pedido: 21/06/2016 12:46:11.661; Altura: 1;
      Anchura: 5; Posición: (5.0)(10.1); Rotación: False"
      public override string ToString() {...}
      //Este método devuelve una representación resumida en formato texto
      de la clase
      //Ej: "Capa -> Área: 10; Área útil: 10 (100,00%);  N° Piezas: 2"
      public string ToStringResumen() {...}
   }
```

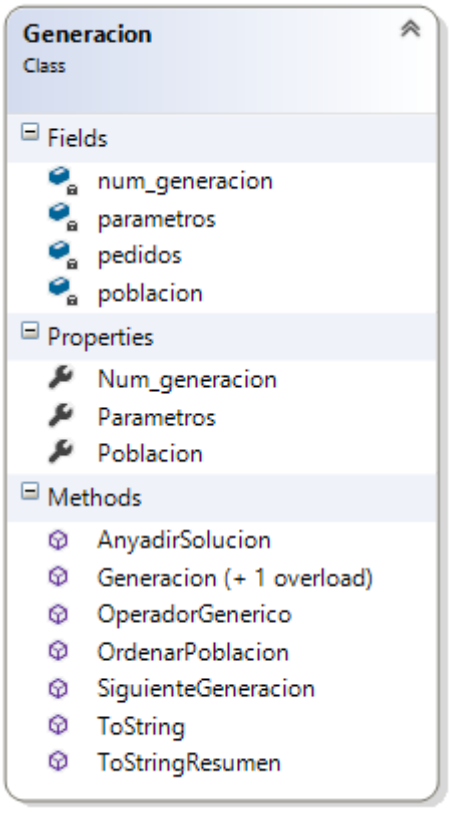
Clase Solucion:




```
public class Solucion

    {
        //Variables privadas
        private int anchura;
        //Anchura de la solución
        private int altura;
        //Altura total de la solución
        private int area;
        //Área de superficie ocupada por la
        solución
        private int area_util;
        //Área de superficie ocupada por las
        piezas que contiene la solución
        private int area_desperdiciada;
        //Área de superficie no ocupada por las
        piezas que contiene la solución
        private double por_aprovechamiento;
        //Porcentaje de aprovechamiento de la
        superficie de la solución
        private ArrayList capas;
        //Listado de las capas que contiene la
        solución
        //Propiedades públicas que encapsulan
        las variables privadas {get; set}
        public int Area {...}
        public int Area_util {...}
        public int Area_desperdiciada {...}
        public double Por_aprovechamiento {...}
        public ArrayList Capas {...}
        public int Anchura {...}
        public int Altura {...}
        //Este método añade una capa recibida
        por parámetro a la solución
        public void AnyadirCapa(Capa c) {...}
        //Este método intenta combinar las
        piezas de dos capas insertando las
        piezas de la segunda en la primera
        public bool CombinarCapas(int c1, int
c2, int numCuchillas) {...}
        //Este método realiza una copia exacta de la capa en otro espacio de
        memoria
        public Solucion DeepCopy(){...}
        //Este método ordena las capas de la solución de mayor
        aprovechamiento a menor
        public void OrdenarCapas(){...}
        //Este método recalcula los indicadores que dependen del contenido
        de la solución
        public void Recalcular() {...}
        //Constructor de la solución que crea una estructura vacía sin capas
        public Solucion(int anchura) {...}
        //Sobrecarga privada del constructor que crea una solución con las
        variables que recibe
        private Solucion(int anchura, int altura, int area, int area_util,
int area_desperdiciada, double por_aprovechamiento, ArrayList capas) {...}
        //Este método devuelve una representación en formato texto de la
        clase
        //Ej: Solución -> Altura: 484; Anchura: 10; Área: 4840; Área útil:
        3776 (78,02%); N° Capas: 83
        //      Capa -> Área: 60; Área útil: 60 (100,00%); Coordenadas:
        (0.0).(10.6)
```

```
        //        Pieza -> Pedido: 21/06/2016 13:26:52.262; Altura: 6;
        Anchura: 5; Posición: (0.0)(5.6); Rotación: False
        //        Pieza -> Pedido: 21/06/2016 13:26:52.262; Altura: 6;
        Anchura: 5; Posición: (5.0)(10.6); Rotación: False
        //      Capa -> Área: 60; Área útil: 60 (100,00%); Coordenadas:
        (0.6).(10.12)
        //       Pieza -> Pedido: 21/06/2016 13:26:52.262; Altura: 6;
        Anchura: 5; Posición: (0.6)(5.12); Rotación: False
        //        Pieza -> Pedido: 21/06/2016 13:26:52.262; Altura: 6;
        Anchura: 5; Posición: (5.6)(10.12); Rotación: False
        //      ...
        public override string ToString() {...}
        //Este método devuelve una representación resumida en formato texto
        de la clase
        //Ej: Solución -> Altura: 484; Anchura: 10; Área: 4840; Área útil:
        3776 (78,02%); N° Capas: 83
        public string ToStringResumen() {...}
    }
```

Clase Generacion:

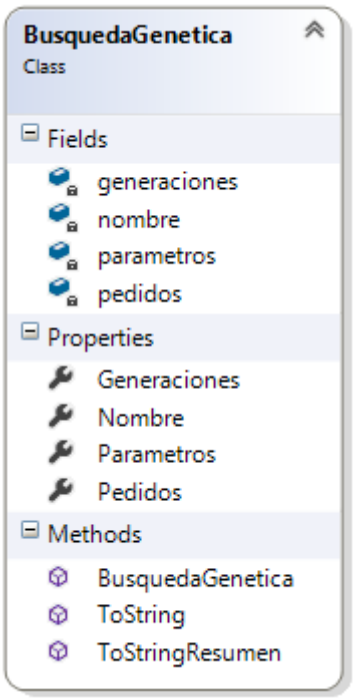



```
public class Generacion
    {
        //Variables privadas
        private int num_generacion;
        //Indica el número de generación en
        la que se encuentra la busqueda
        private ArrayList poblacion;
        //Listado de las soluciones que
        conforman la poblacion
        private ArrayList pedidos;
        //Listado de los pedidos del
        problema
        private Parametros parametros;
        //Parámetros con los que se realiza
        la generación

        //Propiedades públicas que
        encapsulan las variables privadas
        {get; set}
        ublic ArrayList Poblacion {...}
        public int Num_generacion {...}
        public Parametros Parametros {...}
        //Este método añade la solución que
        recibe a la población
        public void AnyadirSolucion(Solucion
s) {...}
        //Constructor que crea una
        generación con una población creada
        por la semilla en función de los
    parámetros y pedidos recibidos
    public Generacion(Parametros parametros, ArrayList pedidos) {...}
    //Utilizamos la sobrecarga con un parámetro más para indicar que no
    se tiene que crear una población inicial
    private Generacion(int num_generacion, Parametros parametros,
ArrayList pedidos, bool siguiente) {...}
    //Este método ejecuta el algoritmo definido para evolucionar las
    soluciones
    public Solucion OperadorGenerico(int por_aleatorio, int por_mejores,
int por_recombinacion, ArrayList soluciones, Parametros parametros)
```

```
 {
       ...
       //Se crea una copia de los pedidos para poder ir descontando
       cantidades
       ...
       //Se crea una copia de las soluciones para poder modificar las
       capas que contiene
       ...
       //Se calcula el número medio de capas para poder trabajar con
       los porcentajes
       ...
       //Se añaden las capas aleatorias de las soluciones fuente que
       aún no se hayan utilizado
       ...
             //Tanto si cabe como si no, se descarta la capa
             ...
             //Se descuentan las piezas utilizadas
             ...
             //Si no se pude añadir, se vuelven a contar sus piezas
             como pendientes
       ...
     //Se añaden las mejores capas de las soluciones fuente que aun
     no se hayan utilizado
     ...
             //Tanto si cabe como si no, se descarta la capa
             ...
             //Se descuentan las piezas utilizadas
              ...
             //Si no se pude añadir, se vuelven a contar sus piezas
             como pendientes
       ...
       //Se completa la solución hasta tener una solución completa
       ...
       //Se intenta recombinar dos capas al azar de la solución
       generada
       ...
      //Se devuelve la nueva solución
}
//Este método ordena la las soluciones que forman la población en
función de su aprovechamiento
public void OrdenarPoblacion() {...}
//Este método crea la siguiente generación en base a los parámetros
que recibe y las soluciones que contiene la actual generación
public Generacion SiguienteGeneracion(Parametros parametros)
{
       ...
       //Se copian las n primeras soluciones de la generación actual
       ...
       //Se ejecuta un operador de forma aleatoria hasta que se
       llegue al tamaño máximo de población
       ...
       //Se intenta la combinación de soluciones el numero indicado
       de veces
       ...
       //Se devuelve la nueva generación
       ...
 }
//Este método devuelve una representación en formato texto de la
clase
//Ej: "Nº Generación: 3
//        Solución -> Altura: 355; Área: 3550; Área útil: 3299
(92,93%); Nº Capas: 100
//        Solución -> Altura: 356; Área: 3560; Área útil: 3299
(92,67%); Nº Capas: 101
//        ..."
```

```
        public override string ToString() {...}
        //Este método devuelve una representación resumida en formato texto
        de la clase y su mejor solución
        //Ej: "N° Generación: 3; Altura: 484; Anchura: 10; Área: 4840; Área
        útil: 3776 (78,02%); N° Capas: 83
        public string ToStringResumen() {...}
    }
```

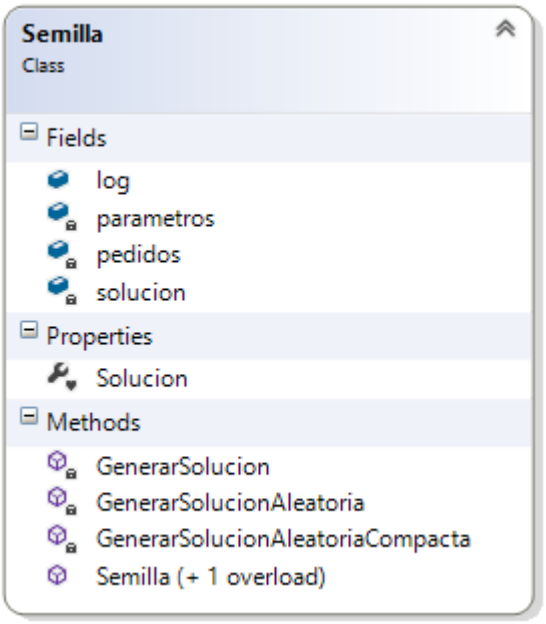
Clase BusquedaGenetica:




```
public class BusquedaGenetica

    {
        //Variables privadas
        private ArrayList pedidos;
        //Listado de pedidos del problema
        private Parametros parametros;
        //Parámetros especificados para el proceso
        de búsqueda
        private ArrayList generaciones;
        //Listado de generaciones creadas en el
        proceso de búsqueda
        private string nombre;
        //Nombre del proceso de búsqueda.
        Automáticamente se le asigna la fecha y
        hora
        //Propiedades públicas que encapsulan las
        variables privadas {get; set}
        public ArrayList Generaciones {...}
        public ArrayList Pedidos {...}
        public Parametros Parametros {...}
        public string Nombre {...}
        //Constructor de la clase que crea una
        nueva búsqueda con una generación con la
        población inicial
        public BusquedaGenetica(Parametros parametros, ArrayList pedidos)
{...}
        //Este método devuelve una representación en formato texto de la
clase
        //Ej: "Búsqueda genética 21/06/2016 14:25:42
        //          Máquina -> N° Cuchillas: 3; Ancho Bobina:10
        //              Pedido 21/06/2016 14:25:41.345 -> Altura: 2;
        Anchura: 4; Cantidad: 43
        //              Pedido 21/06/2016 14:25:41.351 -> Altura: 1;
        Anchura: 5; Cantidad: 12
        //              ...
        //
        //          N° Generaciones: 6
        //              N° Generación: 0; Altura: 254; Área útil: 85,20%; N°
        Capas: 82
        //              N° Generación: 1; Altura: 238; Área útil: 90,92%; N°
        Capas: 90
        //              ...
        //
        //          Mejor Solución -> Altura: 236; Anchura: 10; Área: 2360;
        Área útil: 2164 (91,69%); N° Capas: 89
        //              Capa -> Área: 20; Área útil: 20 (100,00%);
        Coordenadas: (0.0).(10.2)
        //                  Pieza -> Pedido: 21/06/2016 14:25:41.345;
        Altura: 2; Anchura: 4; Posición: (0.0)(4.2); Rotación: False
```

```
    //                  Pieza -> Pedido: 21/06/2016 14:25:41.399;
    Altura: 2; Anchura: 6; Posición: (4.0)(10.2); Rotación: False
    //              Capa -> Área: 20; Área útil: 20 (100,00%);
    Coordenadas: (0.2).(10.4)
    //              ..."
    public override string ToString(){...}
    //Este método devuelve una representación resumida en formato texto
    de la clase y su mejor solución
    //Ej: "Búsqueda genética 21/06/2016 14:25:42 -> Nº Pedidos: 5; Nº
    Piezas: 278"
    public string ToStringResumen(){...}
}
```

Clase Semilla:

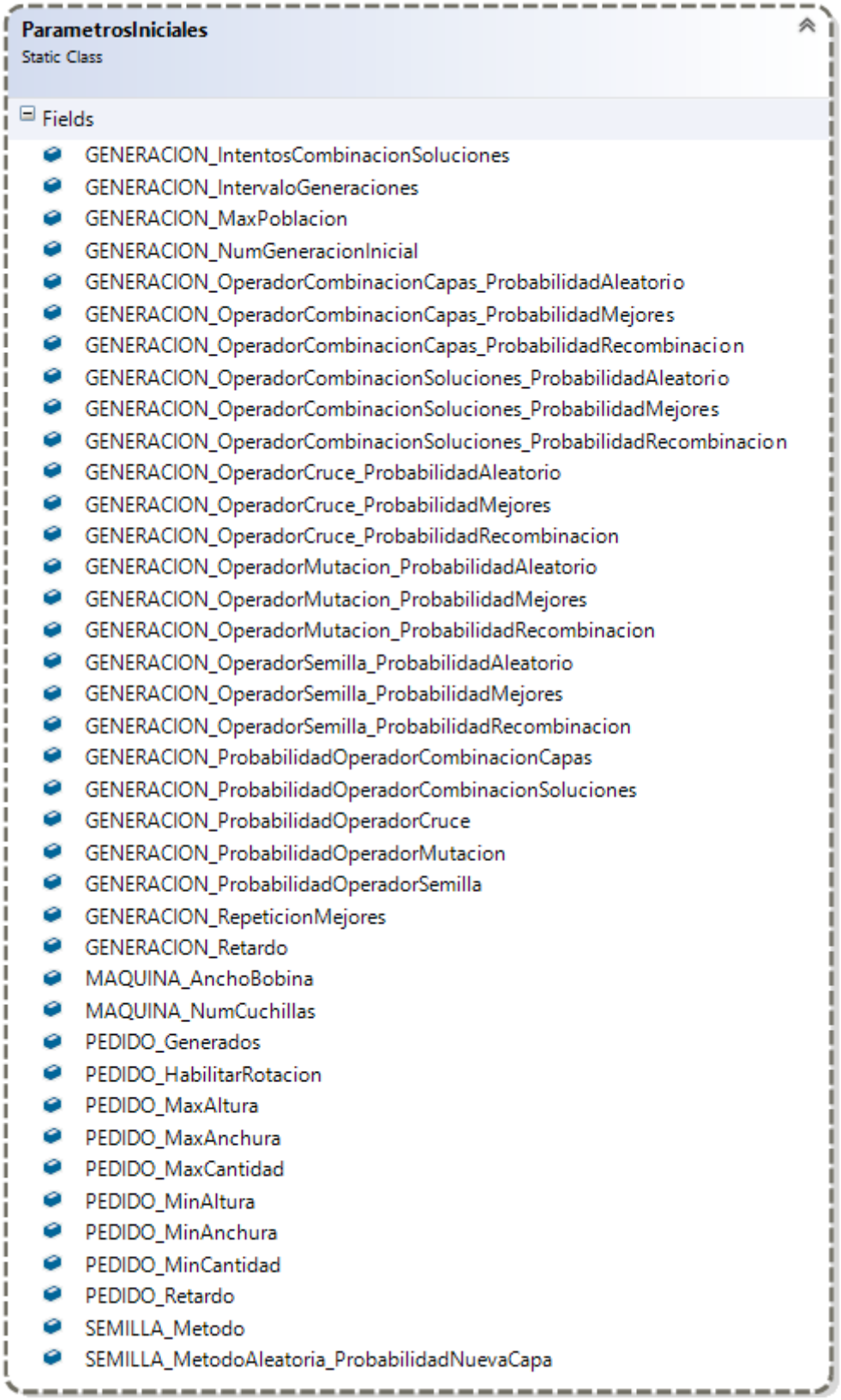



```
public class Semilla

    {
        //Variables privadas
        private ArrayList pedidos;
         //Listado de pedidos para
         generar la solución
        private Solucion solucion;
         //Solucion creada por la
         semilla
        private Parametros parametros;
         //Parametros utilizados en la
         creación de la solucion
         //Propiedades públicas que
         encapsulan las variables
         privadas {get; set}
       internal Solucion Solucion {...}
         //Método que llama al método
         elegido para generar la
         solución
        private void
GenerarSolucion(){...}
```

```
//Método que genera la solución mediante el método "Aleatoria"
private void GenerarSolucionAleatoria()
{
      ...
      //Elegir una pieza al azar
      ...
      //Se elige al azar si se crea una capa nueva o se intenta
      incorporar a una ya existente
      //Por estadística, las capas que se crean primero, tienen más
      probabilidad de llenarse que las últimas (han estado más veces
      en el bombo)
      ...
      //Si hay capas en las que quepa, se selecciona una al azar y
      se incorpora la pieza
            ...
            //Si no cabe, se descarta la capa y se prueba con la
            siguiente
            ...
            //Si no cabe en ninguna capa o se decide crear una
            nueva directamente, se crea una capa nueva con la nueva
            pieza
            ...
```

```
                    //Se resta 1 a las piezas pendientes de incorporar del
                    pedido correspondiente
                ...
                }
        }
        //Método que genera la solución mediante el método "Aleatoria
        Compacta". En este método solo se creará una nueva capa si no cabe
        en ninguna de las existentes
        private void GenerarSolucionAleatoriaCompacta()
        {
               ...
               //Elegir una pieza al azar
               ...
               //Si hay capas en las que quepa, se selecciona una al azar y
               se incorpora la pieza
               ...
               //Si no cabe, se descarta la capa y se prueba con la siguiente
               ...
               //Si no cabe en ninguna capa, se crea una capa nueva con la
               nueva pieza
               ...
               //Se resta 1 a las piezas pendientes de incorporar del pedido
               correspondiente
               ...
             }
        }
        //Constructor que genera una solución desde cero en base a unos
        parámetros y unos pedidos
        public Semilla(ArrayList pedidos, Parametros parametros) {...}
        //Constructor que completa una solución en base a unos parámetros y
        unos pedidos
        public Semilla(ArrayList pedidos, Parametros parametros, Solucion s)
{...}
    }
```

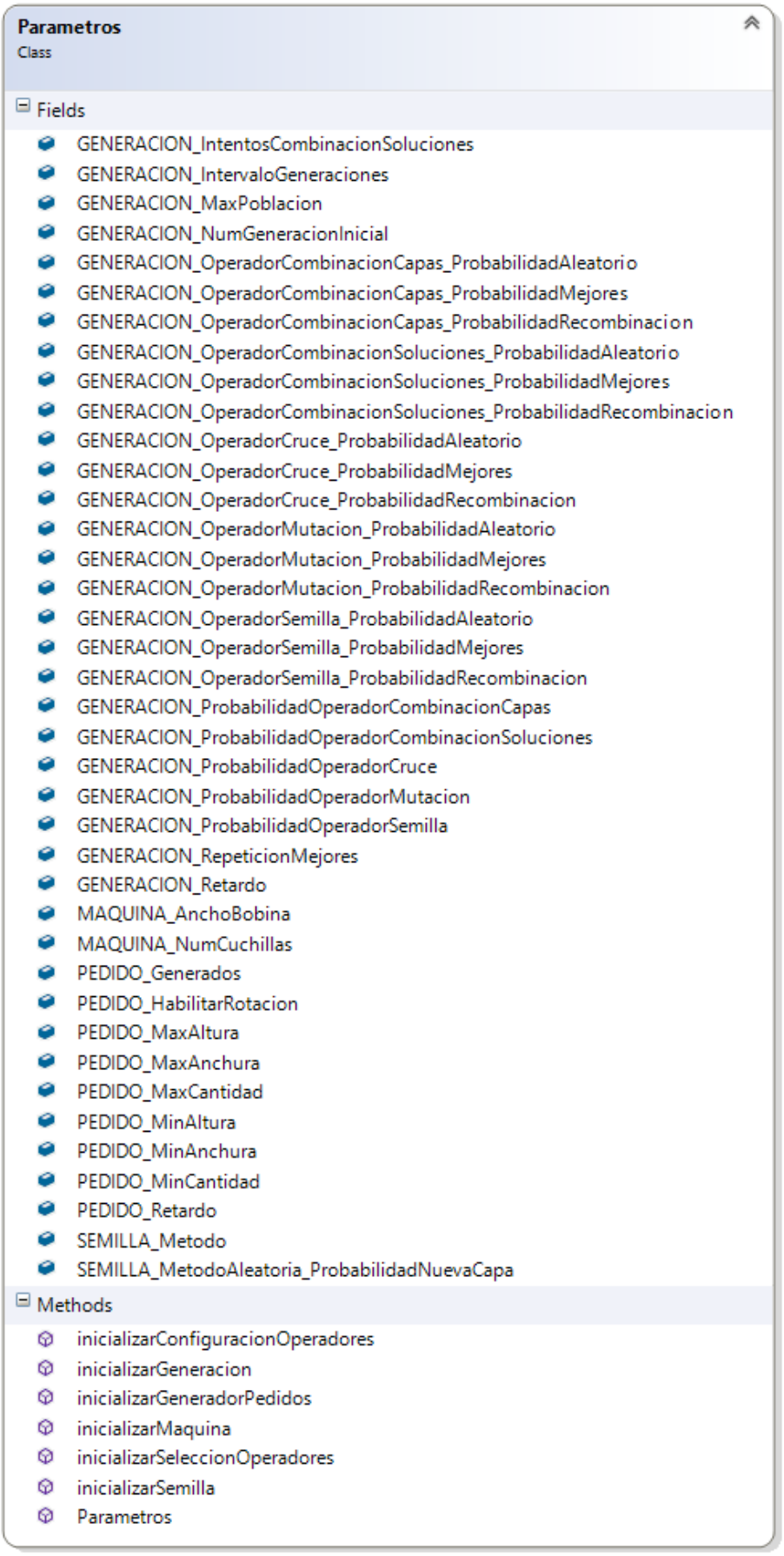
Clase ParametrosIniciales:



```
public static class ParametrosIniciales
    {
        //Parámetros Iniciales de creacion de una maquina
        public static int MAQUINA_NumCuchillas = 3;
        public static int MAQUINA_AnchoBobina = 10;
        //Parámetros Iniciales de creación de pedidos
        public static int PEDIDO_Retardo = 500;
        //Retraso en la creación entre pedidos para que no coja el mismo
        valor
        public static int PEDIDO_MinAltura = 1;
        //Altura mínima de la pieza que se creara en un pedido aleatorio
        public static int PEDIDO_MaxAltura = MAQUINA_AnchoBobina;
        //Altura  máxima de la pieza que se creara en un pedido aleatorio
        public static int PEDIDO_MinAnchura = 1;
        //Anchura mínima de la pieza que se creara en un pedido aleatorio
        public static int PEDIDO_MaxAnchura = MAQUINA_AnchoBobina * 3 / 4;
        //Anchura  máxima de la pieza que se creara en un pedido aleatorio
        public static int PEDIDO_MinCantidad = 1;
        //Cantidad mínima de la pieza que se creara en un pedido aleatorio
        public static int PEDIDO_MaxCantidad  = 100;
        //Cantidad máxima de la pieza que se creara en un pedido aleatorio
        public static bool PEDIDO_HabilitarRotacion = false;
        //Valor por defecto de la posibilidad de rotación de las piezas del
        pedido
        public static int PEDIDO_Generados = 5;
        //Cantidad de pedidos generados cada vez que se pulsa en generación
        aleatoria

         //Parámetros Iniciales de creación de semilla
        public static string SEMILLA_Metodo = "Aleatoria";
        //Seleccionar uno de todos los métodos disponibles
        public static int SEMILLA_MetodoAleatoria_ProbabilidadNuevaCapa =
50;
        //Valores entre 1 y 100. 1: muy poca - 100: seguro

        //Parámetros Iniciales de generación
        public static int GENERACION_MaxPoblacion = 20;
        //N° de soluciones que formarán la población de una generación
        public static int GENERACION_Retardo = 500;
        //Retraso en la creación entre soluciones para que no coja el mismo
        valor
        public static int GENERACION_NumGeneracionInicial = 0;
        //Valor por defecto de la generación inicial
        public static int GENERACION_IntervaloGeneraciones = 5;
        //Cuantas generaciones se crearán cada vez que se pulse el botón
        public static int GENERACION_RepeticionMejores = 2;
        //Cuantas soluciones se copian directamente a la generación
        siguiente
        public static int GENERACION_IntentosCombinacionSoluciones = 2;
        //Cuantas veces se intenta combinar dos soluciones existentes

        //Probabilidad de seleccionar un operador u otro.
        public static int GENERACION_ProbabilidadOperadorSemilla = 20;
        //Probabilidad de que la solución se genere desde 0
        public static int GENERACION_ProbabilidadOperadorCruce = 20;
        //Probabilidad de que la solución se genere con las mejores capas de
        los padres
        public static int GENERACION_ProbabilidadOperadorMutacion = 20;
        //Probabilidad de que la solución se genere con % de capas
        aleatorias y % de mejores de los padres
        public static int GENERACION_ProbabilidadOperadorCombinacionCapas =
20;
```

```
        //Probabilidad de que se copie una solución padre con posibilidad de
        que se optimice a si misma
        public static int
GENERACION_ProbabilidadOperadorCombinacionSoluciones = 20;
        //Probabilidad de que la solución se genere con % de capas
        aleatorias de los padres y % de mejores de capas nuevas
        //Parametrizacion de cada operador
        public static int GENERACION_OperadorSemilla_ProbabilidadAleatorio =
0;
        //Parametrizacion del operador genérico, para seleccionar un % de
        capas aleatorias
        public static int GENERACION_OperadorSemilla_ProbabilidadMejores =
0;
        //Parametrizacion del operador genérico, para seleccionar un % de
        las mejores capas
        public static int
GENERACION_OperadorSemilla_ProbabilidadRecombinacion = 0; /
        /Probabilidad de que además, se realice un intento de recombinación
        public static int GENERACION_OperadorCruce_ProbabilidadAleatorio =
0;
        public static int GENERACION_OperadorCruce_ProbabilidadMejores =
100;
        public static int GENERACION_OperadorCruce_ProbabilidadRecombinacion
= 0;
        public static int GENERACION_OperadorMutacion_ProbabilidadAleatorio
= 30;
        //Si se ponen los dos valores al 40, no suele quedar espacio para
        capas nuevas (semilla)
        public static int GENERACION_OperadorMutacion_ProbabilidadMejores =
30;
        public static int
GENERACION_OperadorMutacion_ProbabilidadRecombinacion = 0;
        public static int
GENERACION_OperadorCombinacionCapas_ProbabilidadAleatorio = 0;
        public static int
GENERACION_OperadorCombinacionCapas_ProbabilidadMejores = 100;
        //Si solo se envía una solución, con el 100 de copiar las mejores,
        se copiará tal cual
        public static int
GENERACION_OperadorCombinacionCapas_ProbabilidadRecombinacion = 100;
        public static int
GENERACION_OperadorCombinacionSoluciones_ProbabilidadAleatorio = 90;
        public static int
GENERACION_OperadorCombinacionSoluciones_ProbabilidadMejores = 0;
        public static int
GENERACION_OperadorCombinacionSoluciones_ProbabilidadRecombinacion = 0;
    }
```

Clase Parametros:

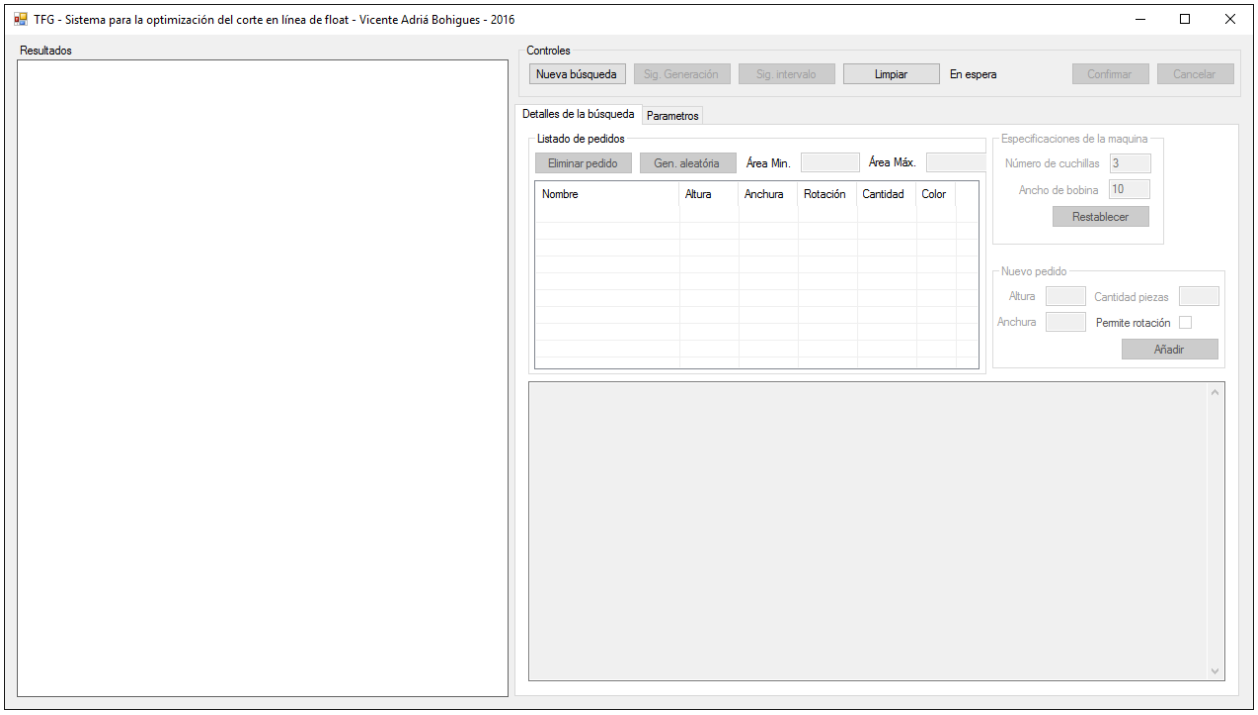


Las variables de esta clase son las mismas que las de la clase Parámetros Iniciales, por tanto no se repetirán aquí sus descripciones.

```
public class Parametros
    {
        //Parámetros de creación de una maquina
        public int MAQUINA_NumCuchillas;
        public int MAQUINA_AnchoBobina;
        //Parámetros de creación de pedidos
        public int PEDIDO_Retardo;
        public int PEDIDO_MinAltura;
        public int PEDIDO_MaxAltura;
        public int PEDIDO_MinAnchura;
        public int PEDIDO_MaxAnchura;
        public int PEDIDO_MinCantidad;
        public int PEDIDO_MaxCantidad;
        public bool PEDIDO_HabilitarRotacion;
        public int PEDIDO_Generados;
        //Parámetros de creación de semilla
        public string SEMILLA_Metodo;
        public int SEMILLA_MetodoAleatoria_ProbabilidadNuevaCapa;
        //Parámetros de generación
        public int GENERACION_MaxPoblacion;
        public int GENERACION_Retardo;
        public int GENERACION_NumGeneracionInicial;
        public int GENERACION_IntervaloGeneraciones;
        public int GENERACION_RepeticionMejores;
        public int GENERACION_IntentosCombinacionSoluciones;
        //Probabilidad de seleccionar un operador u otro.
        public int GENERACION_ProbabilidadOperadorSemilla;
        public int GENERACION_ProbabilidadOperadorCruce;
        public int GENERACION_ProbabilidadOperadorMutacion;
        public int GENERACION_ProbabilidadOperadorCombinacionCapas;
        public int GENERACION_ProbabilidadOperadorCombinacionSoluciones;
        //Parametrización de cada operador
        public int GENERACION_OperadorSemilla_ProbabilidadAleatorio;
        public int GENERACION_OperadorSemilla_ProbabilidadMejores;
        public int GENERACION_OperadorSemilla_ProbabilidadRecombinacion;
        public int GENERACION_OperadorCruce_ProbabilidadAleatorio;
        public int GENERACION_OperadorCruce_ProbabilidadMejores;
        public int GENERACION_OperadorCruce_ProbabilidadRecombinacion;
        public int GENERACION_OperadorMutacion_ProbabilidadAleatorio;
        public int GENERACION_OperadorMutacion_ProbabilidadMejores;
        public int GENERACION_OperadorMutacion_ProbabilidadRecombinacion;
        public int
GENERACION_OperadorCombinacionCapas_ProbabilidadAleatorio;
        public int GENERACION_OperadorCombinacionCapas_ProbabilidadMejores;
        public int
GENERACION_OperadorCombinacionCapas_ProbabilidadRecombinacion;
        public int
GENERACION_OperadorCombinacionSoluciones_ProbabilidadAleatorio;
        public int
GENERACION_OperadorCombinacionSoluciones_ProbabilidadMejores;
        public int
GENERACION_OperadorCombinacionSoluciones_ProbabilidadRecombinacion;
        //Constructor de la clase que inicializa los parámetros a los
       valores de ParametrosIniciales
        public Parametros() {...}
        //Este método inicializa los parámetros de la maquina
        public void inicializarMaquina() {...}
        //Este método inicializa los parámetros del generador de pedidos
```

```
        public void inicializarGeneradorPedidos() {...}
        //Este método inicializa los parámetros de la semilla
        public void inicializarSemilla() {...}
        //Este método inicializa los parámetros de las generaciones
        public void inicializarGeneracion() {...}
        //Este método inicializa los parámetros de selección de operador
        public void inicializarSeleccionOperadores() {...}
        //Este método inicializa los parámetros de configuración de los
        operadores
        public void inicializarConfiguracionOperadores() {...}
    }
```

Clase FormPrincipal:

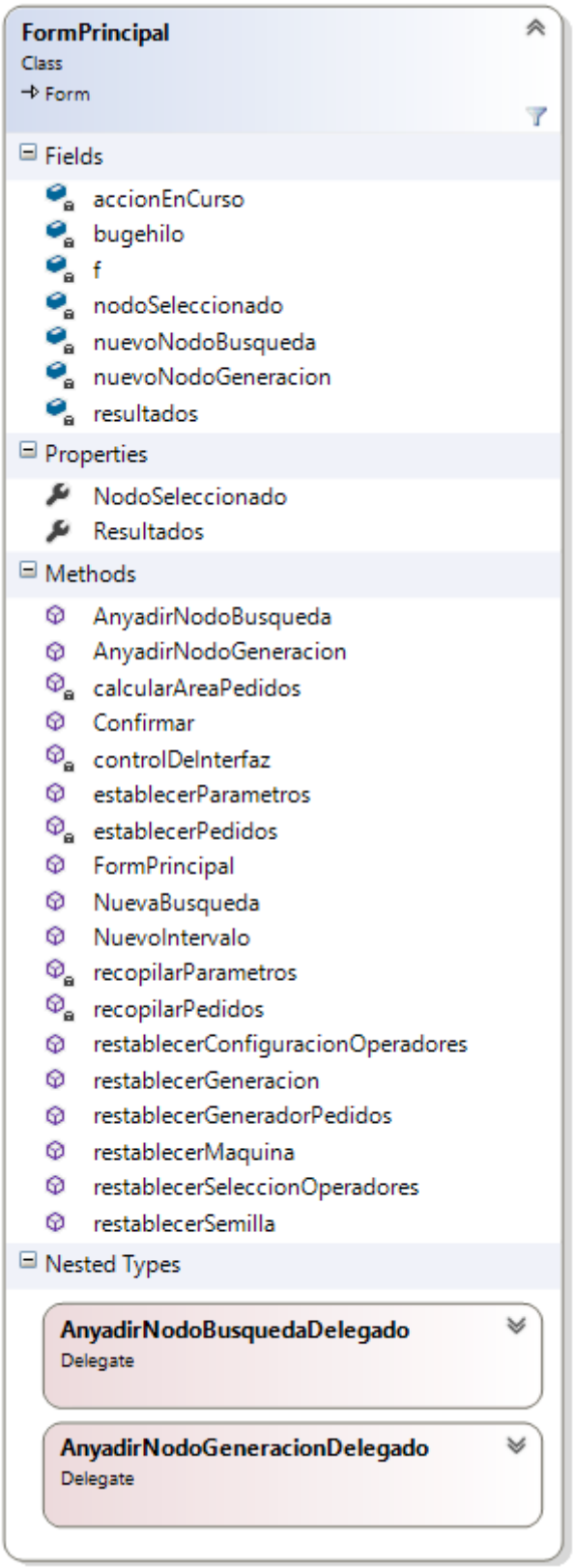


En la imagen representativa de la clase, se han ocultado los ítems correspondientes a controles de la interfaz y sus métodos para no excederse en tamaño.

En esta clase, a diferencia de las anteriores, no se han ordenado los métodos por orden alfabético sino por grupos relacionados por su funcionalidad. Además, como muchos de ellos son básicos y repetitivos, solo se muestran aquí algunos de los más importantes.

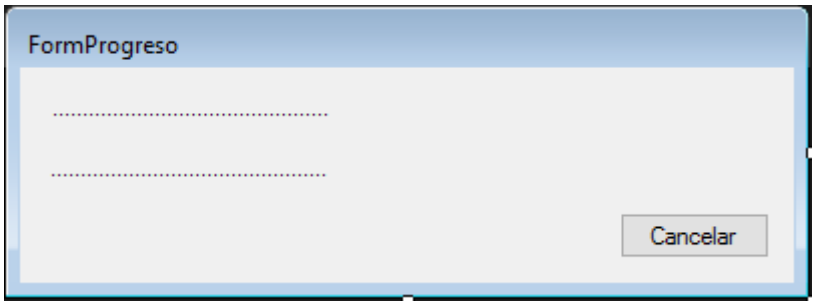
```
public partial class FormPrincipal : Form
    {
        //Variables privadas
        private ArrayList resultados; //Listado de búsquedas realizadas en
        la aplicación
```

```
        private TreeNode
        nodoSeleccionado; //Nodo
        seleccionado del árbol. Sirve
        para tener una referencia del
        nodo seleccionado aunque el
        control pierda el foco
        private string accionEnCurso;
        //Acción en curso. Sirve para
        gestionar la interfaz
        //Variables globales utilizadas
        por el programa a través de hilos
        public FormProgreso f;
        //Variable que contiene el
        formulario de progreso para poder
        interactuar con el
        public BusquedaGenetica bugehilo;
        //Variable que contiene la
        búsqueda actual para poder
        interactuar con ella
        public TreeNode
nuevoNodoGeneracion;
      //Nodo a añadir tras un proceso de
      búsqueda
       public TreeNode
       nuevoNodoBusqueda; //Nodo a
       añadir tras un proceso de
       búsqueda
       //Propiedades publicas que
      encapsulan las variables privadas
      {get; set}
      public ArrayList Resultados {...}
      public TreeNode NodoSeleccionado
{...}
      //Constructor del formulario
      principal
      public FormPrincipal(){...}
      /*********************************
*/
      /* Bloque de métodos de
      inicialización de la interfaz
      */
      /*********************************
*/
      public void restablecerMaquina()
{...}
      public void
restablecerGeneradorPedidos() {...}
      public void restablecerSemilla()
{...}
      public void
restablecerGeneracion() {...}
      public void
restablecerSeleccionOperadores() {...}
      public void
restablecerConfiguracionOperadores()
{...}
      /*********************************/
      /* Bloque de métodos de
      interacción del usuario con la interfaz       */
      /*********************************/
      ...
      /*****************************************************************/
      /* Bloque de métodos con interacción entre el modelo y la interfaz*/
      /*****************************************************************/
```

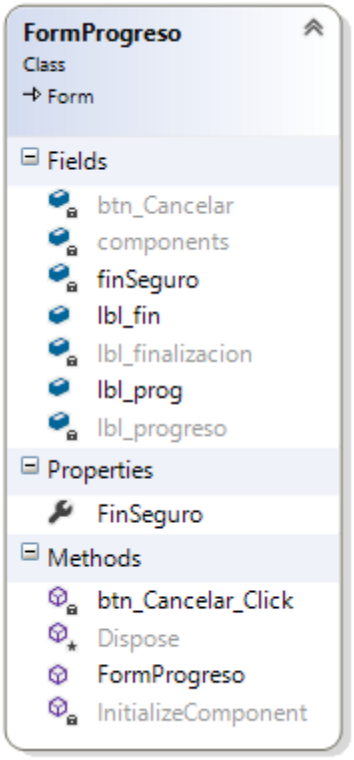


```
//Este método calcula el área máxima y mínima utilizada por las
piezas de los pedidos
private void calcularAreaPedidos(){...}
//Este método crea un ArrayList con los pedidos del listado de la
interfaz
private ArrayList recopilarPedidos(){...}
//Este método rellena el listado de pedidos de la interfaz desde un
ArrayList de pedidos
private void establecerPedidos(ArrayList peds) {...}
//Este método crea una instancia de la clase Parametros desde los
valores introducidos en la interfaz
private Parametros recopilarParametros(){...}
//Este método rellena los valores de los parámetros en la interfaz
desde una instancia de la clase Parametros
public void establecerParametros(Parametros param) {...}
/*****************************************************************/
/* Bloque de Lógica de validación de datos de la vista (interfaz) */
/*****************************************************************/
...
/*********************************/
/* Bloque de Lógica de aplicación */
/*********************************/
//Método que modifica los controles activos y desactivados en
función de la acción que se esté realizando
private void controlDeInterfaz(string control) {...}
//Este método contiene la lógica necesaria para generar una nueva
búsqueda y actualizar el formulario de progreso y el formulario
principal según el estado del proceso
public void NuevaBusqueda(){...}
//Definición de un método delegado para poder actualizar el
formulario principal desde el hilo que realiza el proceso de
búsqueda
public delegate void AnyadirNodoBusquedaDelegado();
//Este método añade un nodo búsqueda al árbol de resultados. Invoca
a su delegado en caso de ser necesario
public void AnyadirNodoBusqueda(){...}
//Este método contiene la lógica necesaria para generar un nuevo
intervalo de generaciones y actualizar el formulario de progreso y
el formulario principal según el estado del proceso
public void NuevoIntervalo(){...}
//Definición de un método delegado para poder actualizar el
formulario principal desde el hilo que realiza el proceso de
búsqueda
public delegate void AnyadirNodoGeneracionDelegado();
//Este método añade un nodo generación al árbol de resultados.
Invoca a su delegado en caso de ser necesario
public void AnyadirNodoGeneracion(){...}
//Este método ejecuta la coliga de confirmación de una acción en
aquellas acciones que lo requieran
public void Confirmar()
 {
      ...
      //Si la acción es una nueva búsqueda se lanza el proceso de
      nueva búsqueda en un hilo separado de la ejecución principal
      ...
      //Si la acción es calcular la siguiente generación se realiza
      el proceso directamente en el hilo principal
      ...
      //Si la acción es un nuevo intervalo se lanza el proceso de
      nuevo intervalo en un hilo separado de la ejecución principal
      ...
      //Si la acción es limpiar, se restablecen los campos al valor
      por defecto al inicio de la ejecución del programa
      ... }}}
```

Clase FormProgreso:

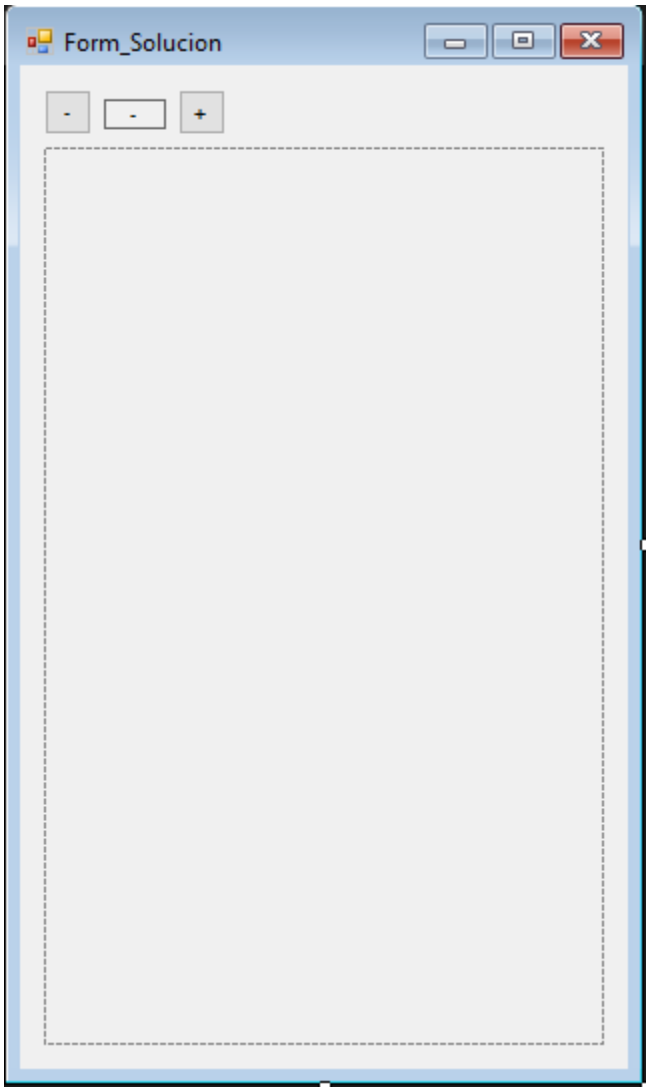


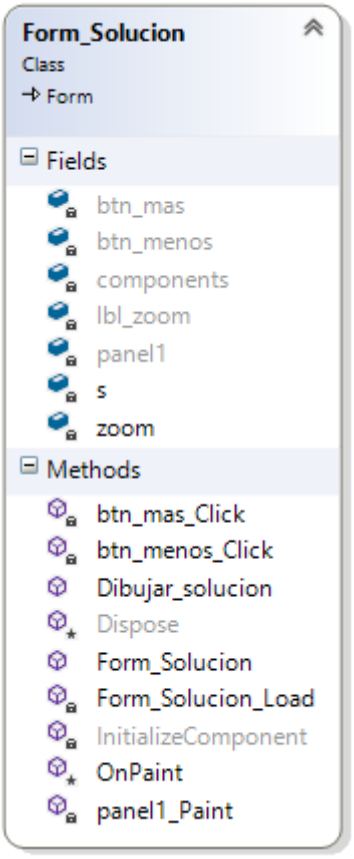
Esta clase define el formulario que se muestra cuando hay un proceso de búsqueda en segundo plano. Algunos de sus controles se han definido como públicos para poder modificar su contenido desde otros hilos de ejecución.




```
public partial class FormProgreso : Form
{
    //Variables del formulario
    private bool finSeguro; //Variable que
    indica si se ha pulsado el botón cancelar
    public Label lbl_prog; //Etiqueta que
    mostrará el estado del proceso
    public Label lbl_fin; //Etiqueta que
    muestra la acción de finalizar el proceso
    actual
    //Propiedades publicas que encapsulan las
    variables privadas {get; set}
    public bool FinSeguro {...}
    //Constructor de la clase
    public FormProgreso(){...}
    //Método que actualiza las variables
    relacionadas con la cancelación de la
    busqueda
     private void btn_Cancelar_Click(object
sender, EventArgs e) {...}
}
```

Clase Form_Solucion:






```
public partial class
Form_Solucion : Form
   {
        //Variables privadas
        private Solucion s;
        private int zoom;
//zoom que se aplicara al
dibujar la solucion
      //Constructor de la clase
      public Form_Solucion(Solucion s) {...}

      //Este método recorre y dibuja la solución que contiene el
      formulario
       public void Dibujar_solucion()
       {
           //Se redimensión al formulario
            ...
           //Se crea el área de dibujo
            ...
           //Se recorre la solución dibujando sus piezas
            ...
           //Se liberan recursos
             ...
       }
       //Este método aumenta el zoom y vuelve a dibujar la solución
       private void btn_mas_Click(object sender, EventArgs e) {...}
       //Este método reduce el zoom y vuelve a dibujar la solución
       private void btn_menos_Click(object sender, EventArgs e) {...}
      //Este método vuelve a dibujar la solución cuando se redibuja el
      formulario
       private void panel1_Paint(object sender, PaintEventArgs e) {...}
}
```