Pontifica Universidad Católica del Ecuador
Facultad de Ingeniería
Escuela de Sistemas

Disertación Previa a la Obtención del Título de Ingeniería de Sistemas y Computación

Aplicación de Algoritmos Genéticos para la Optimización de Corte de Material Bidimensional Rectangular

Autor:

Henry Daniel Trujillo Chunés

Director:

Ing. Guido Ochoa Moreno Msc.

Quito, marzo 2019

Tabla de contenido

Índice de Figuras

Índice de Tablas

Capítulo 1

Generalidades

1.1 **El Problema de Corte de Materiales**

"Este problema es denominado Problema de Corte de Existencias o Materiales (Cutting Stock Problem CSP[1]) por sus denominación original en inglés, fue identificado por primera vez en 1939 por Kantorovich en su publicación Métodos matemáticos de organización y planificación de producción", (Ben Amor & Valério de Carvalho, 2005). El problema descrito de manera muy general consiste en encontrar la mejor manera de cortar un conjunto de objetos de gran tamaño en objetos más pequeños.

Una de las aplicaciones prácticas más común de encontrar es el corte de rollos de material. Como ejemplo se ´puede tomar a una compañía que fabrica rollos de plástico. El proceso inicia con la generación de grandes rollos de alrededor de 6 metros de ancho y longitud variable, los cuales son cortados posteriormente en rollos más pequeños que mantienen la longitud del rollo del cual son obtenidos pero cuyo ancho varía entre los 0.3 y 1.2 metros dependiendo del requerimiento del cliente. La cantidad de rollos grandes que se cortarán depende de la cantidad y dimensiones de los rollos pequeños que se busca obtener, en este punto es en donde se debe lidiar con el CSP ya que se tiene que determinar en qué orden se realizará el corte evitando desperdicios y aprovechando al máximo el rollo grande.

1.2 Planteamiento del problema

En diversos tipos de industrias es común encontrar materiales que vienen dispuestos en unidades de dimensiones estándar, las cuales deben ser cortadas para poder ser usadas. Este proceso de corte genera desperdicios y suele representar dificultades en su ejecución pues no resulta fácil identificar la manera correcta de proceder a cortar el material. Además, está el

[1] CSP: Cutting Stock Problem

hecho de que al no ejecutarse adecuadamente el corte de materiales se incurre en gastos extra para el tratamiento o reúso de los residuos generados durante el proceso. Este último punto hace evidente que el proceso de corte está ligado a la optimización ya que, si hablamos de residuos, las industrias que aplican el corte de materiales buscarán lograr un aprovechamiento de los recursos; no solamente de materia prima a tratar sino también del tiempo y personal involucrados en el proceso debido a su complejidad. Con el paso de los años y desde su descubrimiento, se han formulado diversas maneras de dar solución al problema de corte de materiales

1.3 Justificación

La presente investigación pretende resolver el problema de corte de material usando la capacidad de mejora continua con la que cuentan los algoritmos genéticos, esto con la finalidad de alcanzar una solución óptima y viable para el problema de corte de materiales en el caso de que esto sea posible.

Muchas de las formulaciones propuestas para resolver el problema de corte de materiales se enfocan en resolver conjuntos muy reducidos del espacio de búsqueda de la solución. Para su implementación se debe contar con una gran cantidad de parámetros predefinidos que dificultan la construcción y hacen que las soluciones obtenidas sean muy limitadas, teniendo así que construir múltiples métodos de resolución para abarcar las diferentes variaciones del problema. Al usar un algoritmo genético se reduce la cantidad de métodos usados para resolver el problema y a la vez se incrementa el espacio de solución que se explora y la cantidad de soluciones que se obtienen.

La investigación se desarrollará realizando un repaso a la bibliografía relacionada al problema de corte de materiales, las soluciones más estudiadas que se han planteado para el mismo, así como fundamentos de optimización y por supuesto sobre algoritmos genéticos. Con base en dicha revisión bibliográfica se pretende diseñar y construir un algoritmo genético que

permita obtener soluciones para el problema del corte de materiales bidimensionales rectangulares.

1.4 Objetivos

Objetivo General

Desarrollar un algoritmo genético que permita dar solución al problema del corte de materiales en dos dimensiones usando la capacidad adaptativa del mismo, para obtener una solución factible que maximice el uso de material y minimice el desperdicio de este; esto mediante la creación de una aplicación de software que permita visualizar la solución.

Objetivos Específicos

- Analizar los diferentes tipos de algoritmos genéticos para determinar cuál se usará en la solución del problema.
- Diseñar, construir y probar un algoritmo genético que permita resolver el problema del corte de material de tipo orientado con guillotina.
- Implementar una aplicación de software para usar el algoritmo genético.

1.5 Alcance

El presente trabajo de disertación se centrará en construir un algoritmo genético que permita obtener soluciones válidas para una de las cuatro principales variantes del CSP, en capítulos posteriores se discutirán estas variantes y se aclarará sobre cuál de ellas se trabajará. El trabajo se enfocará en diseñar y construir todos y cada uno de los componentes que integran y permiten el funcionamiento de un algoritmo genético. Este trabajo se enfocará enteramente en proponer una solución alterna a los métodos existentes usados para tratar el CSP, esto mediante la demostración del funcionamiento del algoritmo.

Capítulo 2

El problema del corte de materiales

En este capítulo se hace una revisión del problema de corte de materiales o existencias (CSP), iniciando con una descripción muy breve y que puede aplicarse en todos los campos en los que aparece este problema, los objetivos que se persigue al resolver este problema, así como la forma en que se cataloga y los parámetros respecto a los cuales es aplicado. Finalmente se hace mención de las soluciones más destacadas que se han propuestas para resolver el CSP.

2.1 Descripción del problema bidimensional de corte de materiales

"En investigación de operaciones, el problema del corte de materiales hace referencia al corte de piezas de material de tamaño estándar en piezas de tamaños especificados, esto mientras se minimiza el material desperdiciado o se maximiza el material utilizado", (Dyson & Gregory, 1974).

Este problema surge de aplicaciones en diversas industrias tales como la del vidrio, madera, impresión, almacenaje y muchas otras. Dependiendo de la industria la aplicación del problema puede darse en una, dos y hasta tres dimensiones, consistiendo siempre en una de las siguientes tareas:

- Maximizar el número de ítems[2] requeridos que se obtienen de una unidad de tamaño estándar.
- Minimizar la cantidad de material desperdiciado al cortar las piezas deseadas.
- Minimizar la cantidad de unidades estándar usadas para obtener las piezas requeridas.

La figura 1 muestra cómo podría verse una solución del problema de corte de materiales aplicado en dos dimensiones; en este caso la unidad estándar está representada por el rectángulo

[2] Ítem: Elemento rectangular que se obtendrá a partir de una lámina de dimensiones estándar.

que contiene a todos los elementos de las figuras, el área desperdiciada se representa con las secciones que tienen una cuadrícula y los ítems a cortar son todos los rectángulos contenidos en la unidad estándar.

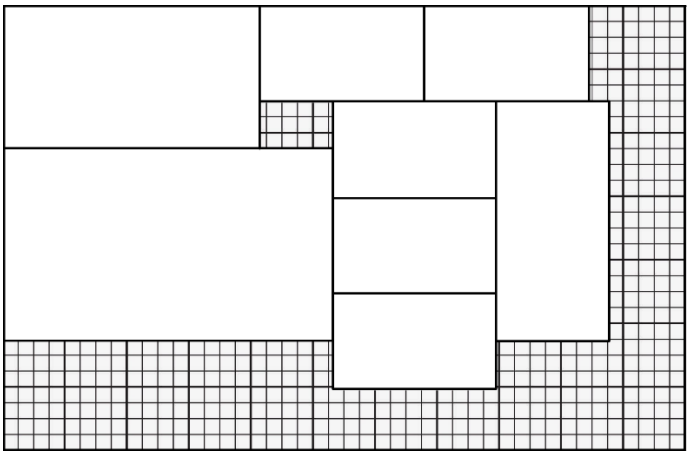

Figura 1. Representación gráfica del CSP.

Hablando específicamente del CSP bidimensional, sin importar la industria analizada, la aplicación en la cual surge el problema tendrá dos componentes en común: "las existencias estándar serán rectangulares y el objetivo será ubicar u obtener la mayor parte de ítems requeridos usando el menor número de unidades", (Lodi, Martello, & Monaci, 2002). Esta variante del problema requiere que los bordes de los ítems sean ubicados de forma paralela a los bordes de las unidades estándar. El problema puede ser catalogado de la siguiente manera, tomando en cuenta características y restricciones descritas a continuación, (Oliveira, Neuenfeldt Júnior, Silva, & Carravilla, 2016):

- Es un problema de dimensión abierta: No se tiene restricciones en cuanto a la dimensión de las existencias o de los ítems.
- La lista de ítems puede ser offline u online: Cada ítem puede ser introducido en la unidad estándar tomando en cuenta las dimensiones del resto de ítems (offline) o puede ser introducido de manera aislada sin tomar en cuenta al resto de ítems (online).
- La geometría es rectangular: La forma de los ítems a obtener es rectangular.

- Aplicado a dos dimensiones: Al ser de geometría rectangular el problema se convierte un uno aplicado a dos dimensiones.
- De tipo ortogonal: Los bordes de las existencias son paralelos a los bordes de los ítems.
- Con restriccion en su orientación: Los ítems pueden tener su ubicación fija o estos pueden ser rotados (90°).
- Cortes de guillotina: Puede o no ser impuesto el obtener los ítems mediante una secuencia de cortes paralelos de borde a borde en las unidades estándar.

En base a las dos últimas características, el problema puede subdividirse de la siguiente manera:

- Orientado sin guillotina: Las piezas se mantienen fijas y los cortes no necesariamente requieren ser de guillotina. (Ver figura 2)

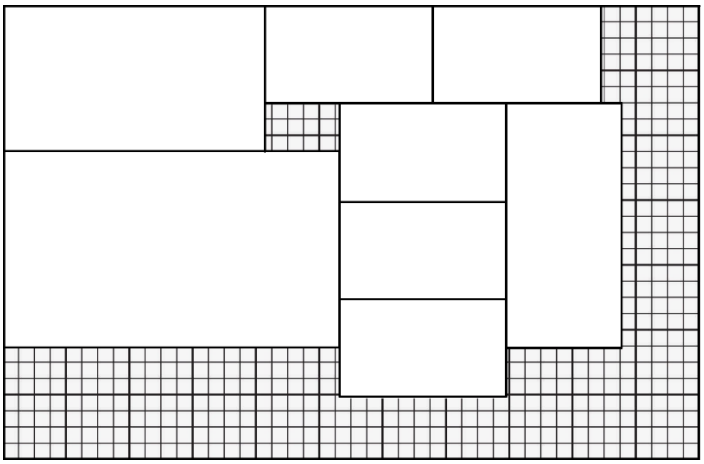

Figura 2. Corte sin rotación y sin guillotina.

- Con rotación sin guillotina: Las piezas son capaces de girar 90° y los cortes no necesariamente requieren ser de guillotina. (Ver figura 3)

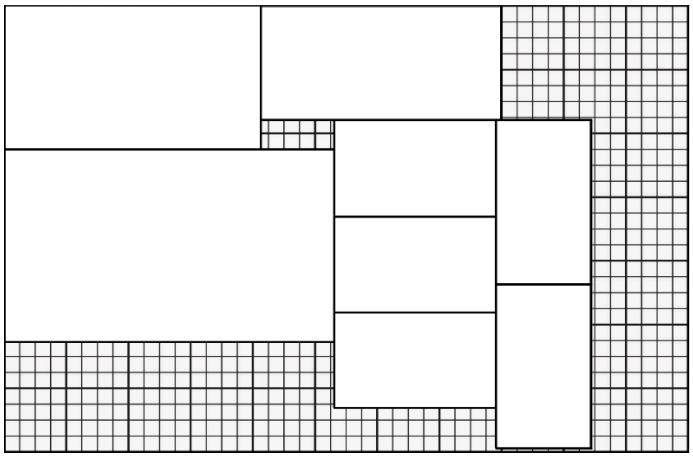

Figura 3. Corte con rotación y sin guillotina.

- Orientado con guillotina: Las piezas se mantienen fijas y los cortes deben ser de guillotina. (Ver figura 4)

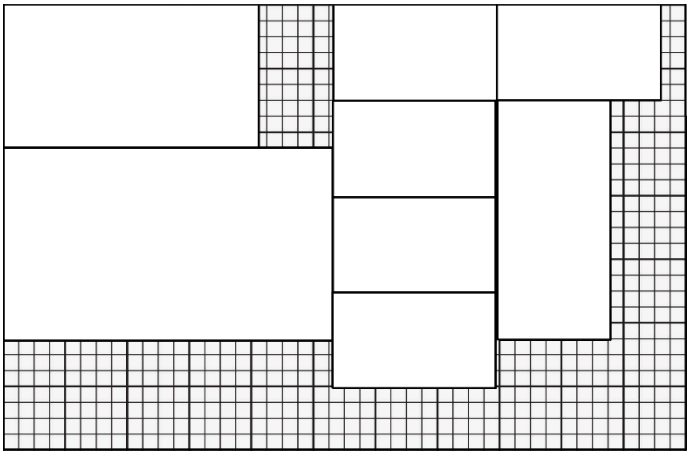

Figura 4. Corte sin rotación y de guillotina.

- Con rotación y guillotina: Las piezas son capaces de girar 90° y los cortes deben ser de guillotina. (Ver figura 5)

En su forma más simple el problema de corte de materiales consiste en determinar un conjunto de instrucciones (patrón de corte) que minimizarán una función objetivo. Cada instrucción de corte determina como una unidad de material debería ser cortada para producir un cierto número de piezas requeridas, sin embargo, "a lo largo de su estudio se determinó que, al ser expresado como un problema de programación entera, debido al gran número de variables

involucradas, su cálculo no era factible. Esto ocurría también al buscar una solución aproximada mediante programación lineal; pues ciertos problemas de la programación lineal que parten de problemas combinatorios se vuelven intratables al requerir de múltiples variables", (Gilmore & Gomory, 1961).

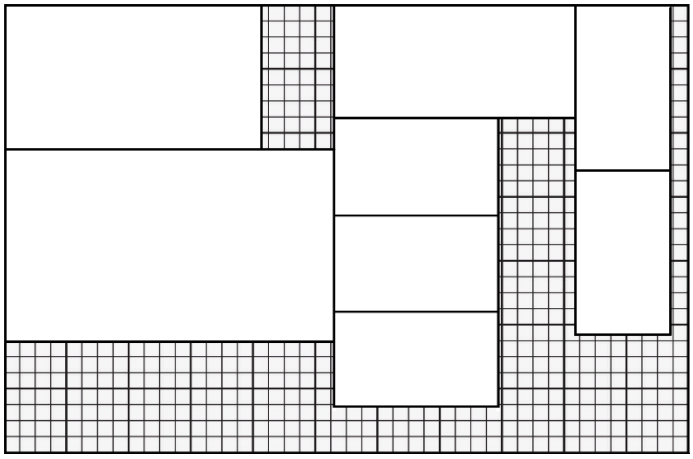

Figura 5. Corte con rotación y de guillotina.

2.1.1 Complejidad del problema de corte de materiales. Para entender de mejor manera la complejidad asociada a la resolución del CSP y dado que una de las principales características que describe a un algoritmo es su complejidad, es necesario repasar algunos conceptos sobre complejidad computacional de problemas.

La complejidad de problemas o algoritmos estudia la manera en la que se clasifican los problemas de acuerdo con la dificultad inherente de resolverlos, principalmente asociada a los parámetros de tiempo o espacio requerido para su resolución.

La complejidad temporal trata al tiempo requerido para solucionar un problema; siempre se busca la solución con menor complejidad temporal. La complejidad espacial analiza el espacio requerido por las estructuras de datos de un algoritmo para solucionar un problema, está ligada a la complejidad temporal.

Una de las clasificaciones de complejidad más usadas corresponde a:

Problemas Polinomiales (P). – Hace referencia a los problemas en los cuales se conoce un algoritmo que puede resolverlo en un tiempo polinomial. Son problemas que se resuelven en tiempos razonables mediante la ejecución de operaciones que crecen de forma polinómica dependiendo del tamaño de entradas del problema.

Problemas No-Polinomiales (NP). – Son problemas en los que se conoce un algoritmo que es capaz de comprobar si algo es o no solución del problema en un tiempo polinómico. En estos problemas se comprueba en tiempo razonable si algo dado es solución del problema.

El CSP está dentro del conjunto de problemas definidos como NP, concretamente es un problema de tipo NP-hard, esto quiere decir que una de sus soluciones puede utilizarse para determinar en tiempo polinomial la solución de todos los problemas NP del mismo tipo. Esto hace que optimizar el proceso de corte sea sumamente complicado, de allí que con el pasar de los años desde su formulación se han propuesto múltiples formas de resolver el problema.

2.2 Soluciones propuestas previamente al problema

Diversos autores han planteado métodos o aproximaciones para dar solución al problema desde que éste fue descrito formalmente. Con el pasar de los años, algunas de las soluciones propuestas han sido dejadas de lado ya sea por su complejidad o porque aparecieron soluciones que generaban mejores resultados. Por el contrario, existen aproximaciones para resolver el problema que con el pasar del tiempo se han mantenido o han sido mejoradas con la contribución de nuevos autores; esto se debe principalmente a que los tiempos de cálculo computacional están dentro de límites razonables. Entre los métodos propuestos se pueden encontrar:

- Corte de guillotina de dos etapas, (Gilmore & Gomory, 1965).
- Algoritmo de la mejor tira finita, (Berkey & Wang, 1987).
- Algoritmo para generación de patrones mediante un árbol búsqueda, (Suliman, 2001).

2.2.1 Corte de guillotina de dos etapas

Esta solución plantea resolver el problema como si se tratara de un problema de programación lineal usando un patrón de corte ininterrumpido en dos etapas. "Un patrón de corte es una instrucción que determina como debe ser cortada una placa (pieza rectangular de material) para producir una orden de placas requeridas", (Dyson & Gregory, 1974).

Para dar solución al problema se parte del supuesto de que se requieren N_i piezas de dimensiones $l_i \times w_i$ y, además, se dispone de una cantidad ilimitada de unidades estándar de dimensiones $L_k \times W_k$. El componente fundamental para llegar a la solución consiste en determinar los patrones de corte adecuados para aplicarlos en las dos etapas; los patrones se obtienen mediante una matriz en la cual cada una de sus columnas corresponde a un posible patrón de corte. Dichas columnas tienen la siguiente estructura $[a_1, a_2, \dots, a_m]$ en donde a_i es el número de rectángulos de dimensiones $l_i \times w_i$ que se obtienen del patrón. Las etapas de solución son las siguientes, (Gilmore & Gomory, 1965):

1. Se procede a cortar las unidades estándar en tiras cuyo ancho corresponda a los diferentes anchos de las piezas que se requiere obtener.
2. La segunda etapa consiste en segmentar las tiras siguiendo las longitudes requeridas de cada una de las N_i piezas que se desea obtener.

En la Figura 6 puede observarse el resultado de la aplicación de las dos etapas de corte con guillotina.

Debe recalcarse que esta solución está planteada de la forma más sencilla, en la cual se asume que, de las tiras obtenidas en la primera etapa, se obtendrán piezas cuyo ancho (w_i) será exactamente igual al de las tiras, sin embargo, el problema puede generalizarse a uno de N-etapas ya que si las dimensiones de todas las piezas a obtener son diferentes sería necesario volver a aplicarlas en las piezas obtenidas inicialmente. Esto incrementa mucho la complejidad

de la solución, haciendo que además el tiempo requerido para obtener el resultado se incremente con cada nueva etapa. La Figura 7 muestra un posible resultado obtenido al aplicar el corte de guillotina en N-etapas.

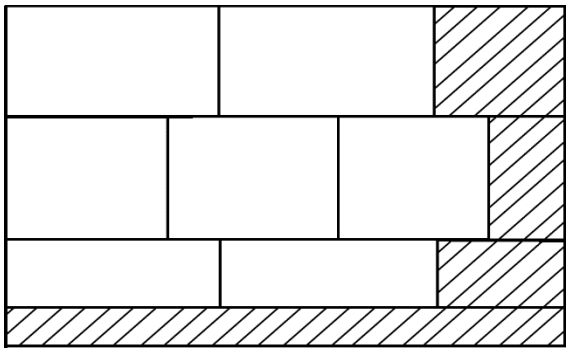

Figura 6. Corte de guillotina en dos etapas.

Fuente: Gilmore, P. C., & Gomory, R. (1965).

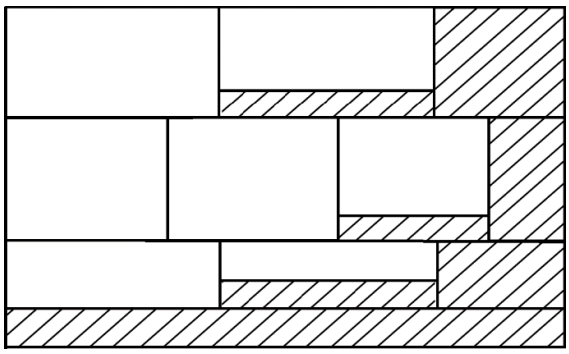

Figura 7. Corte de guillotina en múltiples etapas.

Fuente*:* Gilmore, P. C., & Gomory, R. (1965).

2.2.2 Algoritmo de la mejor tira finita

"Está solución pertenece a la familia de algoritmos de dos fases", (Lodi, 1999). En la primera fase de esta familia de algoritmos se ubican los ítems a obtener en una sola tira, es decir, se considera que la unidad rectangular estándar tiene un ancho W y una altura infinita; en la segunda fase se usa el resultado de la primera para obtener una solución aplicable a material rectangular de dimensiones finitas.

El algoritmo de la mejor tira finita inicia por acomodar los ítems en la tira de acuerdo con el mejor ajuste decreciente y sus etapas son:

1. Si el ítem actual no encaja en alguna de las repisas o subdivisiones generadas en la tira infinita, entonces se inicializa una nueva para este ítem; en caso de encajar en una de las subdivisiones existentes el ítem es ubicado en aquella en la cual se minimice el espacio horizontal restante.
2. Una vez ubicados todos los ítems en las repisas, estas son ubicadas en las unidades rectangulares estándar mediante la heurística del mejor ajuste decreciente de una dimensión: la siguiente repisa (la más alta) es ubicada en la unidad estándar que minimice la capacidad residual vertical o en una nueva unidad estándar en el caso de no poder ser ubicada en una de las que ya estén usadas.

En la Figura 8 está descrito un ejemplo de la aplicación del algoritmo de la mejor tira finita. En este caso se desea obtener siete ítems de diferentes dimensiones. El resultado de la primera fase puede observarse en Figura 8 (i) y el resultado de los ítems ubicados en las unidades estándar de encuentra en Figura 8 (ii).

2.2.3 Algoritmo para generación de patrones mediante un árbol de búsqueda. Lo que Suliman propuso fue encontrar un patrón de corte adecuado mediante un árbol de búsqueda, el cual tiene sus niveles asociados a los diferentes anchos de los ítems que se desean obtener de una unidad estándar y la cantidad requerida de cada uno de estos. Dichos niveles están asociados de manera decreciente ubicando en el nodo del primer nivel al ancho de la unidad estándar. El nodo inicial de cada nivel representa el ancho del residuo obtenido luego de que se haya satisfecho los cortes especificados en las ramas del nivel previo; finalmente los nodos ubicados en el nivel más alto del árbol representan las pérdidas resultantes. El algoritmo consta de los siguientes pasos:

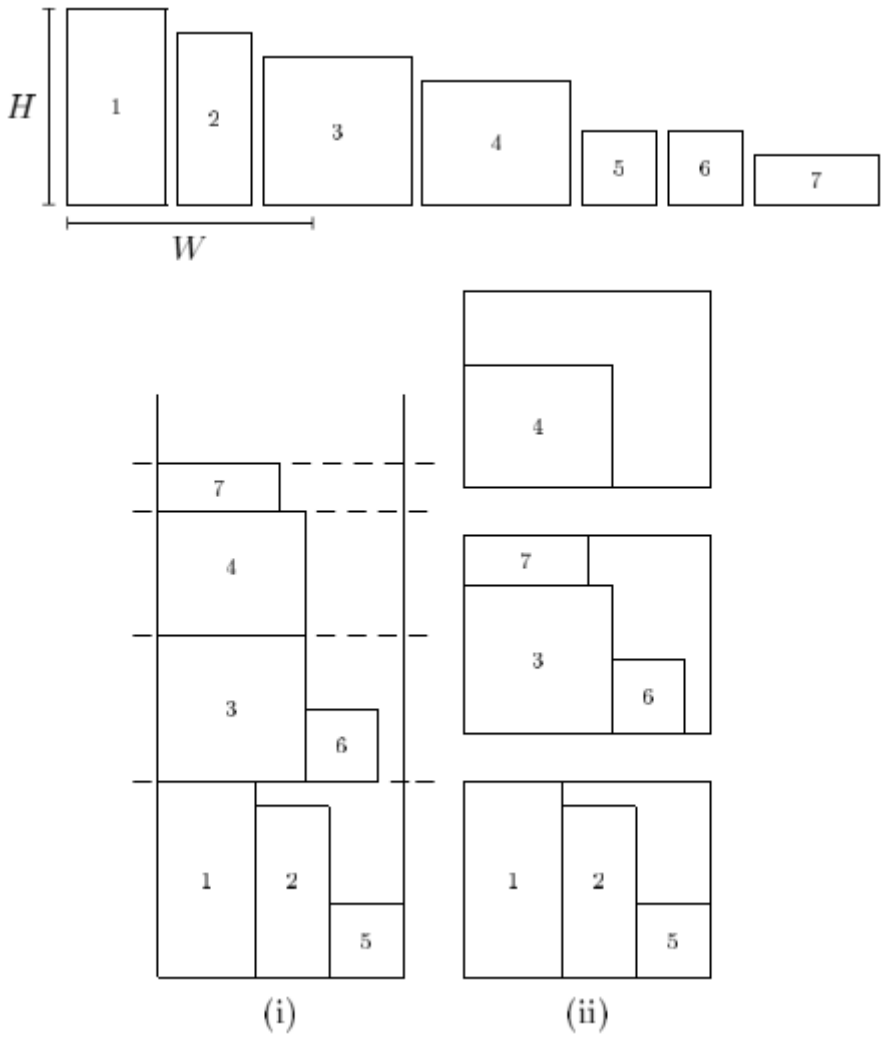

Figura 8. Aplicación de algoritmo mejor tira finita.

Fuente*:* Lodi, A., Martello, S., & Vigo, D. (1999).

1. Ordenar de manera decreciente el valor del ancho de cada uno de los ítems que se desea obtener.
2. Se obtiene el valor del entero más pequeño de la división entre el residuo del nivel previo del árbol y el ancho del ítem que se desea obtener para llenar la primera columna de la matriz.
3. Se calcula la pérdida producida en el corte del ítem obtenido en el paso previo.
4. Se asigna el valor del índice de la fila de la matriz.
5. Se verifica el valor del nodo actual (celda si hablamos de la matriz)

a. Si el valor en la celda es cero debe irse al paso 7.
b. Caso contrario se genera una nueva columna en la matriz con los siguientes valores
    i. Los nodos previos al nodo actual se llenarán con los valores que les preceden en la fila a la que pertenecen.
    ii. El nodo actual se llenará con el valor del que le precede en la fila disminuido en 1.
    iii. Los nodos restantes se llenarán usando el paso 2.

6. Se calcula la pérdida producida en el corte del ítem obtenido en el paso previo y se vuelve al paso 4.
7. Si aún restan ítems por obtener, se toma el ancho del siguiente ítem y se repite el paso 5 Caso contrario se termina la ejecución del algoritmo.

Este algoritmo necesita generar un árbol de búsqueda basado en las dimensiones de la unidad estándar, es decir que si se cuenta con unidades estándar de diversas dimensiones sería necesario aplicar el algoritmo para cada una de ellas.

Capítulo 3

Algoritmos Genéticos

Los algoritmos genéticos son técnicas usadas en inteligencia artificial para encontrar soluciones optimas en problemas de búsqueda basados en teoría de selección natural y evolución biológica. Son una técnica que, si bien no garantizan hallar una solución perfecta u óptima en su totalidad, permiten encontrar soluciones satisfactorias para los objetivos inmediatos. En este capítulo haremos un breve repaso sobre optimización, para a continuación revisar los conceptos fundamentales de los algoritmos genéticos, entre los cuales están: origen, aplicabilidad, estructura y funcionamiento.

3.1 Optimización

"La optimización consiste en probar variaciones de una idea inicial y usando información adquirida mejorar dicha idea", (Haupt & Haupt, 2004). En el diseño, construcción e implementación de cualquier sistema de ingeniería es necesario tomar decisiones de distinto índole en cada una de las etapas; estas decisiones tienen por objetivo minimizar el esfuerzo o maximizar el beneficio. Ya que tanto el esfuerzo como el beneficio pueden expresarse como funciones de determinadas variables de decisión, la optimización se convierte en el proceso de encontrar las condiciones que entregan el máximo o mínimo valor de una función.

La optimización es una herramienta matemática que permite determinar si la solución obtenida después de procesar información asociada a un problema es la única solución o si existen otras y cual de dichas soluciones es la mejor que se podría encontrar. Un problema de optimización puede establecerse de la siguiente manera, (Rao, 2009):

$$\text{Encontrar } X = \begin{Bmatrix} x1 \\ x2 \\ \vdots \\ xn \end{Bmatrix} \text{ que minimiza } f(X) \qquad (1)$$

Sujeto a las restricciones

$$g_j(X) \leq 0, \;\; j = 1,2,\dots,m$$

$$l_j(X) = 0, \;\; j = 1,2,\dots,p$$

En esta definición X es un vector n-dimensional conocido como vector de diseño, $f(X)$ es la función objetivo o de ajuste y $g_j(X)$, $l_j(X)$ son las restricciones de desigualdad e igualdad respectivamente.

Vector de diseño. – Todos los sistemas o problemas de ingeniería están definidos por un conjunto de cantidades, algunas variarán durante el proceso y otras no lo harán. A las que permanecen fijas se les conoce como parámetros preasignados y las demás se les llama variables de decisión. Son las variables de decisión las que conforman al vector de diseño.

Función objetivo o función de costo. – Es el criterio en base al cual se optimiza el problema planteado con las variables de decisión. Esta función se escoge de acuerdo con la naturaleza del problema y dependiendo de qué criterio del problema se desee maximizar o minimizar.

Costo. – Salida o resultado de la optimización, se trata de lo que se desea maximizar o minimizar mediante la aplicación de la optimización.

3.2 Conceptos básicos de algoritmos genéticos

Los algoritmos genéticos son métodos adaptativos usados generalmente para tratar con problemas de búsqueda y optimización de parámetros, basados en la reproducción y en el principio de supervivencia del más apto. De manera más formal Goldberg dice que "los algoritmos genéticos son algoritmos de búsqueda basados en la mecánica de la selección natural y de la genética natural", (Goldberg, 1989). Llegan a combinar elementos tales como la supervivencia del individuo más apto con un intercambio estructurado de información, el cual por lo general se realiza de manera aleatoria; construyendo así un algoritmo de búsqueda que

tenga algo de las características de búsqueda humanas. Estos algoritmos le permiten a una población compuesta por varios individuos evolucionar bajo reglas determinadas hasta alcanzar un estado en el cual se maximiza el ajuste.

Los algoritmos genéticos usan exploración aleatoria del espacio del problema combinada con procesos evolutivos para mejorar sus suposiciones sobre la solución del problema que buscan resolver. Además, ya que estos algoritmos no tienen experiencia en el dominio del problema, ellos prueban todo tipo de posibles soluciones que les es factible generar. "Es decir, una persona usando un algoritmo genético puede aprender más acerca del problema y sus posibles soluciones. Esto les brinda a las personas la habilidad de hacer mejoras al algoritmo, en un ciclo de mejora continua", (Sheppard, 2016).

Los algoritmos genéticos, o al menos sus predecesores, fueron usados por primera vez en la década de los cincuenta por biólogos que trataban de obtener modelos de la evolución natural. Su aplicación en el campo de la ingeniería llegó en la década de los sesenta cuando varios investigadores desarrollaron algoritmos basados en la evolución, sin embargo, estos no tenían componentes complejos como en la actualidad. Con el desarrollo de la programación evolutiva en las décadas siguientes aparecieron conceptos como población, cruzamiento, mutación y selección; de este modo su conceptualización pasó a ser la que se conoce en la actualidad. En este punto a este tipo de algoritmos se los denominaba sistemas digitales adaptativos pasando a llamarse posteriormente algoritmos genéticos.

El obtener la solución a un problema mediante la aplicación de algoritmos genéticos implica el tener que codificar dicho problema de manera que a los mecanismos de los que se sirve un algoritmo genético les sea posible contribuir a encontrar una respuesta. Para proceder a codificar un problema es importante entender que cualquier solución potencial de este puede ser presentada mediante una serie de parámetros a los cuales se les asigna valores. En el caso

de los algoritmos genéticos “el conjunto de todos los parámetros se codifica en una cadena de valores denominada cromosoma”, (Gestal, Rivero, Rabuñal, Dorado, & Pazos, 2010).

Al hablar de una cadena de valores, en el caso del cromosoma, es evidente que estos valores no siempre serán los mismos y por lo tanto se tendrán distintos cromosomas. De este hecho nace la necesidad de denominar genotipo al conjunto de parámetros representado por un cromosoma específico. El genotipo al ser un conjunto de parámetros contiene la información que se necesita para construir la solución real del problema, a la cual se conoce como fenotipo. En términos más sencillos el genotipo es el conjunto de información obtenida de los parámetros y el fenotipo es la expresión de dicha información.

La codificación de los elementos descritos en el párrafo anterior suele hacerse, no como regla general, mediante valores binarios. A cada parámetro se le asigna una cantidad determinada de bits y se discretiza la variable representada por cada uno de los genes. Los bits que se asignan a los parámetros dependerán del grado de ajuste que se busque lograr y no todos los parámetros deben estar necesariamente codificados con igual número de bits. Es importante señalar que los parámetros pueden codificarse con valores reales, enteros o de punto flotante.

Un algoritmo genético además de los cromosomas obtenidos mediante la codificación requiere de tres componentes fundamentales para encontrar la solución a un problema; la población inicial, función objetivo y los operadores genéticos.

3.2.1 Generación de la población inicial. La población inicial de un algoritmo genético no es nada más que un grupo de posibles soluciones del problema, en donde a cada una de ellas se le denomina también como individuo de la población.

De un conjunto de individuos pertenecientes al dominio del problema que se pretende optimizar, se elige de manera aleatoria a un determinado número de ellos con el objetivo de conseguir una población inicial heterogénea. Así se podrá buscar la solución óptima teniendo

en cuenta la diversidad del problema y abarcando todos los posibles resultados. A estos individuos se los evalúa de forma efectiva mediante la codificación.

Dado el caso de que el problema se haya estudiado previamente en forma parcial, en lugar de elegir aleatoriamente a la población inicial, esta puede ser elegida en base a cálculos previos.

3.2.2 Función objetivo. La función objetivo es la que el algoritmo buscará optimizar. Es un componente fundamental en la construcción de un algoritmo genético. En este tipo de algoritmos, la función objetivo es el ambiente en el cual los individuos de la población deberán adaptarse. Es el paso que determina como cambiaran los cromosomas a lo largo del tiempo, la función será la diferencia entre encontrar la solución óptima o no encontrar una solución. Al ser los algoritmos genéticos un método de optimización, su función objetivo se determina del mismo modo; es decir dependerá de la naturaleza del problema a tratar y de él o los parámetros a optimizar.

3.3 Operadores genéticos

Para abarcar una mayor diversidad de posibles soluciones tomando en cuenta una mayor parte de la población es necesario introducir variaciones en esta. Los operadores genéticos son los encargados de proporcionar heterogeneidad a las distintas poblaciones con las que se trabaja. Su propósito es permitir que el algoritmo obtenga mejores resultados al hacerlo más parecido a la realidad.

3.3.1 Evaluación de los individuos. La evaluación de individuos permite identificar que individuos en la población son los mejor adaptados. El método de evaluación al igual que la codificación dependerá del tipo de problema que se esté tratando y cada problema tendrá una derivación distinta de evaluación.

La función de evaluación de un algoritmo genético es la encargada de determinar si los integrantes de la población representan o no una buena solución al problema que se busca resolver. La función establece una medida numérica de la bondad de cada solución, la medida es conocida como ajuste. Esta medida se utiliza para controlar la forma en que se aplican los operadores genéticos en el algoritmo. El mecanismo usado más comúnmente es asignar una medida de ajuste para cada individuo en la población.

Los cuatro principales tipos de ajuste son:

Ajuste Puro. – Medida de ajuste definida en la terminología natural del problema que se está tratando. En los casos de maximización los individuos cuyo ajuste sea más elevado serán de interés para la solución, para los problemas de minimización aquellos individuos con ajuste reducido serán los interesantes.

$$r(i,t) = \sum_{j=1}^{Nc} |s(i,j) - c(i,j)| \tag{2}$$

$$r(i,t): Bondad\ del\ individuo\ i\ en\ generacion\ t$$

$$s(i,j): Valor\ deseado\ para\ individuo\ i\ en\ caso\ j$$

$$c(i,j): Valor\ obtenido\ por\ el\ individuo\ i\ en\ caso\ j$$

$$Nc: Número\ de\ casos$$

Ajuste Estandarizado. – Se usa para eliminar la dualidad entre problemas de maximización y minimización. Con este método si el problema a tratar es de minimización se usa el ajuste puro de manera directa y si el problema es de maximización, resta de una cota superior el valor del ajuste puro.

$$s(i,t) = \begin{cases} r(i,t) & minimización \\ r_{max} - r(i,t) & maximización \end{cases} \tag{3}$$

Ajuste Estrecho. – Se aplica una transformación al ajuste estandarizado de manera que los valores estarán en un intervalo entre 0 y 1. Siendo la bondad de un individuo mayor cuando su ajuste esté más cerca de 1. La fórmula del ajuste estrecho es la siguiente:

$$a(i,t) = \frac{1}{1+s(i,t)} \tag{4}$$

$$s(i,t): Ajuste\ estandarizado\ del\ individuo\ i\ en\ generación\ t$$

Ajuste Normalizado. – Este método a deferencia de los demás que solo hacen referencia a la bondad del individuo analizado, indica la bondad de una solución respecto a las demás dentro de la población. Siempre tomará valores en el intervalo [0,1], siendo los mejores individuos los que se encuentren más cerca de uno. Este método tiene la ventaja de indicar si una solución es mejor entre todas las demás obtenidas de la población.

$$n(i,t) = \frac{a(i,t)}{\sum_{k=1}^{M} a(k,t)} \tag{5}$$

$$M: Tamaño\ de\ la\ población$$

3.3.2 Selección de individuos. Luego de la evaluación de los individuos de la población se procede a asignar una probabilidad de supervivencia a cada uno de los miembros en base a la bondad que hayan presentado durante la evaluación. En este punto todos los miembros de la población tienen posibilidades de seguir adelante, pero evidentemente y como ocurre en la naturaleza, aquellos que estén mejor adaptados tendrán más oportunidades de subsistir. Sin embargo, en el proceso de selección también interviene un proceso de aleatoriedad en el cual se determina que individuos pasarán a la siguiente generación y cuáles no. Los métodos más comunes de selección son el de la ruleta y por torneo.

Selección por ruleta

El método de la ruleta consiste en elaborar una ruleta con tantas secciones como individuos en la población, haciendo que el área de cada sección sea proporcional a la bondad del individuo

al que representa. Por lo general la población se ordena en base a la bondad, ubicando así las porciones más grandes al inicio de la ruleta. Mediante un número aleatorio se selecciona uno de los individuos ubicados en la ruleta, en otras palabras, se elige de forma aleatoria una de las soluciones donde la probabilidad de selección depende de la calidad de esa solución.

Selección por torneo

"La idea principal de este método de selección consiste en escoger a los individuos genéticos en base a comparaciones directas entre sus genotipos", (Gestal et al., 2010). Existen dos versiones de este método de selección el determinístico y el probabilístico.

La versión probabilística requiere que se defina un parámetro p el cual será fijo y se usará a lo largo de todo el proceso evolutivo. Para seleccionar a un individuo se genera un número aleatorio y se lo compara con el parámetro p, si el número es mayor que p entonces se seleccionará al individuo más apto y al menos apto en el caso contrario.

En la variante determinística se escoge al azar un número de individuos y de entre ellos se selecciona al más apto, el cual pasa a la siguiente generación de la población. En este caso es importante variar la cantidad de individuos que participan para modificar la presión de la selección evitando así que los menos aptos tengan menos posibilidades de ser escogidos.

3.3.3 Cruce. Permite obtener nuevos miembros en una población, el cruce consiste en recombinar a los individuos para producir descendencia que será parte de la siguiente generación. Este operador es de gran importancia para la transición entre generaciones ya que comúnmente se trabaja con una tasa de cruce del 90%. Con el cruce se obtendrá una población con mayor variedad que la inicial. Este operador busca simular lo que ocurre en la naturaleza al obtener a un individuo como resultado de la combinación de sus progenitores. "La idea principal del cruce se basa en que, si se toman dos individuos correctamente adaptados al medio y se obtiene una descendencia que comparta genes de ambos, existe la posibilidad de que los

genes heredados sean precisamente los causantes de la bondad de los padres", (Gestal et al., 2010). Si se comparten las características buenas de los individuos se espera que la descendencia tenga una bondad mejor que la expresada por sus padres de manera individual.

Los métodos de cruce pueden operar utilizando una de las siguientes estrategias:

- Estrategia destructiva. – Luego de realizada la operación de cruce se insertará en la nueva generación a la descendencia, sin considerar si tienen o no mejor ajuste que el de sus padres, descartando automáticamente a los progenitores.
- Estrategia no destructiva. – Una vez realizada la operación de cruce se conservará en la nueva generación únicamente a la descendencia cuyo ajuste sea superior al de los padres. Si los padres tienen un ajuste superior al de su descendencia, son los padres los que pasan a formar parte de la nueva generación.

Las operaciones de cruce dependen de la forma en que estén representados los cromosomas, estos pueden no ser solamente binarios. Entre las representaciones más comunes para problemas de optimización combinatoria están aquellas con números enteros o permutaciones.

Los métodos de cruce más empleados se detallan a continuación.

Cruce de 1 punto. – Se seleccionan a dos individuos de la población, se cortan sus cromosomas en un punto seleccionado al azar generando así dos segmentos diferentes: la cabeza y la cola. Se intercambian las colas entre los individuos obteniendo así a los descendientes. De esta manera se consigue heredar información genética de los padres. Observar Figura 9 para comprender el resultado de este tipo de cruce.

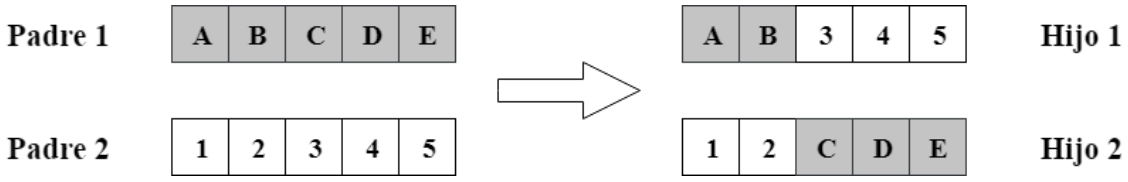

Figura 9. Cruce de 1 punto.

Fuente: Gestal, M., Rivero, D., Rabuñal, J., Dorado, J., & Pazos, A. (2010).

Cruce de 2 puntos. – En este caso se realizan dos cortes en los cromosomas de cada padre para obtener tres segmentos. La descendencia se obtiene combinando el segmento central de uno de los padres con los segmentos laterales del otro. El resultado de este método de cruce puede contemplarse con mayor claridad en la Figura 10.

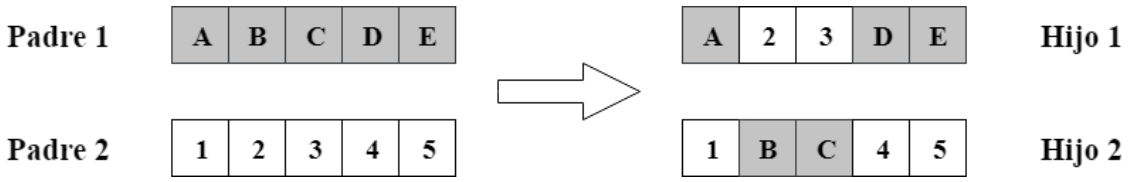

Figura 10. Cruce de 2 puntos.

Fuente: Gestal, M., Rivero, D., Rabuñal, J., Dorado, J., & Pazos, A. (2010).

Cruce uniforme. – Esta técnica permite que cada uno de los genes de la descendencia tenga la misma probabilidad de pertenecer a cualquiera de los padres. Esta técnica requiere de la generación de una máscara de cruce en base a valores binarios. En la parte superior de la Figura 11 puede encontrarse un ejemplo de máscara, el cual variará dependiendo del problema en el que se esté utilizando. Si el valor en la máscara es 1 el gen de la descendencia se tomará del primer padre y se el valor en la máscara es 0 se tomará del segundo padre. Para generar al segundo descendiente se intercambia la interpretación de los ceros y unos, o se invierte el orden de los padres.

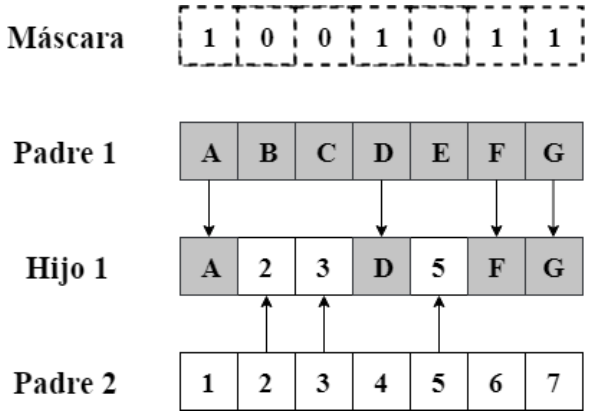

Figura 11. Cruce uniforme.

Fuente: Gestal, M., Rivero, D., Rabuñal, J., Dorado, J., & Pazos, A. (2010).

Cruce parcialmente mapeado (Partially Mapped Crossover PMX[3]). – Este método de cruce inicia por obtener un segmento de los cromosomas de cada padre cortando en los mismos dos puntos a cada uno de ellos. A continuación, se intercambian los segmentos seleccionados para obtener así las nuevas expresiones. Hasta este punto el método es igual al cruce de dos puntos; sin embargo, debido a la naturaleza combinatoria o de permutación de los cromosomas padres este método ejecuta pasos adicionales. En cada uno de los nuevos cromosomas determina la relación basada en mapeo en los segmentos seleccionados y a partir de esto procede a legalizar o verificar que los hijos producto del cruce son válidos. La figura 12 ilustra de mejor manera el procedimiento.

[3] PMX: Partially mapped Crossover

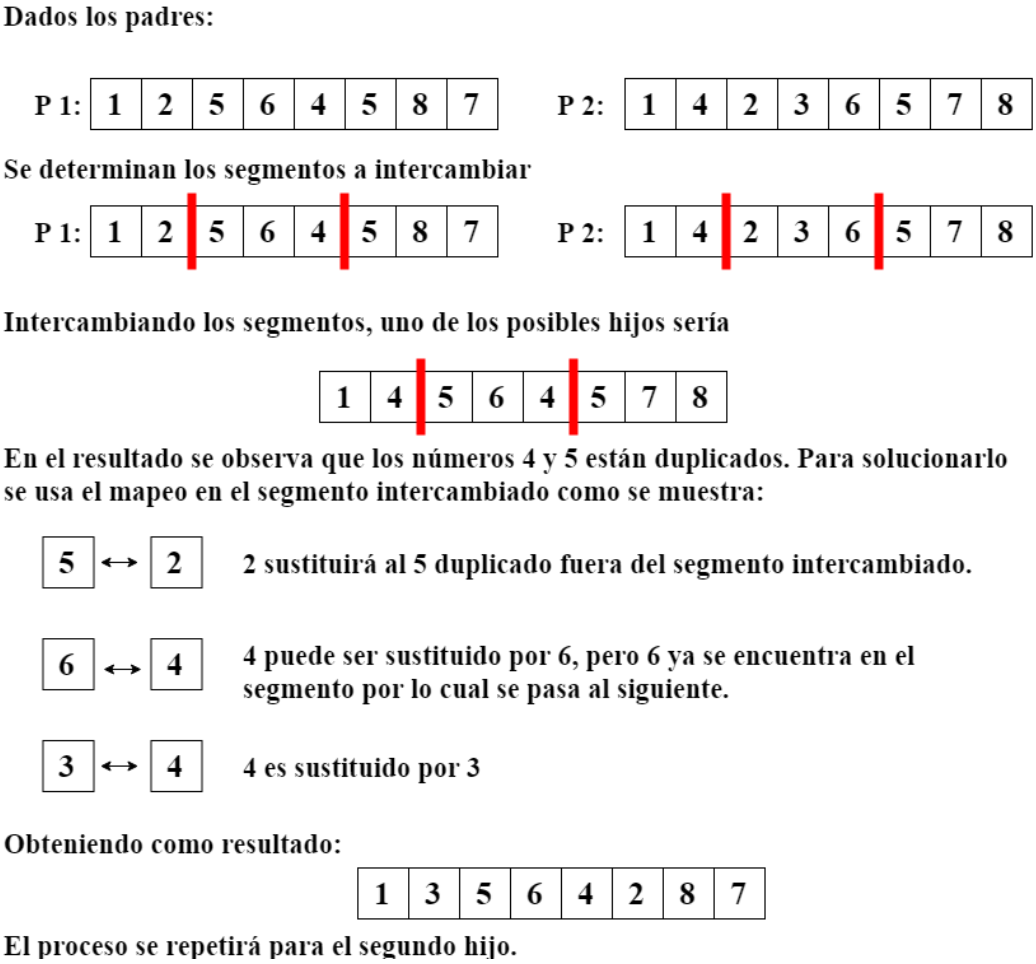

Figura 12. Ejemplo cruce PMX.

Fuente: Deep, K., & Mebrahtu, H. (2012).

3.3.4 Reemplazo. Este operador es utilizado cuando se trabaja con una única población en la cual se realizan inserciones y selecciones. Las inserciones en una población única pueden realizarse luego de haber eliminado previamente a otro integrante de esta. Los métodos de remplazo más comunes son:

Aleatorio. – De manera aleatorio se escoge el lugar en el cual un nuevo individuo será insertado.

Reemplazo de padres. – Luego del cruce la descendencia pasa a ocupar el lugar de los padres.

Reemplazo de similares. - Una vez obtenida la descendencia se analiza el ajuste de esta y se selecciona a un grupo de entre 6 y 10 individuos de la población con un ajuste similar. De manera aleatoria se reemplaza a los que se requiera.

Reemplazo de los peores. - Se toma a un determinado porcentaje de los peores en la población y de ellos se selecciona de manera aleatoria a los que serán reemplazados por la descendencia.

3.3.5 Copia. Es común conservar elementos de la población previa en una nueva, este proceso es conocido como copia y se considera como un tipo de reproducción asexual pues los individuos pasan a una nueva generación sin sufrir variaciones. La cantidad de individuos a los que se decide copiar en una nueva generación es muy reducida para prevenir una convergencia prematura de la población hacia estos individuos.

3.3.6 Elitismo. Es un caso particular de la copia y se emplea para asegurar que el proceso de búsqueda de la solución nunca de un paso atrás si hablamos de la calidad de la solución. Consiste en copiar a los mejores individuos de una generación en la siguiente. Este operador tiene una variante en la que se pasa los mejores individuos de una generación a otra solamente se estos siguen siendo mejores que los elementos que componen la nueva generación.

3.3.7 Mutación. "La mutación de un individuo provoca que alguno de sus genes, generalmente uno sólo, varíe su valor de forma aleatoria.", (Gestal et al., 2010). Aunque la probabilidad de la mutación en una población es muy baja, esta contribuye a la diversificación de la población y a abarcar una mayor cantidad de soluciones posibles; se usa con menor frecuencia que los otros operadores ya que en ocasiones sus resultados pueden ser peores a los obtenidos sin ella. Para imitar el comportamiento de la naturaleza, la forma más común de aplicar la mutación es hacerlo sobre los descendientes luego del cruce. De este modo se simula la alteración genética que ocurre durante la combinación de los genes de los progenitores.

La forma más común en que se aplica la mutación consiste en variar aleatoriamente un gen en un cromosoma, en el caso de la codificación binaria se hace negando un bit. También puede hacerse intercambiando aleatoriamente dos componentes de un cromosoma. En caso de codificaciones no binarias suele recurrirse a incrementar o decrementar una pequeña cantidad a un gen o multiplicar un gen por un valor aleatorio cercano a 1.

"Una de las principales ventajas de los algoritmos genéticos y que a primera vista puede parecer una desventaja es que los Algoritmos Genéticos no saben nada del problema que van a resolver. Como sus decisiones están basadas en la aleatoriedad, todos los caminos de búsqueda son posibles; en contraste, cualquier estrategia de resolución de problemas que dependa de un conocimiento previo, debe inevitablemente empezar descartando muchos caminos a priori, perdiendo así cualquier solución novedosa que pueda existir", (Álvarez, Jiménez, & Lanzagorta, 2012).

3.4 Funcionamiento de un algoritmo genético

Como se vio previamente para que un algoritmo funcione es necesario definir la función objetivo, el costo, las variables y determinar los operadores genéticos que usará el algoritmo. Con estos elementos definidos procede a generar la población inicial usando la codificación que se haya elegido. Definidos estos elementos se procede de la siguiente manera:

1. Definir el costo para los cromosomas de la población.
2. Se seleccionan los compañeros para el cruce.
3. Se realiza el cruce entre cromosomas.
4. Se aplica la mutación.
5. Revision de la convergencia.
6. Si no se cumple la condición de terminación se vuelve al paso 1, de lo contrario se termina la ejecución del algoritmo.

En la Figura 13 tenemos un diagrama de flujo del funcionamiento de un algoritmo genético.

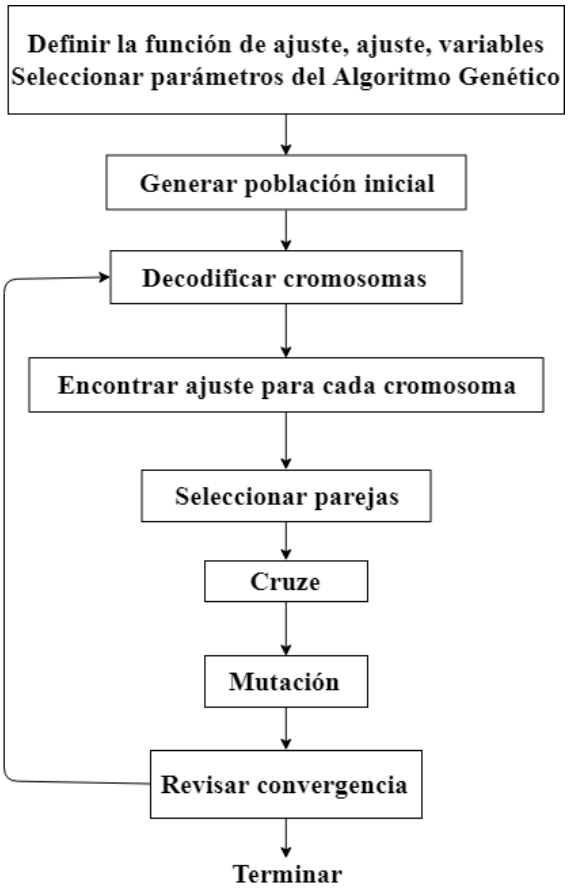

Figura 13. Diagrama de flujo Algoritmo Genético.

Fuente: Haupt, R. L., & Haupt, S. E. (2004).

3.5 Clasificación de algoritmos genéticos

Hemos visto que los algoritmos genéticos son considerados un proceso que busca encontrar la solución a problemas siguiendo una serie de pasos. Estos son capaces de alcanzar eficiencia y efectividad en la búsqueda de soluciones en el dominio del problema que tratan, debido a esto su aplicación se está extendiendo a campos fuera de la ingeniería y la investigación científica; debido a su amplia variedad de aplicación se han formulado diferentes aproximaciones en la formulación de algoritmos genéticos. Entre las más destacadas se encuentran:

3.5.1 Algoritmo genético simple. El mecanismo de funcionamiento del algoritmo genético simple involucra únicamente la copia de individuos en la población y el intercambio parcial de cromosomas entre individuos. Un algoritmo de este tipo y que produce buenos resultados en diversos problemas prácticos está compuesto por tres operaciones:

1. Selección de individuos
2. Cruce
3. Mutación

Este tipo de algoritmo es útil cuando el espacio de búsqueda de la solución es extenso, complejo o muy poco entendido. También usado cuando no existe un análisis matemático del problema o los métodos tradicionales no funcionan.

3.5.2 Algoritmo genético paralelo y distribuido. Un algoritmo genético paralelo consiste en la ejecución simultanea de varios algoritmos genéticos simples, esto se hace con el propósito de reducir los tiempos de ejecución asociados a los algoritmos simples durante la búsqueda de soluciones óptimas en problemas cuyo campo de búsqueda es realmente grande. Este tipo de algoritmos tienen mejor desempeño y escalabilidad. La paralelización depende de algunos elementos:

- Forma de cálculo del ajuste y aplicación de la mutación.
- Como se intercambian individuos se manejan múltiples poblaciones
- Como se aplica la selección en cada uno de los algoritmos paralelizados o en el conjunto de todos ellos.

La paralelización consiste en ejecutar múltiples copias de un mismo algoritmo genético simple, cada una en un elemento de procesamiento diferente. Cada copia del algoritmo iniciará con una población inicial distinta, avanzando y deteniéndose independientemente; el algoritmo

genético paralelo se detendrá cuando cada uno de los algoritmos simples que lo conforman se detenga.

3.5.3 Algoritmo genético hibrido. Este tipo de algoritmo se diseñó para combinar un operador de cruce ya existente con una heurística[4]; entre las heurísticas usadas más comúnmente están las enfocadas a la generación y mejora de la población inicial, la obtención de la descendencia por cruce o de manera aleatoria. Un algoritmo hibrido trabaja de la siguiente manera:

1. Inicialización de la población
    a. Se inicializa una parte de la población siguiendo la heurística seleccionada para dicho propósito.
    b. Se inicializa el resto de la población de manera aleatoria.
2. Se aplican las heurísticas de mejora de la población inicial.
3. Aplicación de las heurísticas de selección y cruce de los padres.
4. Se aplica al azar las heurísticas de mejora a la nueva población.
5. Se repiten los pasos 3 y 4 hasta alcanzar el número de iteraciones determinadas.

3.5.4 Algoritmo genético adaptativo. Este es un tipo de algoritmo genético en el cual sus parámetros tales como tamaño de la población, probabilidad de cruce o probabilidad de mutación varían mientras el algoritmo es ejecutado. La variación más simple de este algoritmo consiste en hacer que la tasa de mutación cambie de acuerdo con los cambios que se den en la población, mantener constante el tamaño de la población y la máxima tasa de mutación se escoge inicialmente. El proceso que sigue el algoritmo adaptativo es el siguiente:

1. Población inicial: Se obtiene de manera aleatoria.
2. Operadores genéticos: Se determinan las operaciones de cruce, mutación.

[4] Heurística: Método usado para incrementar el conocimiento. Estrategias que ayudan a resolver un problema.

3. Aplicación de la búsqueda local de manera iterativa de la mejor solución en la población.
4. Aplicación de las heurísticas de regulación de tasas de cruce y mutación.
5. Detención del algoritmo.

3.6 Análisis comparativo

Una vez descritos de manera breve los tipos de algoritmos genéticos más ampliamente conocidos, se había contemplado realizar un análisis comparativo entre estos; sin embargo, esto no es una tarea sencilla y es importante aclarar que si deseáramos obtener datos determinantes de una comparación acerca de su forma de empleo, complejidad de construcción, implementación o tiempo de ejecución, debería ser posible solucionar un determinado problema mediante la construcción de algoritmos genéticos de cada uno de los tipos descritos en la sección 5 de este capítulo. Siendo esta última opción no muy factible pues no todos los tipos de algoritmos genéticos pueden llegar a resolver un mismo problema.

La dificultad del análisis comparativo radica en que el diseño, construcción y desempeño de un algoritmo genético depende directamente del problema que este busca solucionar, los parámetros que intervienen, las variables que se consideran y muchos otros aspectos. Por lo cual, si se tomaran algoritmos de cada uno de los tipos pero que están planteados para resolver distintos problemas, los datos que se obtendrían de su comparación no serían concluyentes ya que no estarían operando sobre una base o con un objetivo común.

El tratar de comparar los distintos tipos de algoritmos genéticos en un panorama que no sea el planteado en primer párrafo de esta sección solo arrojaría resultados ambiguos y, por el contrario, tratar de diseñar y construir un algoritmo genético de cada tipo para resolver un mismo problema con la intensión de comparar el desempeño de estos llegaría a ser demasiado

extenso, complicado e incluso infructuoso pues podría darse el caso que no todos los tipos de algoritmo sean aplicables a un mismo problema.

Por último, en cuanto al análisis comparativo, cabría la posibilidad de tratar de realizarlo haciendo uso únicamente de las características y forma de ejecución de cada tipo de algoritmo genético. Por ejemplo, comparar la secuencia de ejecución del algoritmo genético simple con la del algoritmo genético hibrido. Viéndolo a breves rasgos podría considerarse que un análisis de este tipo llegaría a ser útil, pues podría concluirse en base al número de pasos que cada uno de estos algoritmos ejecuta para llegar a una posible solución, que el hibrido tiene una complejidad mayor que el simple; sin embargo esto continua siendo ambiguo ya que solamente al ligar la estructura de un algoritmo genético con el problema que este plantea resolver se puede llegar a observar cuan simple o compleja es la ejecución del algoritmo.

En resumen, no es tarea fácil llegar a comparar los distintos tipos de algoritmos genéticos por lo cual, si se desea usar este tipo de algoritmos para resolver un problema, lo más aconsejable es considerar o determinar los puntos clave del problema y lo que se espera obtener como solución, para con base en ellos procurar elegir el tipo de algoritmo genético que mejor se ajuste al tipo de problema que se pretende resolver.

Capítulo 4

Diseño del algoritmo genético

En este capítulo se describe de forma concreta el problema que se plantea resolver mediante un algoritmo genético, del mismo modo se describe al algoritmo y cada uno de los operadores de los que este se servirá para encontrar una solución óptima al problema planteado.

4.1 Planteamiento

El algoritmo genético que se construirá pretende resolver un problema de optimización del área usada en el proceso de corte de materiales rectangulares (Bidimensionales). El objetivo de este problema de optimización es:

> **Maximizar el área usada de cada una de las unidades de dimensiones estándar requeridas para satisfacer una orden de corte[5].**

Para verificar el cumplimiento de este objetivo se analizará la relación existente entre el área disponible en una unidad de tamaño estándar y la sumatoria de las áreas de todos los ítems obtenidos del corte de esta.

En este punto es importante aclarar que lo que se busca en este trabajo es plantear una solución al problema de corte de materiales haciendo uso de un algoritmo genético, no se pretende de ninguna manera proponer un nuevo tipo de algoritmo genético similar a los analizados en el capítulo 3, de ahí que en los capítulos siguientes no se buscará comparar este algoritmo con otro tipo de algoritmos genéticos sino con soluciones propuestas previamente al CSP.

[5] Orden de corte: listado de todos los ítems que se requiere obtener luego del corte de las unidades estándar.

4.1.1 Condiciones

- El algoritmo generará una solución para un conjunto de ítems cuyas áreas sumadas no excedan el área total de la unidad estándar.
- Los cortes aplicados serán de tipo guillotina.
- Los ítems pueden ser rotados 90 grados para definir su ubicación antes de ser cortados.
- El área usada para cortar los ítems tendrá una proporción entre largo y ancho que sea similar a la proporción entre las mismas dos dimensiones de la unidad estándar.
- El algoritmo generará la solución para una sola unidad estándar, por lo cual, si los ítems que se desea obtener requieren de más de una, el algoritmo se aplicará una vez por cada una de las unidades estándar requeridas.
- Se parte del supuesto que ninguno de los ítems en la orden de corte tiene dimensiones que excedan las dimensiones de la unidad estándar.
- Para el caso en que se requiera de varios ítems de las mismas dimensiones, cada ítem deberá ser especificado individualmente en la orden de corte.

4.2 Codificación

En el tercer capítulo se explicó la importancia de codificar los elementos que intervienen en el problema que será resuelto por el algoritmo genético, especialmente sus soluciones o posibles soluciones. Tomando en cuenta esto, y debido a la naturaleza del problema tratado, la codificación elegida para las soluciones del problema consiste en el uso combinado tipos de datos:

- Números enteros, los cuales se usan para identificar a cada uno de los ítems que se desea obtener.
- Letras, usadas para describir la configuración entre ítems.

4.2.1 Configuración de ítems. Para la construcción de las soluciones del problema, en lugar de optar por describir la ubicación de un ítem en la lámina estándar, se decidió describir la posición de un ítem respecto a otro y es a esta posición relativa a la que llamamos configuración. Se establecerán ocho posibles configuraciones entre un par de ítems a cortar:

- A: Los dos elementos se disponen uno junto a otro en forma horizontal.
- B: Los dos elementos se disponen de forma horizontal y el que se encuentra a la derecha es rotado 90°.
- C: Los dos elementos se disponen de forma horizontal y el que se encuentra a la izquierda es rotado 90°.
- D: Los dos elementos se disponen de forma horizontal y los dos son rotados 90°.
- E: Los dos elementos se disponen uno sobre otro en forma vertical.
- F: Los ítems se disponen de forma vertical uno sobre otro, siendo rotado 90° el ítem que se encuentra en la parte inferior.
- G: Los ítems se disponen de forma vertical uno sobre otro, siendo rotado 90° el ítem que se encuentra en la parte superior.
- H: Los ítems se disponen de forma vertical uno sobre otro, siendo rotados 90° los dos ítems.

Es importante explicar que una vez dos ítems de la lista de corte son agrupados por una de estas configuraciones, el menor rectángulo que contiene dicha configuración es considerado un nuevo ítem y pasará a reemplazar en la lista de corte a los dos que lo componen; por lo tanto, estará disponible para volver a combinarse con el resto de ítems.

En la Figura 14, se muestra de forma gráfica cada una de las configuraciones descritas previamente.

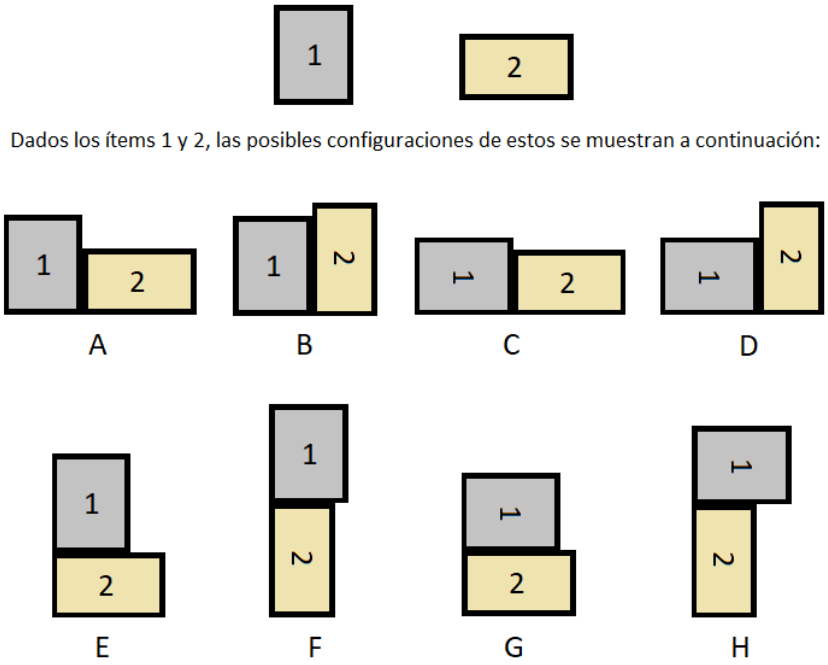

Figura 14. Configuraciones ítems.

4.2.2 Proceso de codificación. La codificación de cada uno de los individuos que conformarán las poblaciones usadas por el algoritmo genético requiere de dos pasos fundamentales:

1. De la lista de ítems que se desean cortar se toman estos en pares y se combinan usando las configuraciones descritas en la sección 4.2.1, este proceso debe repetirse iterativamente hasta que se obtenga un único ítem resultado de la combinación de todos los elementos en la lista.
2. A la expresión obtenida en el paso uno se procede a transformarla a notación postfija de manera que la disposición de los ítems en la solución quede expresada en función de los números que identifican a cada uno de los ítems y las condiciones de disposición sin ninguna ambigüedad. Esta transformación se realiza para que sea posible distinguir con facilidad que ítems se combinaron bajo que configuración sin que haya necesidad de introducir más símbolos a la codificación.

Para aclarar de mejor manera este procedimiento a continuación tenemos un ejemplo de su aplicación.

Sea la lista de corte L1, la cual contiene diez ítems cuyas áreas sumadas no exceden el área de la unidad estándar de la cual se planea, se desea obtener una posible solución aplicando el proceso de codificación.

$$L1 = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$$

Con el propósito de simplificar el ejemplo, la lista de corte contiene únicamente los identificadores de cada uno de los ítems a cortar.

Al realizar la primera iteración del paso número uno, suponiendo que el primer par de ítems a combinar es $\{2, 5\}$ y la configuración elegida es "C" el resultado de la lista de corte es el siguiente:

$$L1 = \{1, 3, 4, 6, 7, 8, 9, 10, (2\ C\ 5)\}$$

Para cuando se hayan aplicado todas las iteraciones requeridas, el estado final de la lista de corte contendrá un solo ítem:

$$L1 = \{\Big(\Big(\big(\big(\,1\ \mathrm{A}\ 7\,\big)\mathrm{C}\,\big(\,10\ \mathrm{B}\ 6\,\big)\big)\mathrm{E}\ 3\,\Big)\,\mathrm{D}\,\big(\,4\ \mathrm{H}\,\big(\,8\ \mathrm{F}\ 2\,\big)\big)\mathrm{A}\ 5\,\Big)\,\mathrm{C}\ 9\}$$

Una vez obtenida la expresión del ítem (super ítem) que contiene a todos los ítems a cortar puede decirse que ya se ha conseguido la codificación de una posible solución al problema, siendo esta la siguiente:

$$\Big(\Big(\big(\big(\,1\ \mathrm{A}\ 7\,\big)\mathrm{C}\,\big(\,10\ \mathrm{B}\ 6\,\big)\big)\mathrm{E}\ 3\,\Big)\,\mathrm{D}\,\big(\,4\ \mathrm{H}\,\big(\,8\ \mathrm{F}\ 2\,\big)\big)\mathrm{A}\ 5\,\Big)\,\mathrm{C}\ 9$$

Para propósitos del ejemplo se usan los paréntesis pues permiten dejar en claro de qué manera se combinaron los ítems en parejas usando las configuraciones de disposición, a pesar de que la expresión codificada real no los tiene. Sin embargo, para poder expresar la solución

de manera no ambigua y que el computador sea capaz de procesarla es necesario aplicar el segundo paso.

Aplicando la transformación a notación postfija del paso dos, la codificación de la posible solución se vería así:

1	7	A	10	6	B	C	3	E	4	8	2	F	H	5	A	D	9	C

4.3 Generación de población inicial

Los individuos que conforman la población inicial serán generados de manera aleatoria utilizando el listado de ítems que se desea obtener y las opciones de configuración descritas en la codificación. Se optó por obtener la población inicial de esta manera debido a que así evitamos agregar más restricciones a la disposición de los ítems al momento de ser cortados, esto facilitará el cruce y mutación de los individuos. Además, al hacerlo de manera aleatoria, se generarán todo tipo de posibles soluciones al problema y evitaremos que los individuos que componen la población inicial pertenezcan a un subconjunto del universo de todas las posibles soluciones.

4.4 Ajuste

Ya que el algoritmo tiene por objetivo maximizar el área usada en una unidad estándar y genera una solución para un conjunto de ítems cuyas áreas sumadas no exceden el área total de la unidad estándar, la función de ajuste se aplicará a cada uno de los individuos de la población, en lugar de aplicarlo a toda la población.

El valor de ajuste para cada individuo se obtendrá del cociente entre la sumatoria de las áreas de los ítems que se cortaran de la unidad estándar y el área del mínimo rectángulo que les contiene. Matemáticamente la función de ajuste se expresa:

$$f(s) = \frac{\sum_i l_i . a_i}{l_m . a_m} \quad (6)$$

Siendo:

s: Una posible solución del problema.

i: La representación de todos los ítems que se busca cortar de la unidad estándar.

$l_i \; y \; a_i$: Corresponden al largo y alto de cada ítem respectivamente.

$l_m \; y \; a_m$: Corresponden al largo y alto del menor rectángulo que contiene a la configuración de todos los ítems, es decir son las dimensiones del super ítem resultado de la codificación del individuo (solución) analizada.

Se optó por calcular el ajuste de esta manera y no usando directamente el área de la unidad estándar porque tanto este dato como las áreas de los ítems cortados son conocidos desde antes de ejecutar el algoritmo y si se calculará el ajuste con estos valores el resultado sería el mismo para cualquier posible solución.

4.4 Selección

La selección de los individuos de la población que serán cruzados se realizará mediante el método de la ruleta.

Luego de que se haya determinado el ajuste para cada uno de los individuos de la población, se procede a obtener la sumatoria de todos esos valores. Usando el resultado de la sumatoria se determina a que porcentaje del total corresponde el valor de ajuste de cada uno de los individuos, este porcentaje se convertirá en el valor que cada individuo tiene en la ruleta.

Para seleccionar a uno de los individuos de la población se genera aleatoriamente un valor entre 0 y 1. Tomando uno a uno los ítems, sumamos los valores de ruleta de cada uno de

estos, el ítem cuyo valor de ruleta haya permitido que el valor de la suma sea igual o mayor al número aleatorio será el ítem seleccionado.

4.5 Cruce

Ya que la codificación de los individuos involucra dos tipos de datos (ítems y configuraciones) la operación de cruce consistirá en una serie pasos, dentro de las cuales hay dos operaciones parciales de cruce.

En primer lugar, se seleccionan dos individuos los cuales serán los padres y se procede a descomponer a cada uno de ellos en dos subconjuntos, el primero contendrá a los ítems y el segundo las configuraciones de disposición, esto manteniendo el orden que tienen en la codificación.

Una vez descompuestos los padres se toman los subconjuntos de las configuraciones de disposición de cada uno de ellos y con estos se aplica el cruce de un punto, determinando de forma aleatoria el punto de cruce.

Para realizar el cruce entre los subconjuntos de los ítems se aplicará una variación del método de cruce parcialmente mapeado PMX, la principal razón de esto se debe a que al realizar el cruce entre ítems se tiene que verificar que los conjuntos resultantes no contengan ítems repetidos.

En este caso, al igual que con los subconjuntos de configuraciones, se realiza el cruce de un punto; una vez determinado el punto de cruce y se han intercambiado los segmentos se procede a verificar que los ítems en el nuevo segmento no se encuentren en el segmento que no se intercambió en cada uno de los subconjuntos. En caso de que se encuentren ítems repetidos se los debe reemplazar por los ítems que se perdieron en el segmento intercambiado.

Una vez realizados los procesos de cruce en los subconjuntos se construyen los individuos resultado del cruce usando los nuevos subconjuntos de ítems y configuraciones. Una vez construidos, se procede al cálculo de sus ajustes y solamente aquellos descendientes cuyo ajuste sea superior o igual que el de sus padres pasarán a formar parte de la siguiente generación.

4.6 Mutación

La mutación se aplicará únicamente a un porcentaje de individuos que pertenecen al grupo de descendientes resultado del cruce. A diferencia del proceso de cruce, la mutación es relativamente sencilla. El proceso consiste en tomar al azar un individuo del grupo de descendientes e intercambiar los valores de dos componentes de su codificación. Estas componentes también son tomadas de forma aleatoria tomando en consideración que no resulten ser la misma componente. En la Figura 15 tenemos un ejemplo del proceso de mutación.

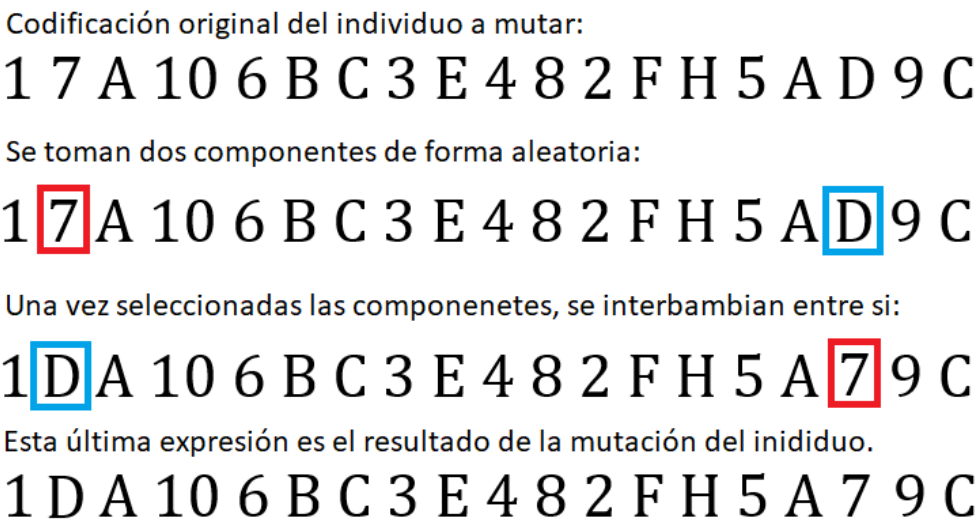

Figura 15. Ejemplo de mutación de un individuo.

Es importante que, una vez obtenido el resultado de la mutación este sea verificado para determinar si su estructura concuerda con la de una posible solución y respeta la estructura de la codificación.

4.7 Construcción de la siguiente generación

La creación de la siguiente generación de individuos hará uso de todos los operadores genéticos definidos en los puntos previos bajos ciertos parámetros definidos sobre la nueva población, los cuales se especifican a continuación:

- Cada nueva generación tendrá el mismo número de individuos que la población inicial.
- Como primer individuo de la nueva generación se agregará al individuo con el mejor ajuste de la generación previa.
- A cada nueva generación se le agregaran nuevos individuos obtenidos mediante codificación, estos no pueden superar el 20% del tamaño total de la generación. Este porcentaje se definirá como parámetro antes de iniciar la ejecución del algoritmo. Al agregar nuevos individuos que no dependen de la generación previa se puede evitar que las soluciones converjan prematuramente a un máximo local y se agrega diversidad a la nueva población.
- El resto de los individuos que componen a la nueva generación son obtenidos del cruce de los integrantes de la generación previa.
- Sobre los individuos obtenidos del cruce se aplicará mutación a un porcentaje de ellos el cual es definido como variable del algoritmo.

4.8 Funcionamiento del algoritmo

Una vez descritos los componentes fundamentales del algoritmo se procede a explicar la secuencia y condiciones bajo las cuales se ejecutarán para así lograr que el algoritmo genético

busque las soluciones del problema. En la figura 16 se puede observar cómo está diseñado el algoritmo y la secuencia en que se ejecutan los pasos que lo componen.

1. Como entradas del algoritmo se definen las dimensiones de la unidad estándar que su utilizará en el proceso de corte, el tamaño de la población inicial, el porcentaje de mutación que se aplicará a cada nueva generación, el mínimo valor de ajuste que se espera de la solución y por supuesto el listado de ítems que se obtendrán de la lámina estándar.
2. El algoritmo procede a ejecutar la función de generación de la población inicial usando el listado de ítems y el valor del tamaño de la población inicial. Este proceso a su vez utilizará la función de codificación para la obtención de cada uno de los individuos de la población.
3. Para cada uno de los individuos de la población inicial se calcula su ajuste.
4. Se determina el individuo con mayor ajuste de la población y se verifica si su ajuste cumple con el valor mínimo esperado.
5. Si la condición del mínimo ajuste esperado se satisface el algoritmo termina su ejecución. De no ser satisfecha esta condición, se continua con el paso 6.
6. Se determinan los valores de ruleta para cada individuo de la población.
7. Se determina la siguiente generación.
8. Se vuelve a ejecutar el proceso desde el paso 4.

Es importante aclarar que el algoritmo contará con un mecanismo que verificará si las soluciones han dejado de mejorar con el paso de las generaciones y de ser así detendrá su ejecución.

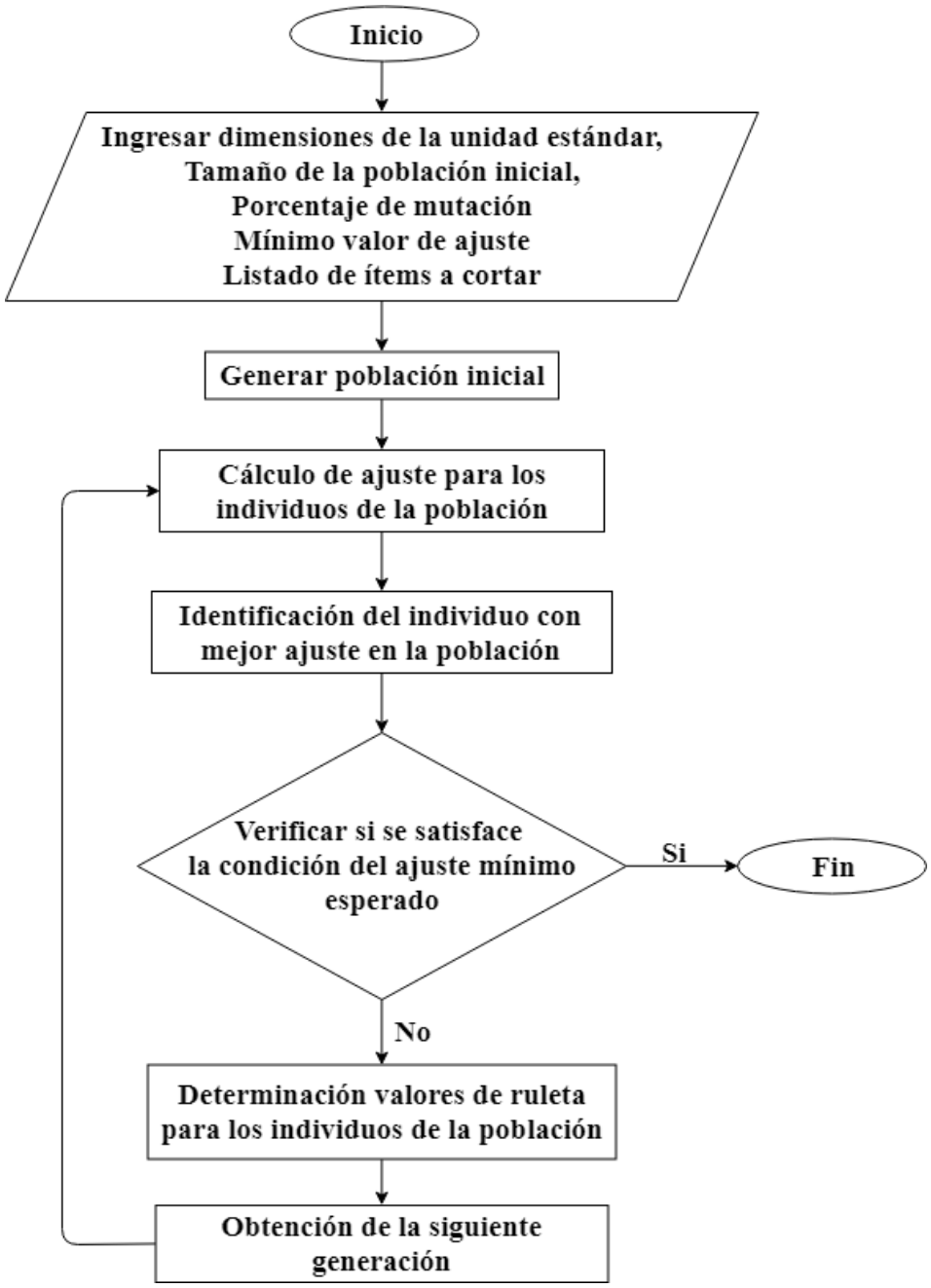

Figura 16. Diagrama de flujo del algoritmo diseñado.

En la sección 4.1.1 de este capítulo se explicó que el algoritmo está diseñado de manera que encuentre la solución para una lámina estándar, sin embargo, el problema a tratar involucra el obtener como solución la disposición de corte para N láminas estándar requeridas para satisfacer una orden de corte. Teniendo en cuenta esto, la solución integral propuesta consta de una función que genera subconjuntos de la lista de corte los cuales se obtienen en base a un parámetro que especifica que porcentaje del área de cada unidad estándar se espera poder

utilizar. De esta manera el algoritmo genético cuyo funcionamiento se describió en este punto se aplicará a cada uno de los subconjuntos.

Capítulo 5

Implementación

Este capítulo describe la estructura bajo la cual se implementaron los componentes clave del algoritmo genético y que fueron descritos en el capítulo 4; además, parte de una descripción la herramienta utilizada para la implementación del algoritmo.

5.1 Recursos usados

5.1.1 Python. "Python es un lenguaje de programación de alto nivel de propósito general.", (Kuhlman, 2009). Este lenguaje de programación fue ideado a finales de los 80 por Guido van Rossum, un programador holandés especializado en ciencias de la computación, quien liberó Python al público en 1991. Las versiones 2.0 y 3.0 se publicaron en los años 2000 y 2008 respectivamente, estas versiones siguen siendo mantenidas en desarrollo por la Fundación de Software Python, la cual fue dirigida por van Rossum hasta julio del 2018. Debido a que es un lenguaje de alto nivel, el programar con Python es sumamente fácil y el tiempo requerido para codificación es menor. Python es un lenguaje interpretado y compilado a la vez, lo cual hace que el tiempo requerido para la ejecución de programas creados con este lenguaje sean menores a los requeridos por lenguajes no interpretados.

"Python ofrece una estructura de construcción de software robusta, permite utilizar bloques de código anidados, clases, funciones, módulos y paquetes, además de objetos", (Kuhlman, 2009). Estas características junto a su sintaxis sencilla y su amplia biblioteca de herramientas permiten crear fácilmente aplicaciones lógicas para tareas grandes o pequeñas.

Python, tradicionalmente ha tenido un uso amplio como herramienta de scripting[6]. Pero en los últimos años el uso de Python se ha extendido de gran manera en los campos de

[6] Herramienta que permite escribir programas para tiempos de ejecución especiales y automatizar tareas.

Inteligencia artificial, Big Data[7] y Ciencia de datos[8]. Pues estos campos se convirtieron en áreas de trabajo con requerimientos de especialistas propios y han sido ellos quienes han explotado las características de este lenguaje y han contribuido a su desarrollo.

5.1.2 Spyder. "Spyder es un poderoso ambiente científico escrito en Python para Python; diseñado por y para científicos, ingenieros y analistas de datos.", (The Spyder Website Contributors, 2018). Este software es un entorno de desarrollo que ofrece una combinación avanzada de funcionalidades para edición, análisis y depuración mediante una ejecución interactiva, inspección profunda y muy buena visualización.

5.2 Implementación del algoritmo

La codificación del algoritmo genético está contenida en dos scripts[9], Clases.py y Algoritmo_Genetico.py; además, se cuenta con un tercer script Principal.py para la representación gráfica de las soluciones del algoritmo y su ejecución iterativa. En el anexo 1 puede encontrarse la extensión completa de los scripts.

El algoritmo trabajará haciendo uso de objetos de dos clases, *Item()* y *Solucion()*, estos objetos se encuentran definidos en el script Clases.py. Los objetos de tipo *Item()* se usan para representar a los ítems que conforman la lista de corte, cada uno de ellos tiene como atributos: *largo, alto, id* y *area*. Se optó por definir el área de cada ítem como un atributo y no como una función de la clase debido al número de veces que es requerido este dato a lo largo de la ejecución del algoritmo.

[7] Big Data: Término que describe grandes conjuntos de datos que pueden resultar demasiado complejos para el procesamiento tradicional de información.
[8] Ciencia de datos: Campo interdisciplinario de extracción de información y conocimiento mediante métodos científicos, procesos, algoritmos y sistemas.
[9] Script: Lista de comandos que son ejecutados por un determinado programa o motor

Los objetos de tipo *Solucion()* serán los individuos que conformen cada una de las generaciones usadas por el algoritmo, desde su población inicial. Los objetos de tipo *Solucion* tienen tres atributos fundamentales:

- *valor*: Contiene el resultado de la codificación de dicho individuo, este atributo consiste en una serie de listas anidadas las cuales tienen la siguiente estructura:

  $$[\,Item, Item, Configuración]$$

  En donde Item puede ser ocupado por un objeto de tipo *Item* o por otra lista con la misma estructura. Y el componente de configuración corresponderá a una de las letras usada para expresar la configuración de los ítems en la sección 3.2.1.
- *codificacion*: Contiene una expresión de tipo string de la codificación del individuo. El propósito de este segundo atributo es ser de utilidad para las operaciones de cruce y mutación, pues trabajar con esta expresión de tipo string es más sencillo que trabajar directamente con la codificación. Este atributo esta expresado en la forma descrita en la sección 4.2.2 del capítulo de diseño.
- *area_util*. – Contiene la sumatoria de las áreas de los ítems usados para generar esta solución.

Las funciones principales del script Clases.py son:

generar_solucion(). – Esta función recibe como parámetros la lista de ítems a cortar y las dimensiones de la unidad estándar que se utilizará. La función toma al azar uno de los ítems de la lista y usando los ítems restantes determina cuál de ellos tiene al menos una de sus dimensiones (ancho, alto) más cercana a una de las dimensiones del ítem tomado al azar y a este par de ítems los dispone en la configuración que menos área ocupe; este resultado se guarda en una lista auxiliar. Este proceso se repite hasta que no sea posible generar más pares de ítems, en este punto pasa a considerar la lista auxiliar como la lista de ítems y vuelve a ejecutar el

emparejamiento hasta que al final se obtenga un solo elemento (ítem) que conforme la lista de corte. Usando la función *codificar_solucion()* se obtiene la expresión en string de la codificación resultado del proceso de emparejamiento. Usando estos dos datos la función retorna un nuevo objeto de tipo *Solucion*.

codificar_solucion(). – Esta función recibe como parámetro la expresión codificada de una solución, y mediante una iteración recursiva en las listas anidadas toma únicamente el identificador de cada ítem y de las configuraciones para obtener en un string la codificación de la solución.

area_usada(). – Recibe como parámetros un objeto de tipo *Solucion* y se encarga de determinar las dimensiones del mínimo rectángulo que contiene la codificación de esta. Tomando la expresión codificada de un individuo, itera en las listas anidadas, tomando cada uno de los ítems y configuraciones que la conforman y calcula el área de cada una de estas configuraciones hasta llegar al cálculo del área del rectángulo más externo en la codificación, es decir aquel que contiene a todos los ítems.

Antes de continuar con la descripción de la implementación del algoritmo es importante aclarar que debido a la facilidad que proporciona Python para la implementación se crearon únicamente funciones, las cuales en conjunto conforman el algoritmo genético.

El script Algoritmo_Genetico.py contiene la implementación de los operadores genéticos y una serie de funciones auxiliares. Entre las más relevantes están:

ajuste(). – Recibe como parámetros un individuo de la población (*Solucion*) y las dimensiones de la unidad estándar; siguiendo lo especificado en el diseño, calcula la relación existente entre la sumatoria del área de los ítems a cortar de la lámina y el área usada por el rectángulo que contiene a la codificación del individuo. Para esto se calcula el cociente entre la propiedad

area_util del parámetro de tipo *Solucion* y el resultado de aplicar la función *area_usada()* al mismo parámetro.

seleccion_ruleta(). – Se aplica a una población del algoritmo, la función genera un número aleatorio entre 0 y 1 y a continuación mediante un bucle empieza a sumar los valores de ruleta de cada uno de los individuos de la población hasta que el resultado de la suma sea igual o superior al valor obtenido aleatoriamente. El ítem cuyo valor de ruleta haga que se cumpla la condición del bucle será el ítem seleccionado.

cruce(). – Usando como parámetros las codificaciones expresadas en string de los dos individuos que se desea cruzar (atributo *codificacion*), la función procede a separar estas expresiones para obtener un listado de ítems y un listado de configuraciones de cada uno de ellos. Para el caso de las configuraciones, se toma la misma expresión de la codificación y se reemplazan los identificadores de ítems por una 'X'. Una vez separados, se determina el punto de corte para cada una de las listas, tanto de ítems como de configuraciones. En el caso de los ítems, se debe realizar el intercambio y adicionalmente la función verifica que como resultado del intercambio no se repitan los ítems en las nuevas listas resultado. Para el caso de las configuraciones se procede a intercambiar las partes de cada uno de los padres para obtener los operadores de configuración de cada descendiente. Luego de realizado el intercambio, usando el listado de ítems para cada uno de los descendientes se procede a intercambiar las 'X' en las expresiones que contiene las configuraciones por cada uno de los ítems y como resultado la función de cruce devuelve las codificaciones de los descendientes, expresadas como un string.

codificar_descendencia(). – Esta función trabaja con las expresiones string que resultan del cruce y con la lista de items que se cortarán. La función toma uno a uno los elementos que componen el string y que están delimitados por un separador, determina si ese segmento de la expresión corresponde a un caracter usado para representar una configuración de ítems o si

corresponde al identificador de un ítem; dependiendo de esto si corresponde a una configuración se procederá a guardar en una lista temporal que contendrá solamente a estos caracteres y en el caso contrario usando el identificador se procede a buscar al ítem y al resultado de dicha búsqueda se lo almacenará en una lista temporal de *Items*. Una vez identificados todos los elementos que componen la expresión string se procede a construir, de manera similar a la usada en la función *generar_solucion()*, la expresión codificada de una solución al problema usando la estructura de listas anidadas; esto usando únicamente las listas auxiliares de configuraciones e ítems. El resultado de la función es la expresión codificada y lista para usarse del descendiente resultado del cruce.

mutacion(). – Esta función toma como parámetro un individuo de la población y de manera aleatoria selecciona dos elementos que la componen y asegurándose que por efecto de la selecciona aleatoria no se haya seleccionado dos veces el mismo elemento los intercambia.

nueva_generacion(). – Esta función hace uso de las funciones descritas previamente y es una de las más importantes para el funcionamiento del algoritmo. Usa como parámetros:

- La población actual
- El tamaño de la generación
- La lista de ítems con los cuales se generan las soluciones
- Las dimensiones de la unidad estándar
- Porcentaje de mutación.
- Porcentaje de nuevos individuos.

La función inicia por calcular los valores de ruleta de cada uno de los individuos que componen la población actual. Una vez hecho esto identifica al individuo con mejor ajuste de la población actual y lo agrega como el primer individuo de la siguiente generación. Usando el porcentaje

de agregación y el tamaño de la población, determina cuantos nuevos individuos se deben generar y los agrega a la siguiente generación.

A continuación, hasta que el tamaño de la siguiente generación alcance el valor esperado realiza el proceso de selección y cruce sobre los individuos de la población actual, en este paso únicamente se agregaran a la siguiente población a los hijos resultado del cruce cuyo ajuste supere o alcance el ajuste de sus progenitores. Una vez definidos todos los descendientes requeridos, se aplica la función de mutación a estos individuos de acuerdo con el parámetro de porcentaje de mutación que la función tomó como parámetro. Como resultado la función retorna el conjunto de individuos que conforman la siguiente generación.

algoritmo_genetico(). – Es la última función del script Algoritmo_Genetico.py y como su nombre lo indica esta es la función que ejecuta al algoritmo genético como tal. Esta función recibe como parámetros los ítems que se cortarán de la lámina, las dimensiones de la unidad estándar y el tamaño con el que contarán cada una de sus generaciones. Esta empieza por llamar a la función usada para generar la población inicial del algoritmo y calcula el ajuste de cada uno de los individuos de la población inicial; adicionalmente la función define un parámetro de la mínima variación del ajuste entre generaciones (*parametro_v*).

En este punto la función ejecuta un bucle el cual detendrá su ejecución una vez la variación de ajuste entre generaciones sea menor o igual al a *parametro_v*. Dentro del bucle se genera una nueva población y se determina al individuo con máximo ajuste en ella, la fusión calcula la variación del ajuste respecto a la población previa y si se satisface la condición de salida del bucle, el ultimo individuo del cual se determinó el ajuste será la solución más óptima al problema.

Para visualizar de mejor manera funcionamiento del algoritmo, este se describe en el diagrama de bloques de la figura 17.

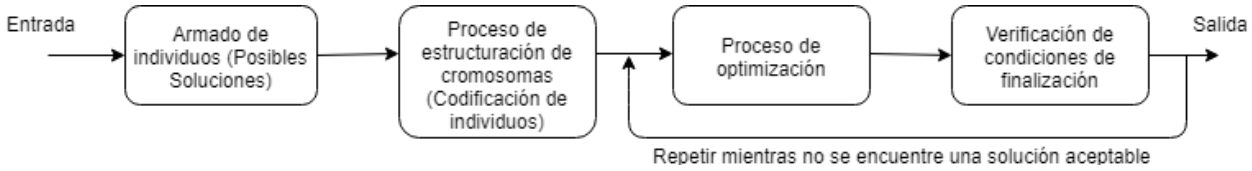

Figura 17. Diagrama de bloques algoritmo genético.

Los resultados que la función *algoritmo_genetico()* arroje serán usados por el script Principal.py.

El script Principal.py consiste en las funciones requeridas para hacer que el algoritmo genético se ejecute cuantas veces sea necesario para satisfacer la orden de corte. El script está compuesto por tres funciones:

grupo_items(). – Esta función se encarga de dividir el listado de ítems que conforman la orden de corte en grupos de ítems que satisfacen el parámetro del porcentaje del área de la unidad estándar que se espera utilizar. Para esto recibe como parámetros la orden de corte, las dimensiones de la unidad estándar y el parámetro de porcentaje de la unidad estándar.

main(). – Esta función es la encargada de ejecutar iterativamente el algoritmo genético hasta que la orden de corte sea satisfecha, debido a esto, toma como parámetros los mismos que el algoritmo genético. La única diferencia es que en lugar de tomar la lista de ítems que se cortarán de una lámina toma la lista de todos los ítems que requieren ser cortados (orden de corte).

Para cada grupo de ítems obtenidos mediante la función *grupo_items()*, la función *main()* aplica el algoritmo genético y retorna la mejor solución junto con la representación gráfica de esta, obtenida mediante la función *imprimir()*.

imprimir(). – Esta función es usada para generar una representación gráfica de la solución obtenida del algoritmo genético. Toma como parámetros a la lista de ítems cortados de la lámina estándar y la solución dada por el algoritmo.

La función inicia por generar imágenes de cada uno de los ítems y las guarda en una lista. Lo que hace es generar una imagen rectangular de color aleatorio basada en las dimensiones del ítem al que representará. Estas imágenes las genera haciendo uso de la librería de imágenes de Python llamada PIL. A continuación usando la misma lógica que la función *area_usada()* del script Clases.py, haciendo uso de la lista de imágenes de los ítems procede a construir una imagen que contiene a todos los ítems cortados en sus configuraciones respectivas y dentro de la representación gráfica de la unidad estándar.

La figura 18 se muestra como ejemplo de lo que la función *imprimir()* obtiene como resultado, es este caso cada uno de los recuadros de color con etiquetas dentro de ellos representan a un ítem, el área de color blanco representa al residuo del rectángulo que contiene a la solución y el área amarilla representa al residuo de la unidad estándar.

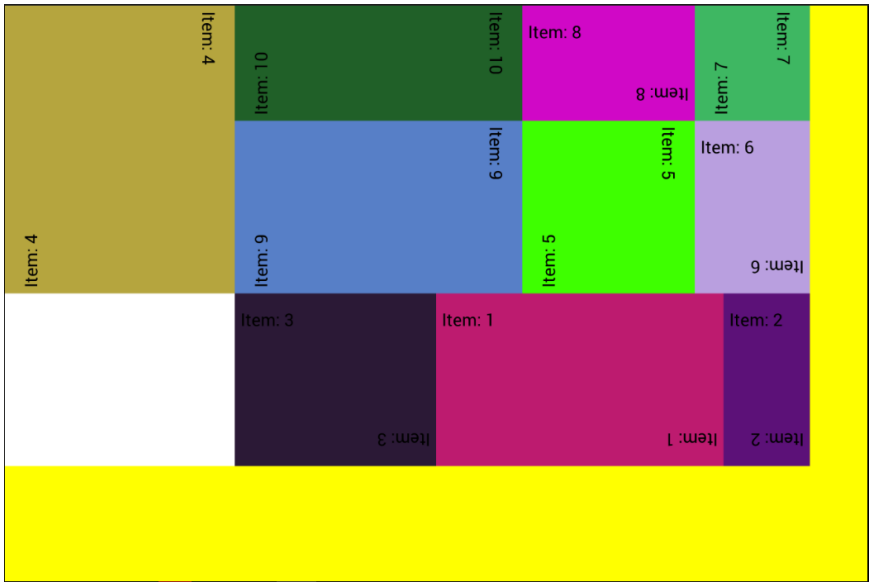

Figura 18. Representación gráfica de soluciones.

5.3 Implementación de la aplicación

La implementación de la interfaz se realizó haciendo uso íntegramente de Tkinter. Tkinter es el módulo estándar de interfaz de Python para Tk, el kit de herramientas de interfaz gráfica de usuario (GUI[10]) para el desarrollo de aplicaciones de escritorio.

La interfaz se implementó de manera sencilla debido a que su propósito es definir los parámetros y mostrar los resultados de la ejecución del algoritmo. Consta de una sola ventana en la cual se ingresan los parámetros tales como: dimensione de la unidad estándar usada, tamaño de las poblaciones, porcentaje de mutación y área usada, etc. También cuenta con un área en la cual se ingresarán uno a uno los ítems a cortar o se cargarán desde un archivo CSV[11] para facilitar este proceso. La ventana tiene además dos áreas de texto en las cuales se visualizará la información de resultados y el listado (orden de corte) con el que está trabajando el algoritmo. En la figura 20 se pueden apreciar los elementos que componen la ventana principal y la disposición de estos. El diagrama de bloques descrito en la figura 19 muestra el funcionamiento de la interfaz.

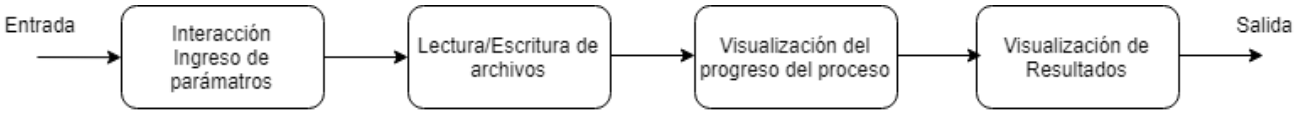

Figura 19. Diagrama de bloques interfaz gráfica

[10] GUI: Graphic User Interface, interfaz gráfica de usuario.
[11] CSV: Comma-separated values. Es un tipo de documento de formato abierto, usado para representar datos en forma de tabla.

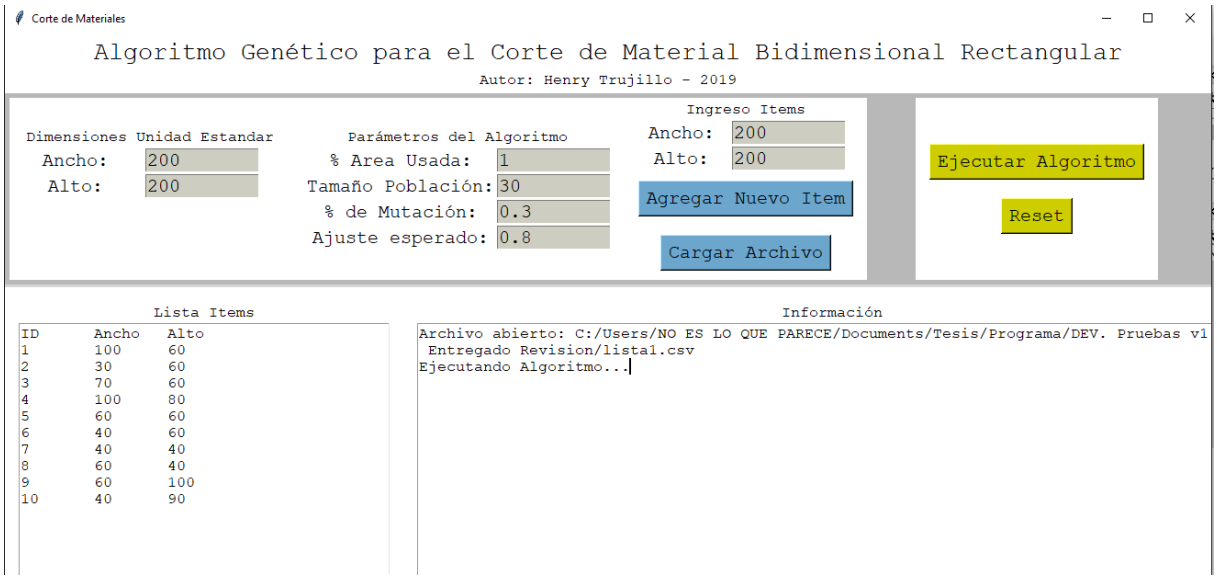

Figura 20. Ventana Principal.

El código fuente de la interfaz gráfica se encuentra en el Anexo 3.

Capítulo 6

Pruebas

Este capítulo describe el procedimiento adoptado para realizar las pruebas del funcionamiento del algoritmo una vez terminada la implementación tanto de este como de la interfaz que permite ingresar los parámetros y observar los resultados.

6.1 Objetivo de las pruebas

La finalidad de las pruebas es determinar que el algoritmo es capaz de generar resultados que puedan ser considerados útiles para la resolución del problema y cumple con su objetivo de optimización.

Por otra parte, con las pruebas se busca determinar el comportamiento del algoritmo dependiendo de las variaciones de los distintos parámetros de los cuales depende y también obtener información sobre el tiempo que le toma al algoritmo encontrar una solución al problema.

6.2 Estructura de las pruebas

Cada una de las pruebas consistirá en fijar las dimensiones de la unidad estándar que se utilizará y tomando una orden de corte aplicar el algoritmo para determinar si este es capaz de encontrar una solución óptima. El algoritmo se ejecutará varias veces en cada prueba y en cada una de las ejecuciones se cambiará uno de los parámetros que intervienen en el proceso de búsqueda de la solución. Para así observar el comportamiento de las soluciones obtenidas con el algoritmo.

Para la ejecución de las pruebas se usarán dos conjuntos diferentes de ítems (ordenes de corte) y para poder observar de mejor manera los cambios ocurridos en la ejecución del

algoritmo al modificar alguno de sus parámetros, se optó por usar en todas las pruebas la misma unidad estándar cuyas dimensiones son:

Ancho: 200 Largo: 200

Las ordenes de corte utilizadas se encuentran descritas en detalle en las tablas 1 y 2. Además, la información correspondiente a la configuración de parámetros para cada una de las pruebas está contenida en la tabla 3.

Tabla 1: Orden de corte 1.

ID	Ancho (x)	Alto (y)
1	40	50
2	80.5	50
3	40	190
4	30	50
5	40	80
6	40	30
7	130	30
8	40	110
9	70	90
10	80	40

Tabla 2: Orden de corte 2.

ID	Ancho (x)	Alto (y)
1	40	50
2	80.5	50
3	40	190
4	30	50
5	40	80
6	40	30
7	130	30
8	40	110
9	70	90
10	80	40

Lo que las tablas 1 y 2 contienen son dos listas cada una de 10 ítems, los cuales se desean obtener mediante el proceso de corte de unidades estándar. Cada uno de los ítems en las tablas cuenta con un identificador el cual ayuda a reconocerlo durante el proceso de ejecución del algoritmo, además están las dimensiones de ancho y altura para cada uno de los componentes de la orden de corte. Es importante decir que previa ejecución de las pruebas se conoce al menos una solución para obtener cada orden de corte usando una lámina estándar.

Tabla 3: Configuración de parámetros para las pruebas

No. Prueba	Orden de corte	Ancho unidad estándar	Largo unidad estándar	Tamaño población	% Área usada	% Mutación	Ajuste esperado
1	1	200	200	30	1	0.1	0.8
2	1	200	200	30	1	0.2	0.8
3	1	200	200	50	1	0.1	0.8
4	2	200	200	30	1	0.1	0.8
5	2	200	200	30	1	0.2	0.8
6	2	200	200	50	1	0.1	0.8

La Tabla 3 contiene un listado con los parámetros que se usarán en cada una de las pruebas. Como se aclaró previamente, todas las pruebas se realizarán haciendo uso de las mismas dimensiones para la unidad estándar. Las poblaciones estarán compuestas por 30 y 50 individuos ya que en pruebas previas se observó que con estos valores se obtenían buenos resultados. Ya que se conoce que existe al menos una solución que permite obtener todos los ítems de la orden usando solo una unidad estándar el porcentaje de área usada se define como 100% para todos los casos. Se utilizará un porcentaje de mutación de 10% y 20% para evitar alterar en exceso las soluciones obtenidas por el cruce. Por último, el ajuste esperado se define como 80% pues se parte del supuesto de que se encontrará por lo menos una solución que resuelve el problema con una unidad estándar.

Los resultados arrojados como resultado de la ejecución del algoritmo fueron:

6.3 Prueba 1

Los resultados obtenidos una vez ejecutada la prueba número 1 muestran que el algoritmo tuvo que ejecutarse quince veces con los parámetros establecidos para esta prueba (Ver Tabla 3). La solución se encontró en la décimo primera generación con un ajuste superior al 97% (Ver Figura 21).

```
Unidad estandar No. 1
El algoritmo se ejecutó 15 veces
Fitnes: 0.9796587926509186
Ancho: 200.0 Alto: 190.5 Area: 38100.0
Generacion No.11
Area Items: 37325.0
El tiempo de ejecucion fue:75.69105935096741
```

Figura 21. Resultados prueba No. 1

Las figuras 22 y 23 muestran la distribución del ajuste de los individuos de la primera y última generación respectivamente, siendo la generación final aquella en la que se encontró la solución para la prueba 1. Puede observarse como el ajuste de los individuos es muy variado en la primera generación y tiene un promedio cercano al 60% (Ver Figura 22).

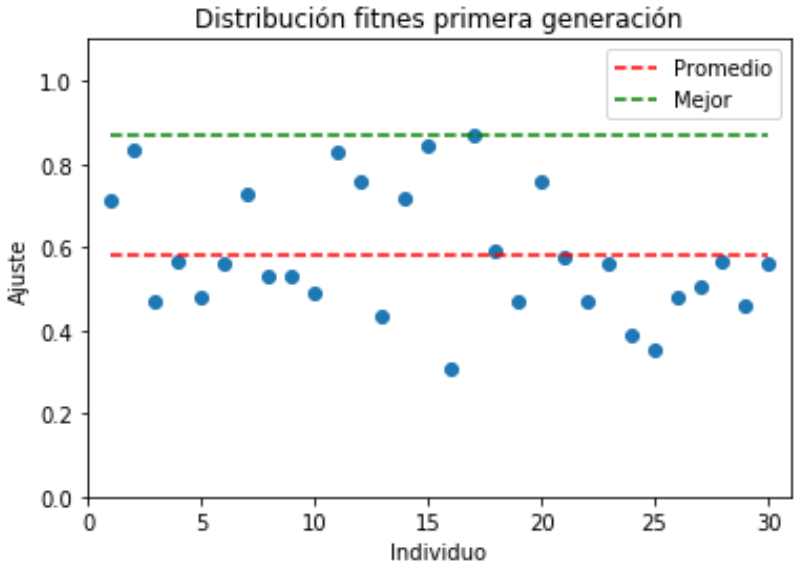

Figura 22. Distribución de ajuste de los individuos de la población inicial de la prueba 1.

Por el contrario, en la generación en que se encontró la solución óptima para la prueba 1, el promedio es superior al 70% y se puede observar claramente como la distribución de los valores de ajuste tiende a ser más compacta (Ver figura 22).

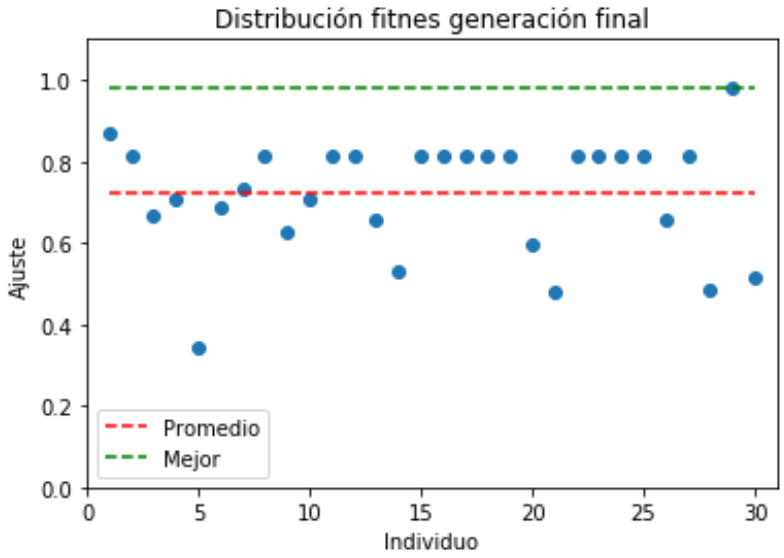

Figura 23. Distribución de ajuste de los individuos de la población en la que se encontró la solución óptima de la prueba 1.

La mejora de las posibles soluciones para la prueba 1 puede evidenciarse con el incremento del ajuste promedio de cada una de las generaciones obtenidas por el algoritmo, este paso de 60% a un 70% aproximadamente, esto con variaciones paulatinas (Ver figura 24).

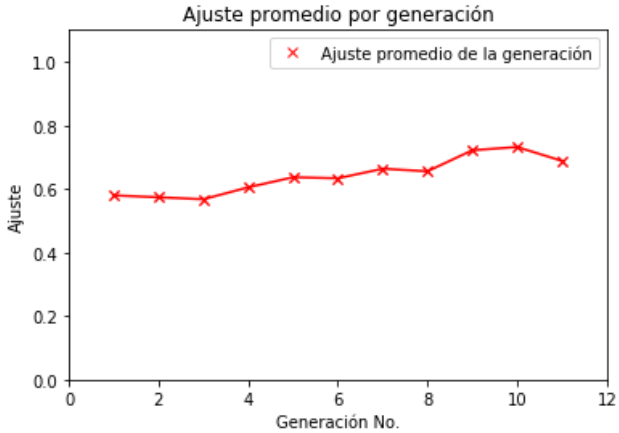

Figura 24. Evolución del ajuste promedio de cada generación de prueba 1.

El valor del ajuste para la mejore solución de cada una de las generaciones obtenidas en la prueba 1 tiene un comportamiento acorde con el visto en el ajuste promedio y este puede evidenciarse en la Figura 25.

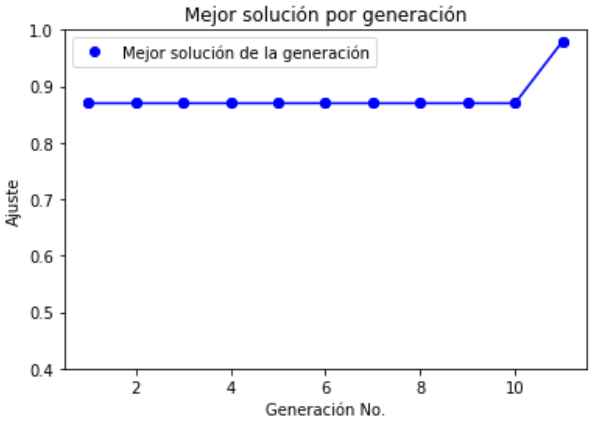

Figura 25. Evolución del mejor individuo de cada generación de prueba 1.

La solución óptima obtenida de la prueba 1 solo necesita de una unidad estándar para satisfacer la orden de corte requerida (Ver Figura 21). La manera en que se debe proceder a cortar cada uno de los ítems de la orden de corte se encuentra representada gráficamente en la Figura 26.

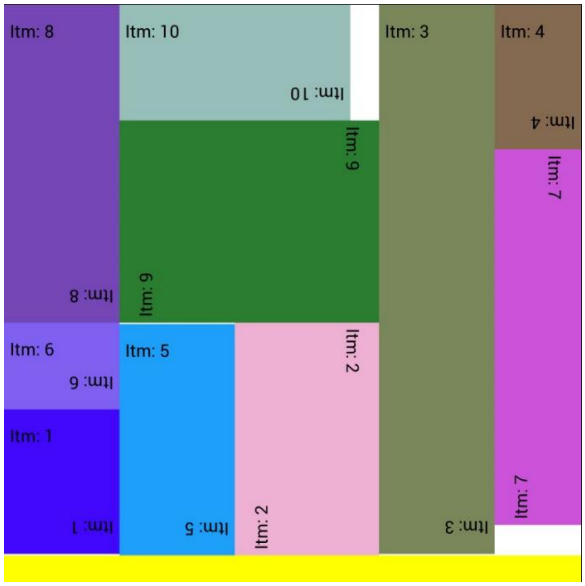

Figura 26. Representación gráfica de la solución obtenida en la prueba 1.

6.4 Prueba 2

Los resultados obtenidos una vez ejecutada la prueba número 2 muestran que el algoritmo tuvo que ejecutarse cuatro veces con los parámetros establecidos para esta prueba (Ver Tabla 3). La solución se encontró en la décimo cuarta generación con un ajuste superior al 93% (Ver Figura 27).

```
Unidad estandar No. 1
El algoritmo se ejecutó 4 veces
Fitnes: 0.933125
Ancho: 200.0 Alto: 200.0 Area: 40000.0
Generacion No.14
Area Items: 37325.0
El tiempo de ejecucion fue:31.97622513771057
```

Figura 27. Resultados prueba No. 2

Las figuras 28 y 29 muestran la distribución del ajuste de los individuos de la primera y última generación respectivamente, siendo la generación final aquella en la que se encontró la solución para la prueba 2. Puede observarse como el ajuste de los individuos es muy variado en la primera generación y tiene un promedio cercano al 50% (Ver Figura 28).

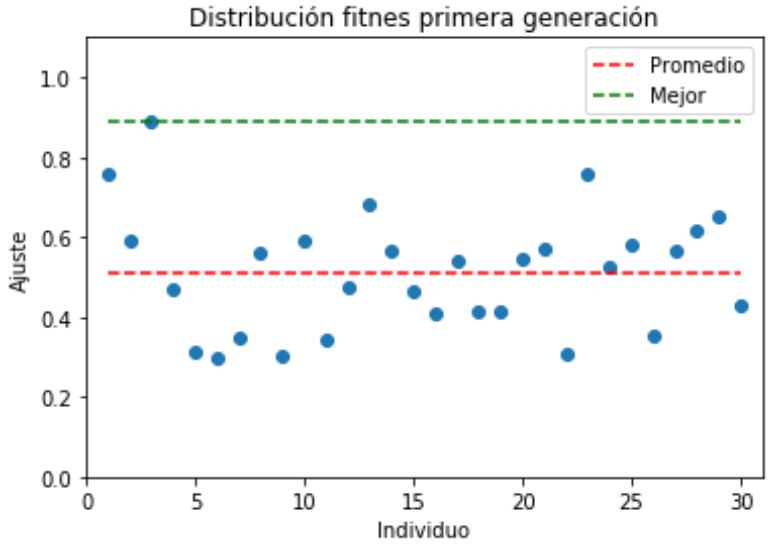

Figura 28. Distribución de ajuste de los individuos de la población inicial de la prueba 2.

Por el contrario, en la generación en que se encontró la solución óptima para la prueba 2, el promedio es superior al 70% y se puede observar claramente como la distribución de los valores de ajuste tiende a ser más compacta (Ver figura 29).

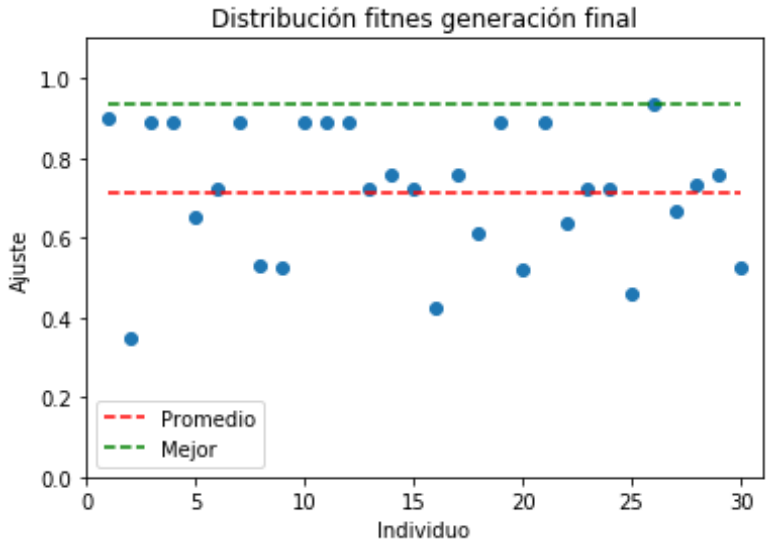

Figura 29. Distribución de ajuste de los individuos de la población en la que se encontró la solución óptima de la prueba 2.

La mejora de las posibles soluciones para la prueba 2 puede evidenciarse con el incremento del ajuste promedio de cada una de las generaciones obtenidas por el algoritmo, este paso de 50% a un 70% aproximadamente, esto con variaciones paulatinas (Ver figura 30).

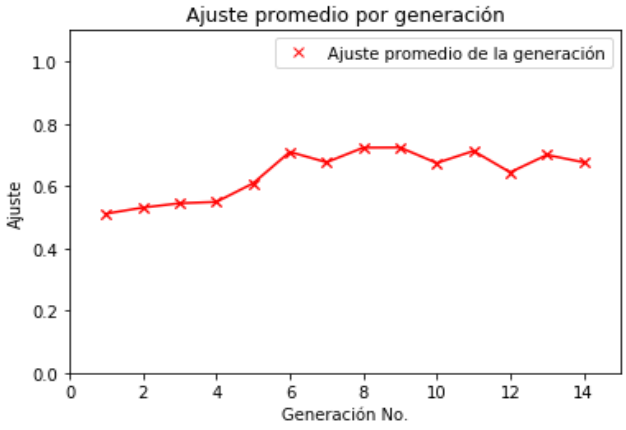

Figura 30. Evolución del ajuste promedio de cada generación de prueba 2.

El valor del ajuste para la mejor solución de cada una de las generaciones obtenidas en la prueba 2 tiene un comportamiento acorde con el visto en el ajuste promedio y este puede evidenciarse en la Figura 31, aunque el incremento en el ajuste obtenido es mínimo, es evidente que existe mejora.

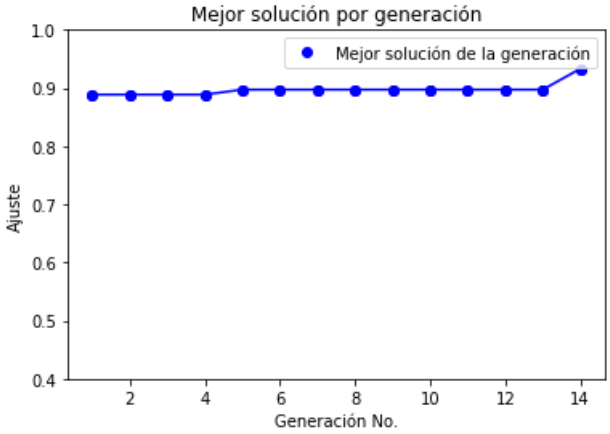

Figura 31. Evolución del mejor individuo de cada generación de prueba 2.

La solución óptima obtenida de la prueba 2 solo necesita de una unidad estándar para satisfacer la orden de corte requerida (Ver Figura 27). La manera en que se debe proceder a cortar cada uno de los ítems de la orden de corte se encuentra representada gráficamente en la Figura 32.

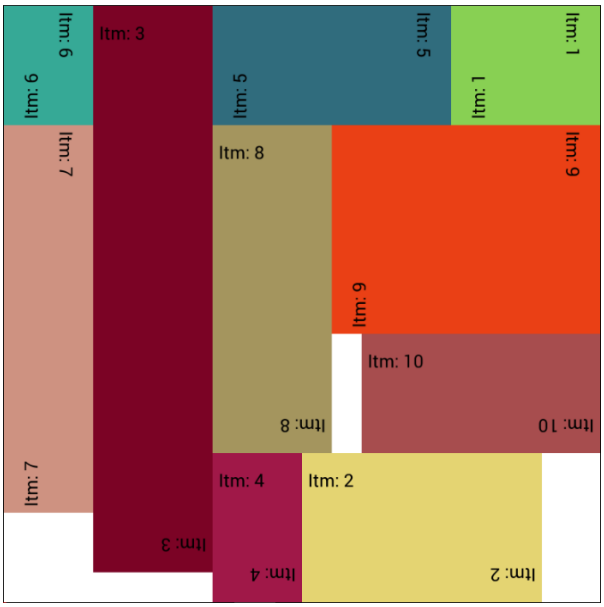

Figura 32. Representación gráfica de la solución obtenida en la prueba 2.

6.5 Prueba 3

Los resultados obtenidos una vez ejecutada la prueba número 3 muestran que el algoritmo tuvo que ejecutarse tres veces con los parámetros establecidos para esta prueba (Ver Tabla 3). La solución se encontró en la décima generación con un ajuste superior al 93% (Ver Figura 33).

```
Unidad estandar No. 1
El algoritmo se ejecutó 3 veces
Fitnes: 0.933125
Ancho: 200.0 Alto: 200.0 Area: 40000.0
Generacion No.10
Area Items: 37325.0
El tiempo de ejecucion fue:26.513764142990112
```

Figura 33. Resultados prueba No. 3

Las figuras 34 y 35 muestran la distribución del ajuste de los individuos de la primera y última generación respectivamente, siendo la generación final aquella en la que se encontró la solución para la prueba 3. Puede observarse como el ajuste de los individuos es muy variado en la primera generación y tiene un promedio cercano al 60% (Ver Figura 34).

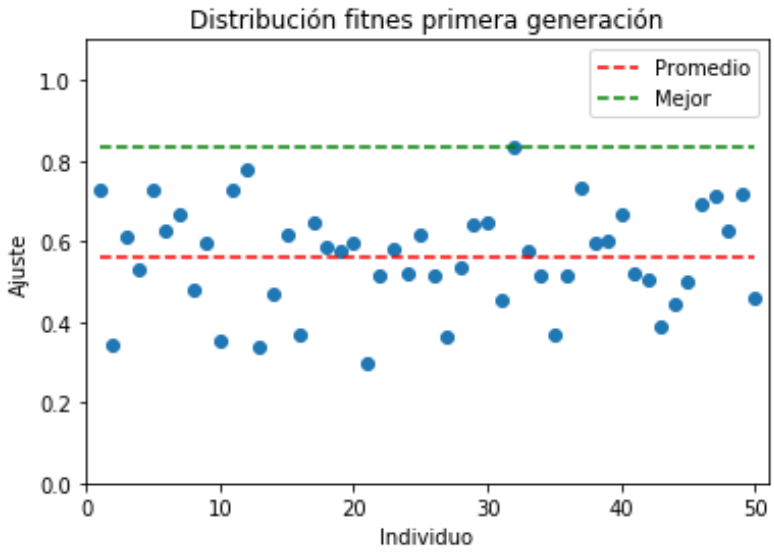

Figura 34. Distribución de ajuste de los individuos de la población inicial de la prueba 3.

Por el contrario, en la generación en que se encontró la solución óptima para la prueba 3, el promedio es superior al 70% y se puede observar claramente como el ajuste de gran parte de los individuos que la conforman tiende a estar en o sobre el promedio, en el caso de esta prueba se observa que una gran parte de la población tiene un ajuste similar (Ver figura 35).

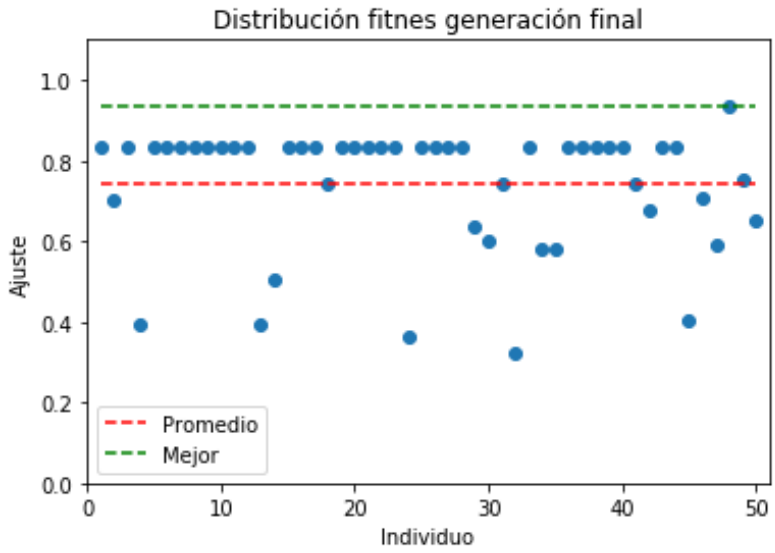

Figura 35. Distribución de ajuste de los individuos de la población en la que se encontró la solución óptima de la prueba 3.

La mejora de las posibles soluciones para la prueba 3 puede evidenciarse con el incremento del ajuste promedio de cada una de las generaciones obtenidas por el algoritmo, este paso de 60% a un 70% aproximadamente, esto con variaciones paulatinas (Ver figura 36).

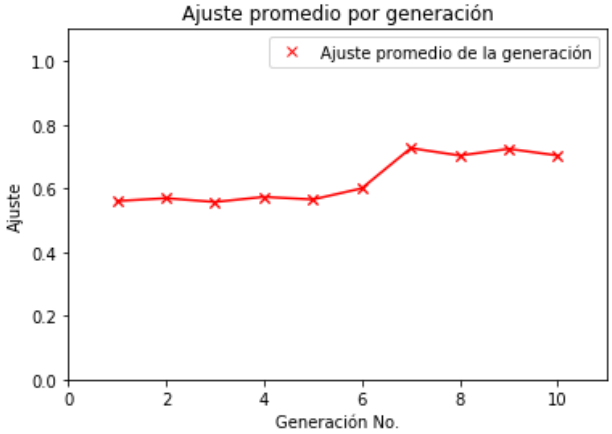

Figura 36. Evolución del ajuste promedio de cada generación de prueba 3.

El valor del ajuste para la mejor solución de cada una de las generaciones obtenidas en la prueba 3 tiene un comportamiento acorde con el visto en el ajuste promedio y este puede evidenciarse en la Figura 37, aunque el incremento en el ajuste obtenido es mínimo, es evidente que existe mejora.

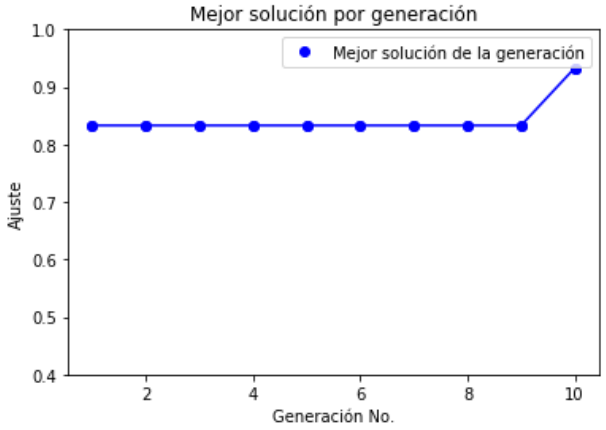

Figura 37. Evolución del mejor individuo de cada generación de prueba 3.

La solución óptima obtenida de la prueba 3 solo necesita de una unidad estándar para satisfacer la orden de corte requerida (Ver Figura 33). La manera en que se debe proceder a cortar cada uno de los ítems de la orden de corte se encuentra representada gráficamente en la Figura 38.

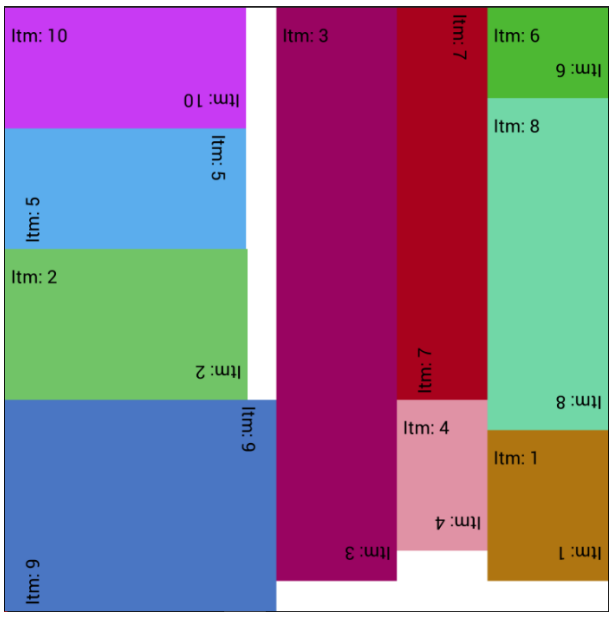

Figura 38. Representación gráfica de la solución obtenida en la prueba 3.

6.6 Prueba 4

Los resultados obtenidos una vez ejecutada la prueba número 4 muestran que el algoritmo tuvo que ejecutarse una vez con los parámetros establecidos para esta prueba (Ver Tabla 3). La solución se encontró en la décimo segunda generación con un ajuste de 100% (Ver Figura 39).

```
Unidad estandar No. 1
El algoritmo se ejecutó 1 veces
Fitnes: 1.0
Ancho: 200.0 Alto: 200.0 Area: 40000.0
Generacion No.12
Area Items: 40000.0
El tiempo de ejecucion fue:3.5073606967926025
```

Figura 39. Resultados prueba No. 4

Las figuras 40 y 41 muestran la distribución del ajuste de los individuos de la primera y última generación respectivamente, siendo la generación final aquella en la que se encontró la solución para la prueba 4. Puede observarse como el ajuste de los individuos es muy variado en la primera generación y tiene un promedio cercano al 70% (Ver Figura 40).

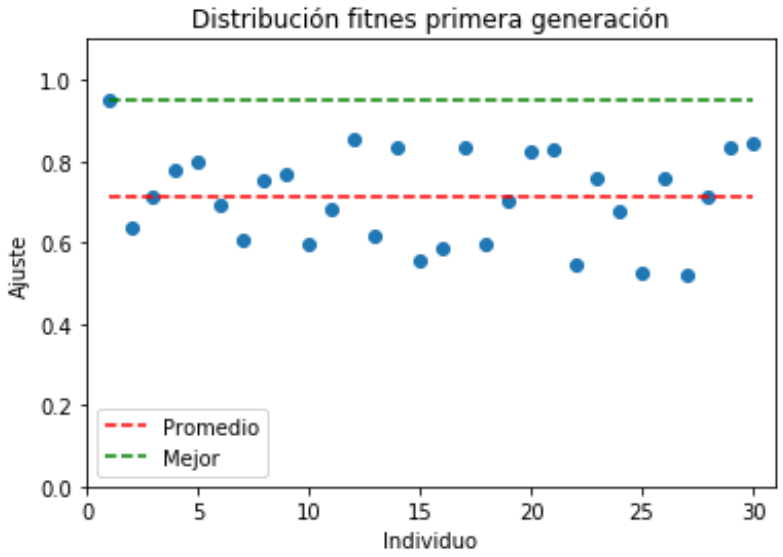

Figura 40. Distribución de ajuste de los individuos de la población inicial de la prueba 4.

Por el contrario, en la generación en que se encontró la solución óptima para la prueba 4, el promedio es cercano al 80% y se puede observar claramente como la distribución de los valores de ajuste tiende a ser más compacta (Ver figura 41).

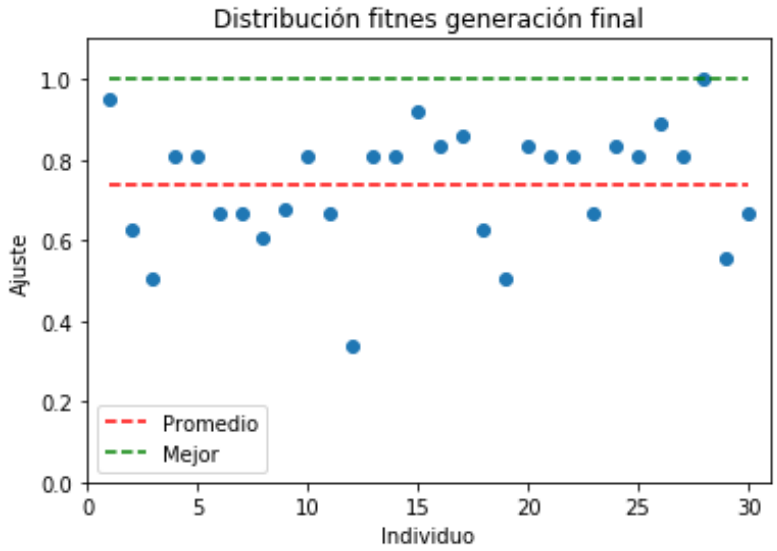

Figura 41. Distribución de ajuste de los individuos de la población en la que se encontró la solución óptima de la prueba 4.

La mejora de las posibles soluciones para la prueba 4 puede evidenciarse con el incremento del ajuste promedio de cada una de las generaciones obtenidas por el algoritmo, este paso de 70% a un 80% aproximadamente, esto con variaciones paulatinas (Ver figura 42).

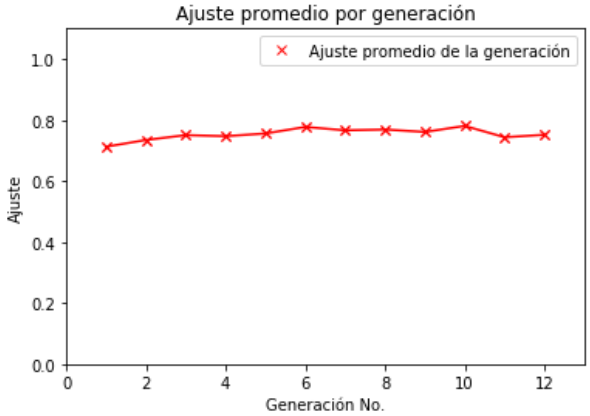

Figura 42. Evolución del ajuste promedio de cada generación prueba 4.

El valor del ajuste para la mejor solución de cada una de las generaciones obtenidas en la prueba 4 tiene un comportamiento acorde con el visto en el ajuste promedio y este puede evidenciarse en la Figura 43, aunque el incremento en el ajuste obtenido es mínimo, es evidente que existe mejora.

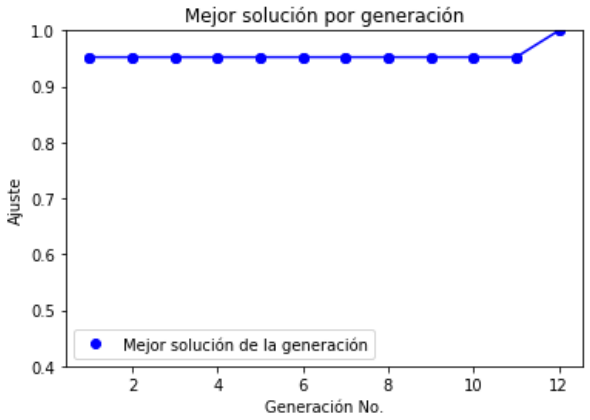

Figura 43. Evolución del mejor individuo de cada generación prueba 4.

La solución óptima obtenida de la prueba 4 solo necesita de una unidad estándar para satisfacer la orden de corte requerida (Ver Figura 39). La manera en que se debe proceder a cortar cada uno de los ítems de la orden de corte se encuentra representada gráficamente en la Figura 44.

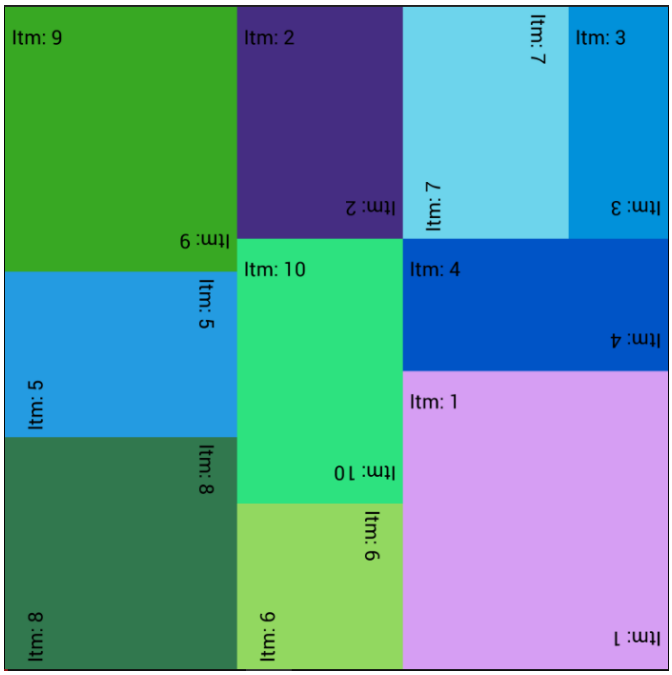

Figura 44. Representación gráfica de la solución obtenida en la prueba 4.

6.7 Prueba 5

Los resultados obtenidos una vez ejecutada la prueba número 5 muestran que el algoritmo tuvo que ejecutarse una vez con los parámetros establecidos para esta prueba (Ver Tabla 3). La solución se encontró en la tercera generación con un ajuste de 100% (Ver Figura 45).

```
Unidad estandar No. 1
El algoritmo se ejecutó 1 veces
Fitnes: 1.0
Ancho: 200.0 Alto: 200.0 Area: 40000.0
Generacion No.3
Area Items: 40000.0
El tiempo de ejecucion fue:2.5520360469818115
```

Figura 45. Resultados prueba No. 5

Las figuras 46 y 47 muestran la distribución del ajuste de los individuos de la primera y última generación respectivamente, siendo la generación final aquella en la que se encontró la solución para la prueba 5. Puede observarse como el ajuste de los individuos es muy variado en la primera generación y tiene un promedio cercano al 70% (Ver Figura 46).

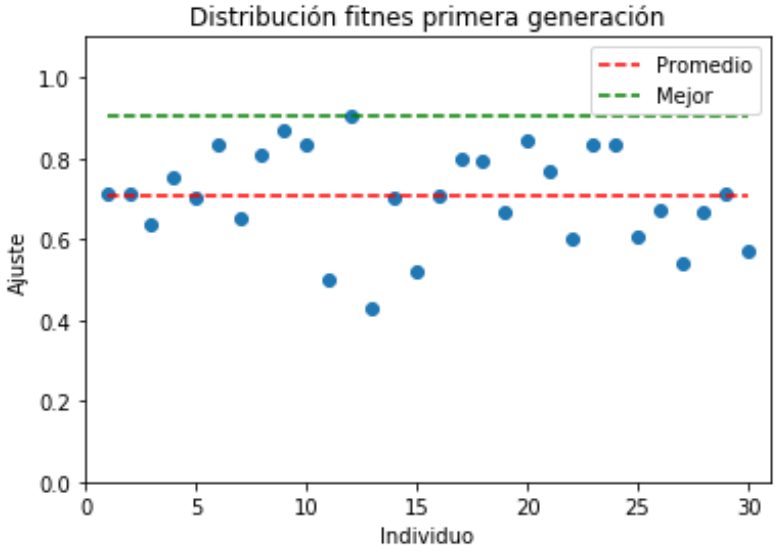

Figura 46. Distribución de ajuste de los individuos de la población inicial de la prueba 5.

En la generación en que se encontró la solución óptima para la prueba 5, el promedio sigue siendo cercano al 70%, pero se puede observar claramente como la distribución de los valores de ajuste tiende a ser más compacta (Ver figura 47).

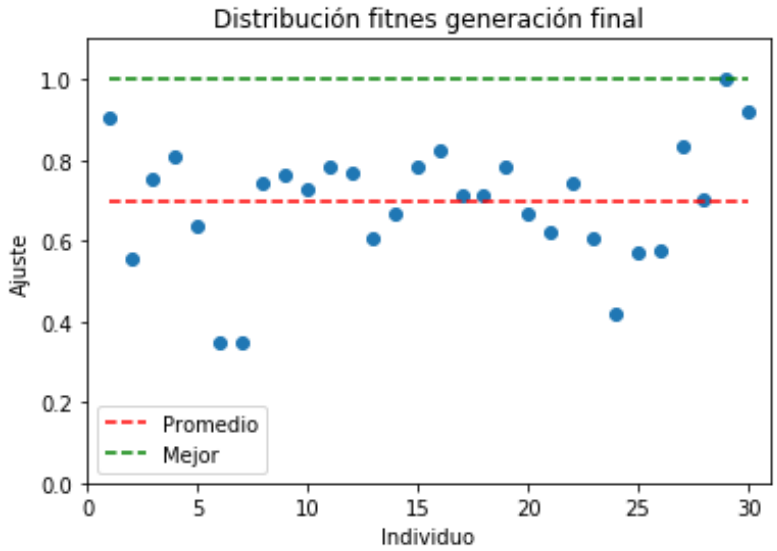

Figura 47. Distribución de ajuste de los individuos de la población en la que se encontró la solución óptima de la prueba 5.

La mejora de las posibles soluciones para la prueba 5 no es fácil de evidenciar mediante el incremento del ajuste promedio de cada una de las generaciones obtenidas por el algoritmo, ya que para las tres generaciones se mantiene cercano al 70%. Pese a esto, si se puede observar una pequeña variación (Ver figura 48).

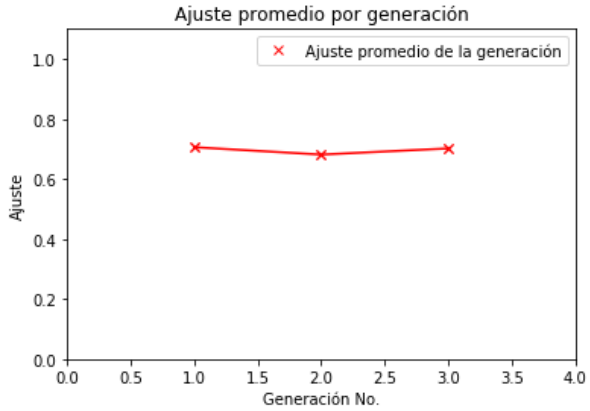

Figura 48. Evolución del ajuste promedio de cada generación de prueba 5.

La variación del ajuste para la mejor solución de cada una de las generaciones obtenidas en la prueba 5 es sumamente clara en cuanto a su incremento, contrario al comportamiento visto en el ajuste promedio, esta variación se puede evidenciar en la Figura 49.

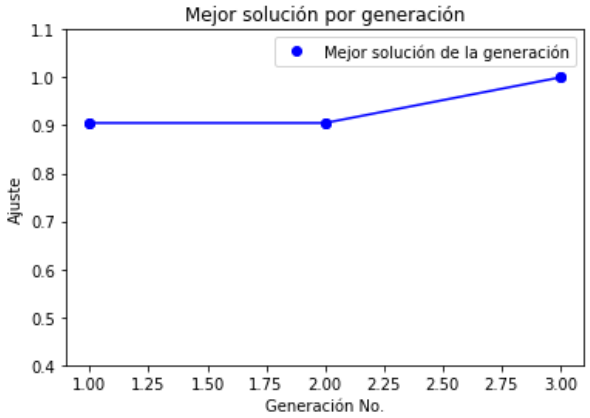

Figura 49. Evolución del mejor individuo de cada generación de prueba 5.

La solución óptima obtenida de la prueba 5 solo necesita de una unidad estándar para satisfacer la orden de corte requerida (Ver Figura 45). La manera en que se debe proceder a cortar cada uno de los ítems de la orden de corte se encuentra representada gráficamente en la Figura 50.

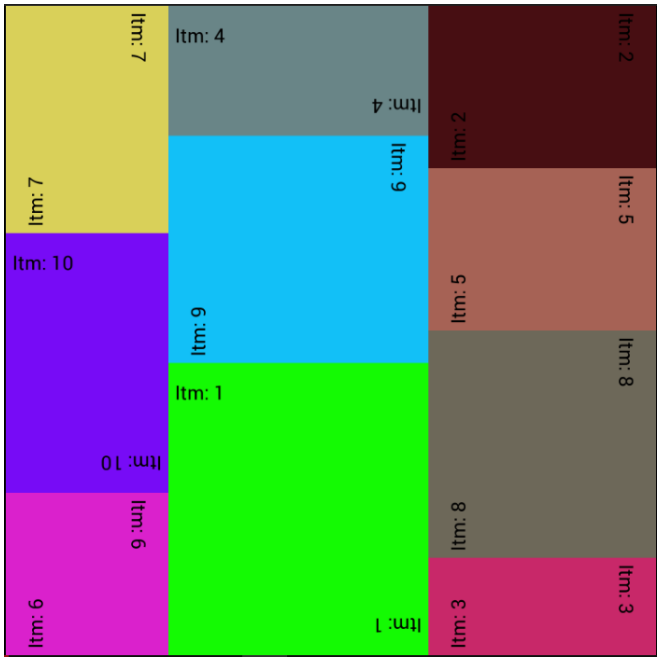

Figura 50. Representación gráfica de la solución obtenida en la prueba 5.

6.8 Prueba 6

Los resultados obtenidos una vez ejecutada la prueba número 6 muestran que el algoritmo tuvo que ejecutarse una vez con los parámetros establecidos para esta prueba (Ver Tabla 3). La solución se encontró en la tercera generación con un ajuste de 100% (Ver Figura 51).

```
Unidad estandar No. 1
El algoritmo se ejecutó 1 veces
Fitnes: 1.0
Ancho: 200.0 Alto: 200.0 Area: 40000.0
Generacion No.3
Area Items: 40000.0
El tiempo de ejecucion fue:3.522433042526245
```

Figura 51. Resultados prueba No. 6

Las figuras 52 y 53 muestran la distribución del ajuste de los individuos de la primera y última generación respectivamente, siendo la generación final aquella en la que se encontró la solución para la prueba 6. Puede observarse como el ajuste de los individuos es muy variado en la primera generación y tiene un promedio cercano al 70% (Ver Figura 52).

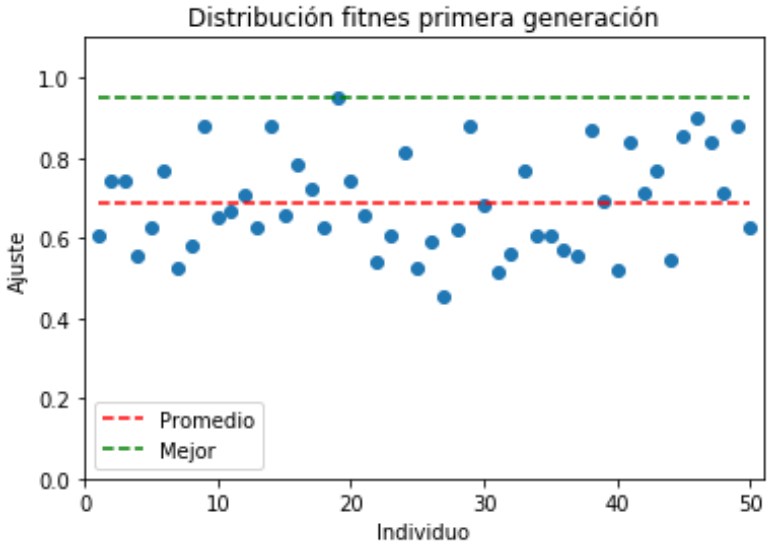

Figura 52. Distribución de ajuste de los individuos de la población inicial de la prueba 6.

En la generación en que se encontró la solución óptima para la prueba 6, el promedio sigue siendo cercano al 70%, pero se puede observar claramente como la distribución de los valores de ajuste tiende a ser más compacta (Ver figura 53).

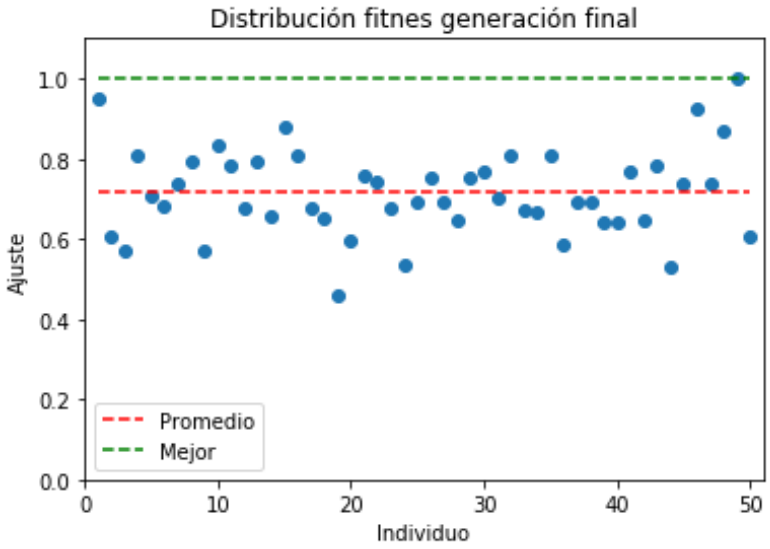

Figura 53. Distribución de ajuste de los individuos de la población en la que se encontró la solución óptima de la prueba 6.

La mejora de las posibles soluciones para la prueba 6 no es fácil de evidenciar mediante el incremento del ajuste promedio de cada una de las generaciones obtenidas por el algoritmo, ya que para las tres generaciones se mantiene cercano al 70%. Pese a esto, si se puede observar una pequeña variación (Ver figura 54).

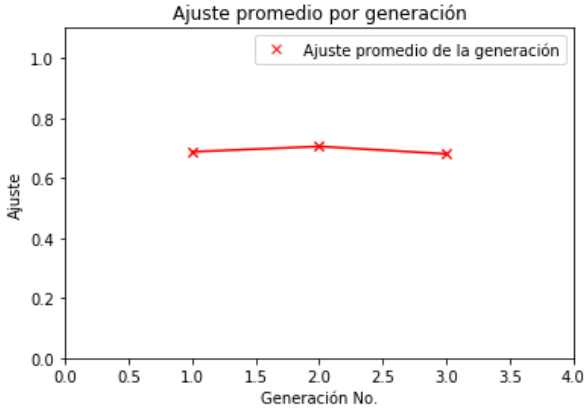

Figura 54. Evolución del ajuste promedio de cada generación en prueba 6.

La variación del ajuste para la mejor solución de cada una de las generaciones obtenidas en la prueba 6 es sumamente clara en cuanto a su incremento, contrario al comportamiento visto en el ajuste promedio, esta variación se puede evidenciar en la Figura 55.

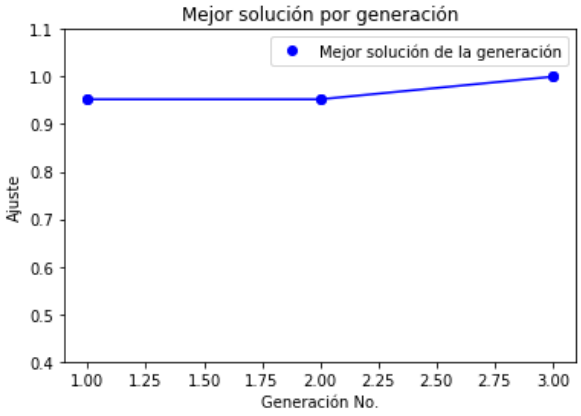

Figura 55. Evolución del mejor individuo de cada generación en prueba 6.

La solución óptima obtenida de la prueba 6 solo necesita de una unidad estándar para satisfacer la orden de corte requerida (Ver Figura 51). La manera en que se debe proceder a cortar cada uno de los ítems de la orden de corte se encuentra representada gráficamente en la Figura 56.

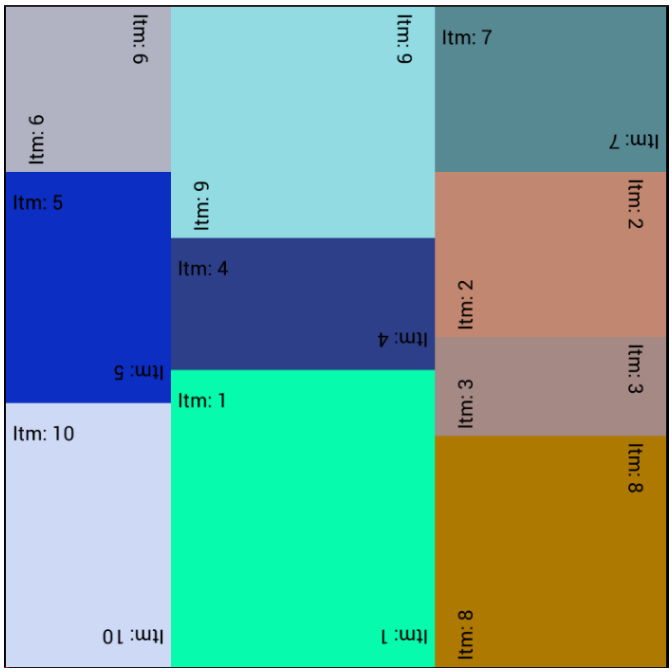

Figura 56. Representación gráfica de la solución obtenida en la prueba 6.

Capítulo 7

Conclusiones

En este capítulo se describen las conclusiones obtenidas como resultado de los hallazgos del trabajo de investigación y desarrollo llevado a cabo en este trabajo de disertación. Además, se consideran las fortalezas y debilidades de esta, así como recomendaciones para trabajos futuros enfocados a la aplicación de algoritmos genéticos en el problema de corte de materiales.

7.1 Conclusiones

El algoritmo diseñado e implementado con el propósito de obtener soluciones óptimas factibles para la variante del problema de corte de materiales abordado, demostró luego de las pruebas que es capaz de generar resultados que cumplen con las restricciones propuestas en la fase de diseño y se apegan al producto deseado, ya que el algoritmo resultante es perfectamente capaz de maximizar el área usada para satisfacer una orden de corte no únicamente respetando el tamaño de la unidad estándar utilizada sino también la proporción existente entre sus dimensiones, lo cual en caso de que el residuo obtenido sea de dimensiones considerables pueda ser aprovechable para futuros usos.

El establecimiento de un análisis comparativo entre los diferentes tipos de algoritmos genéticos, para determinar cuál de ellos debería aplicarse con el afán de resolver un problema, el corte de materiales en el caso de esta investigación, resulta ser una tarea complicada. Pese a que se puede diferenciar varios tipos de algoritmos genéticos la implementación de cada uno de estos, su desempeño, complejidad computacional dependerán directamente del problema que están destinados a resolver. Por lo cual comparar algoritmos genéticos de diferentes tipos y que resuelven diferentes problemas no arrojaría ningún resultado determinante a la hora de realizar un análisis de los datos comparados.

La adopción de la estructura del algoritmo genético hibrido usada para construir el algoritmo que se presenta en esta investigación permitió la integración de heurísticas que facilitaron las etapas de creación de las poblaciones lo cual mejoró la calidad de las soluciones obtenidas de la ejecución de este.

A diferencia de la mayoría de las soluciones propuestas para el problema de corte de materiales desde su descubrimiento, las cuales se enfocan en resolver casos particulares y específicos de este problema; la solución expuesta en el presente trabajo permite obtener soluciones aceptables para uno de los cuatro principales subconjuntos de tipos del problema. Esto se debe al uso de un algoritmo genético el cual puede adaptarse a cualquier orden de corte, dimensiones de unidad estándar y parámetros de operadores genéticos siempre y cuando el problema a solucionar admita una solución que requiera cortes de guillotina y la rotación de los ítems a cortar.

7.2 Fortalezas

Debido a la forma en que se diseñó e implementó el algoritmo, este puede ser usado para determinar en qué manera satisfacer una determinada orden de corte de ítems rectangulares, sin importar el tamaño de la unidad estándar que se desee utilizar ni la cantidad de estas que se requieran.

Que el algoritmo cuente con una interfaz gráfica que complementa la visualización de resultados permite evidenciar de manera clara que las soluciones obtenidas satisfacen y cumplen con los requerimientos del problema a resolver.

7.3 Limitaciones

El diseño de la solución propuesta no permite el tratamiento de problemas de corte de materiales que no sean del tipo que admite rotación y cortes de guillotina, por lo cual tratar de resolver problemas con restricciones distintas a estas no es posible.

7.4 Recomendaciones

Antes de culminar se desea sugerir algunas recomendaciones en base a las conclusiones y resultados obtenidos luego de desarrollar este trabajo de disertación.

- Extender los estudios sobre el empleo de heurísticas para la generación de poblaciones del algoritmo genético pues este elemento contribuye en gran medida al mejoramiento de las soluciones obtenidas.
- Trabajar en mejorar el operador genético de cruce para individuos con codificación de datos combinados, estudiar la implementación del cruce de dos puntos combinado don el método PMX para mejorar la descendencia obtenida.
- Para el desarrollo futuro del tema tratado en este trabajo de disertación podría considerarse parametrizar la restricción de rotación de los ítems para que de este modo el algoritmo sea capaz de generar soluciones para un grupo mayor de problemas de la familia del CSP. Ya que, si bien el método de resolución propuesto en este trabajo no está tan focalizado como los algoritmos propuestos en los diversos desarrollos e investigaciones de años previos, el algoritmo genético expuesto aun no es aplicable a cualquier tipo de problema de corte de materiales.
- Después de ver los resultados obtenidos durante las pruebas del algoritmo, se aconseja considerar cambiar la estrategia usada para la obtención de cada una de las nuevas generaciones ya que la estrategia actual no demostró ser muy eficaz al actuar sin el operador genético de mutación.

Anexos

Anexo 1: Scripts algoritmo genético

Script Clases.py

```
# -*- coding: utf-8 -*-
"""
Created on Sun Nov 18 18:30:49 2018

@author: Henry Trujillo
"""

import random
from copy import deepcopy
import math

class Item:
    def __init__(self, largo, alto,nombre):
        self.largo=largo
        self.alto=alto
        self.id=nombre
        self.area=largo*alto
    def nombre_item(self):
        return self.nombre

#lista de solucion con valores de cada item,lista de la codificacion
class Solucion:
    def __init__(self,solucion,codificada):
        self.valor=solucion
        self.codificacion=codificada
        self.ajuste=0
        self.peso=0
        self.valor_ruleta=0

    def ver_lamina(self):
        return [self.valor,self.codificacion]

    def area_util(self):
        areas=[]
        lista=deepcopy(self.valor)
        def analisis(s_arbol):
            if(type(s_arbol)==Item):
                areas.append(s_arbol.area)
                s_arbol.pop()
                while(len(s_arbol)>0):
                    tmp=s_arbol.pop()
                    if(type(tmp)!=Item):
                        analisis(tmp)
                    else:
                        item_tmp=tmp
                        areas.append(item_tmp.area)
        analisis(lista)
        return sum(areas)

    def set_ajuste(self, valor_fitnes):
```

```
        self.ajuste=valor_fitnes

    def set_peso(self,valor_peso):
        self.peso=valor_peso

def area_usada(solucion):
    operadores=[]
    itms=[]
    def analisis(s_arbol):
        if(type(s_arbol)==Item):
            itms.append(s_arbol)
        else:
            operadores.append(s_arbol.pop())
            tmp=s_arbol.pop()
            if(type(tmp)!=Item):
                analisis(tmp)
            else:
                itms.append(tmp)
            tmp=s_arbol.pop()
            if(type(tmp)!=Item):
                analisis(tmp)
            else:
                itms.append(tmp)
                while(len(itms)>1):
                    i=[]
                    operacion=operadores.pop()
                    tmp1=itms.pop()
                    tmp2=itms.pop()
                    if(operacion=='A'):

i=Item(tmp1.largo+tmp2.largo,max(tmp1.alto,tmp2.alto),1000)
                    elif(operacion=='B'):

i=Item(tmp1.largo+tmp2.alto,max(tmp1.alto,tmp2.largo),1000)
                    elif(operacion=='C'):

i=Item(tmp1.alto+tmp2.largo,max(tmp1.largo,tmp2.alto),1000)
                    elif(operacion=='D'):

i=Item(tmp1.alto+tmp2.alto,max(tmp1.largo,tmp2.largo),1000)
                    elif(operacion=='E'):

i=Item(max(tmp1.largo,tmp2.largo),tmp1.alto+tmp2.alto,1000)
                    elif(operacion=='F'):

i=Item(max(tmp1.largo,tmp2.alto),tmp1.alto+tmp2.largo,1000)
                    elif(operacion=='G'):

i=Item(max(tmp1.alto,tmp2.largo),tmp1.largo+tmp2.alto,1000)
                    elif(operacion=='H'):

i=Item(max(tmp1.alto,tmp2.alto),tmp1.largo+tmp2.largo,1000)
                    itms.append(i)
    lista=deepcopy(solucion)
    analisis(lista)
    return itms[0]
```

```
def codificar_solucion(solucion_completa):
    fenotipo=[]
    lista=deepcopy(solucion_completa)
    def analisis(s_arbol):
        if(type(s_arbol)==Item):
            fenotipo.append(s_arbol.id)
            fenotipo.append(s_arbol.pop())
            while(len(s_arbol)>0):
                tmp=s_arbol.pop()
                if(type(tmp)!=Item):
                    analisis(tmp)
                else:
                    item_tmp=tmp
                    fenotipo.append(item_tmp.id)
    analisis(lista)
    return(fenotipo)

def poblacion(lista_corte,x,y,n_poblacion):
    poblacion=[]
    while(len(poblacion)<n_poblacion):
        poblacion.append(generar_solucion(lista_corte,x,y))
    return{'poblacion':poblacion}

def maximo(poblacion):
    i = 0
    posicion = 0
    bandera=0
    for m in poblacion:
        if (m.ajuste>bandera):
            bandera=m.ajuste
            posicion=i
        i+=1
    return poblacion[posicion]

def generar_solucion(lista_corte,x_estandar,y_estandar):
    solucion=[]
    codificada=[]
    x=x_estandar*10
    y=y_estandar*10
    def verificar_dimensiones(h1, h2, p1, p2):
        if(h1<=p1 and h2<=p2):
            return True
        elif(h2<=p1 and h1<=p2):
            return True
        else:
            return False

    def determinar_unidad_estandar(lista_corte,x,y):
        pendientes=deepcopy(lista_corte)
        opciones=['A','B','C','D','E','F','G','H']
        items_tmp=[]
        while(len(pendientes)>1):
            op_i=deepcopy(opciones)
            random.shuffle(pendientes)
            tmp1=pendientes.pop()
```

```
            tmp1_a=area_usada(tmp1)
            tmp2=[]
            diferencia=1000000
            pendientes_c=deepcopy(pendientes)
            pendientes=[]
            while(len(pendientes_c)>0):
                i_T2=pendientes_c.pop()
                tmpv2=area_usada(i_T2)

                m_dif=min(math.fabs(tmp1_a.largo-tmpv2.largo),
                   math.fabs(tmp1_a.largo-tmpv2.alto),
                   math.fabs(tmp1_a.alto-tmpv2.alto),
                   math.fabs(tmp1_a.alto-tmpv2.largo))
                if(m_dif<=diferencia):
                    diferencia=m_dif
                    if(tmp2!=[]):
                        items_tmp.append(tmp2)
                    tmp2=i_T2
                else:
                    items_tmp.append(i_T2)
            minima_area=1000000000000000
            op_minimo=''
            while(len(op_i)>0):
                op=op_i.pop()
                area_tmp=area_usada([tmp1,tmp2,op])
                if(area_tmp.area<=minima_area):
                    minima_area=area_tmp.area
                    op_minimo=op
                if(len(op_i)==0):
                    items_tmp.append([tmp1,tmp2,op_minimo])
            if(len(pendientes)==1 and len(items_tmp)>0):
                items_tmp.append(pendientes.pop())
            if(len(pendientes)==0 and len(items_tmp)>0):
                pendientes=deepcopy(items_tmp)
                items_tmp=[]
            if(len(pendientes)==1):
                break
        return(pendientes[0])
    lista=deepcopy(lista_corte)
    if(len(lista)==1):
        solucion = lista
        codificada=codificar_solucion(solucion[0])
    else:
        while(True):
            solucion=determinar_unidad_estandar(lista,x,y)
            area_solucion=area_usada(solucion)
            if(verificar_dimensiones(area_solucion.largo, area_solucion.alto,
x_estandar*2, y_estandar*2)):
                break
        codificada=codificar_solucion(solucion)
    return Solucion(solucion,codificada)

def menor_par(lista, x_estandar, y_estandar):
    op_i=['A','B','C','D','E','F','G','H']
    pendientes=deepcopy(lista)
    random.shuffle(pendientes)
```

```
    tmp1=pendientes.pop()
    tmp2=pendientes.pop()
    minima_area=1000000000000000
    op_minimo=''
    while(len(op_i)>0):
        op=op_i.pop()
        area_tmp=area_usada([tmp1,tmp2,op])
        if(area_tmp.area<=minima_area and area_tmp.largo<=x_estandar and
area_tmp.alto<=y_estandar):
            minima_area=area_tmp.area
            op_minimo=op
        elif(area_tmp.area<=minima_area and area_tmp.largo<=y_estandar and
area_tmp.alto<=x_estandar):
            minima_area=area_tmp.area
            op_minimo=op
        if(len(op_i)==0):
            pendientes.append([tmp1,tmp2,op_minimo])
            pendientes.append([op_minimo,tmp2.id,tmp1.id])
            pendientes.append((tmp1.area+tmp2.area)/minima_area)
    return pendientes
```

Script Algoritmo_Genetico.py

```
# -*- coding: utf-8 -*-
"""
Created on Sun Nov 18 18:33:59 2018

@author: Henry Trujillo
"""

import Clases as C
import random
from copy import deepcopy
import math

def ajuste(solucion,ex,ey):

solucion.set_ajuste(solucion.area_util()/C.area_usada(solucion.valor).area)

def peso(solucion, ajuste):
    x=math.ceil(ajuste*10)
    solucion.set_peso(x)

def valores_ruleta(poblacion):
    total=0
    for solucion in poblacion:
        total+=solucion.ajuste
    for solucion in poblacion:
        solucion.valor_ruleta=(solucion.ajuste)/total

def seleccion_ruleta(poblacion):
    objetivo=random.uniform(0, 1)
    suma=0
    seleccionado=C.Solucion([],[])
    for solucion in poblacion:
        suma+=solucion.valor_ruleta
        if suma>=objetivo:
            seleccionado=solucion
            break
    return seleccionado

#codifica a las soluciones resultado del cruce
def codificar_descendencia(items, s_codificada):
    def buscar_item(item_id, lista_items):
        res = C.Item(0,0,0)
        for i in lista_items:
            if(i.id == item_id):
                res=i
                break
        return res

    codificada=deepcopy(s_codificada)
    pila_items=[]
    pila_operadores=[]
    while(len(codificada)>0):
        tmp=codificada.pop()
```

```
        if(type(tmp)==int):
            pila_items.append(buscar_item(tmp,items))
        else:
            pila_operadores.append(tmp)
        if(len(pila_items)>=2 and len(pila_operadores)>0):
            s_1=pila_items.pop()
            s_2=pila_items.pop()
            s_tmp=[s_2,s_1,pila_operadores.pop()]
            pila_items.append(s_tmp)
    return pila_items[0]

def verificar_mutacion(s_codificada):
    codificada=deepcopy(s_codificada)
    pila_items=[]
    pila_operadores=[]
    while(len(codificada)>0):
        tmp=codificada.pop()
        if(type(tmp)==int):
            pila_items.append(tmp)
        else:
            pila_operadores.append(tmp)
        if(len(pila_items)>=2 and len(pila_operadores)>0):
            s_1=pila_items.pop()
            s_2=pila_items.pop()
            s_tmp=[s_2,s_1,pila_operadores.pop()]
            pila_items.append(s_tmp)
    if(len(pila_items)==1 and len(pila_operadores)==0):
        return True
    else:
        return False

def cruce(padre_A,padre_B):
    def separar(lista):
        items=[]
        for i in range(len(lista)):
            if type(lista[i])==int:
                items.append(lista[i])
                lista[i]='X'
        return {'operadores':lista, 'items':items}

    def Diff(li1, li2):
        return (list(set(li1) - set(li2)))

    def Verificar_Redundantes(estatico, cambio, diferentes):
        cont = 0
        for i in estatico:
            if i in cambio:
                estatico[cont]=diferentes.pop()
            cont+=1
    def reconstruir(arg):
        i=len(arg['operadores'])-1
        while(len(arg['items'])>0):
            if(arg['operadores'][i]=='X'):
                arg['operadores'][i]=arg['items'].pop()
            i-=1
```

```
    padre_1=deepcopy(padre_A)
    padre_2=deepcopy(padre_B)
    c1=separar(padre_1)
    c2=separar(padre_2)
    todos_items=deepcopy(c1['items'])
    n_items_cruce=random.randint(1, len(c1['items'])-1)
    cambio_1=c1['items'][0:n_items_cruce]
    estatico_1=c1['items'][n_items_cruce:len(todos_items)]
    cambio_2=c2['items'][0:n_items_cruce]
    estatico_2=c2['items'][n_items_cruce:len(todos_items)]
    diferentes_1 = Diff(cambio_1,cambio_2)
    diferentes_2 = Diff(cambio_2,cambio_1)
    Verificar_Redundantes(estatico_1, cambio_2, diferentes_1)
    Verificar_Redundantes(estatico_2, cambio_1, diferentes_2)
    c1['items']=cambio_2+estatico_1
    c2['items']=cambio_1+estatico_2
    contador=0
    i=len(c1['operadores'])-1
    while(contador<n_items_cruce):
        if(c1['operadores'][i]!='X' and c2['operadores'][i]!='X'):
            tmp=c1['operadores'][i]
            c1['operadores'][i]=c2['operadores'][i]
            c2['operadores'][i]=tmp
        else:
            contador+=1
        i-=1
    reconstruir(c1)
    reconstruir(c2)
    return {'d1':c1['operadores'], 'd2':c2['operadores']}

def mutacion(codificada_o, items):
    limite_posicion = len(codificada_o)-1
    while(True):
        codificada = deepcopy(codificada_o)
        primero = random.randint(0, limite_posicion)
        segundo=primero
        while(segundo==primero):
            segundo = random.randint(0, limite_posicion)
        tmp = codificada[primero]
        codificada[primero] = codificada[segundo]
        codificada[segundo] = tmp
        if(verificar_mutacion(codificada)):
            break
    mutada_valor = codificar_descendencia(items, codificada)
    return C.Solucion(mutada_valor, codificada)

def nueva_generacion(poblacion, tamanio_generacion_total, lista_items, ex, ey,
pm, pa):

    def verificar_dimensiones(h1, h2, p1, p2):
        if(h1<=p1 and h2<=p2):
            return True
        elif(h2<=p1 and h1<=p2):
            return True
        else:
            return False
```

```
    siguiente_generacion=[]
    valores_ruleta(poblacion)
    tamanio_generacion = tamanio_generacion_total -pa
    solucion_max=C.maximo(poblacion)
    siguiente_generacion.append(solucion_max)
    while(len(siguiente_generacion)<tamanio_generacion):
        sel=seleccion_ruleta(poblacion)
        sel2=seleccion_ruleta(poblacion)
        hijos=cruce(sel.codificacion,sel2.codificacion)
        c_hijo1=codificar_descendencia(lista_items,hijos['d1'])
        c_hijo2=codificar_descendencia(lista_items,hijos['d2'])
        hijo_uno=C.Solucion(c_hijo1,hijos['d1'])
        hijo_dos=C.Solucion(c_hijo2,hijos['d2'])
        ajuste(hijo_uno,ex,ey)
        ajuste(hijo_dos,ex,ey)
        peso(hijo_uno,hijo_uno.ajuste)
        peso(hijo_dos,hijo_dos.ajuste)
        area_hijo=C.area_usada(hijo_uno.valor)
        area_p1=C.area_usada(sel.valor)
        area_p2=C.area_usada(sel2.valor)

        if(verificar_dimensiones(area_hijo.largo, area_hijo.alto,
area_p1.largo, area_p1.alto)):
            siguiente_generacion.append(hijo_uno)
        elif(verificar_dimensiones(area_hijo.largo, area_hijo.alto,
area_p2.largo, area_p2.alto)):
            siguiente_generacion.append(hijo_uno)
        if(len(siguiente_generacion)==tamanio_generacion):
            break
        area_hijo=C.area_usada(hijo_dos.valor)
        if(verificar_dimensiones(area_hijo.largo, area_hijo.alto,
area_p1.largo, area_p1.alto)):
            siguiente_generacion.append(hijo_dos)
        elif(verificar_dimensiones(area_hijo.largo, area_hijo.alto,
area_p2.largo, area_p2.alto)):
            siguiente_generacion.append(hijo_dos)
    cte=0
    while(cte<pm):
        x=random.randint(1, len(siguiente_generacion)-1)
        seleccion_mutacion = siguiente_generacion[x]
        seleccion_mutacion =
mutacion(seleccion_mutacion.codificacion,lista_items)
        ajuste(seleccion_mutacion,ex,ey)
        siguiente_generacion[x] = seleccion_mutacion
        cte=cte+1
    cte=0
    while(cte<pa):

        solucion_tmp=C.generar_solucion(lista_items,ex,ey)
        ajuste(solucion_tmp,ex,ey)
        siguiente_generacion.append(solucion_tmp)
        cte=cte+1
    return siguiente_generacion
```

```
def algoritmo(grupo, ex_estandar, ey_estandar, tam_poblacion, ajuste_min,
porcentaje_mutacion):
    itms=deepcopy(grupo)
    p_inicial=C.poblacion(itms,ex_estandar,ey_estandar,tam_poblacion)
    for i in p_inicial['poblacion']:
        ajuste(i,ex_estandar,ey_estandar)
        peso(i,i.ajuste)
    bandera=1
    nucleo=deepcopy(p_inicial['poblacion'])
    p_mutacion=math.ceil(tam_poblacion*porcentaje_mutacion)
    p_agregacion=p_mutacion
    contador=0
    variacion=0
    parametro_V=0.00000000005
    ultimo_mejor=0
    actual_mejor=0
    Solucion_obtenida = {}
    while(bandera<500):

generacion=nueva_generacion(nucleo,tam_poblacion,grupo,ex_estandar,ey_estandar
, p_mutacion, p_agregacion)
        nucleo=deepcopy(generacion)
        contador+=1
        m_sol_gen = C.maximo(generacion)
        xcv=C.area_usada(m_sol_gen.valor)
        actual_mejor=m_sol_gen.ajuste
        variacion=actual_mejor-ultimo_mejor
        ultimo_mejor=actual_mejor
        if(xcv.largo<=ex_estandar and xcv.alto<=ey_estandar and
actual_mejor>=ajuste_min):
            Solucion_obtenida = {'solucion': m_sol_gen,
'generaciones':bandera}
            break
        elif(xcv.largo<=ey_estandar and xcv.alto<=ex_estandar and
actual_mejor>=ajuste_min):
            Solucion_obtenida = {'solucion': m_sol_gen,
'generaciones':bandera}
            break
        if(contador==20):
            contador=0
            if(variacion<parametro_V or variacion==0):
                Solucion_obtenida = {'generaciones': -bandera}
                break
        bandera+=1
    return Solucion_obtenida

def algoritmo_uno(grupo, ex_estandar, ey_estandar):
    respuesta = C.Solucion(grupo[0],grupo[0].id)
    respuesta.set_ajuste(1)
    return respuesta

def algoritmo_par(grupo, ex_estandar, ey_estandar):
    minimo_par=C.menor_par(grupo,ex_estandar, ey_estandar)
    respuesta = C.Solucion(minimo_par[0],minimo_par[1])
    respuesta.set_ajuste(minimo_par[2])
    return respuesta
```

Anexo 2: Script Principal

```
# -*- coding: utf-8 -*-
"""
Created on Sat Dec  1 21:43:14 2018

@author: Henry Trujillo
"""

import Algoritmo_Genetico as AG
import Clases as C
from copy import deepcopy
import PIL.ImageDraw as ImageDraw,PIL.Image as Image, PIL.ImageShow as
ImageShow, PIL.ImageFont as ImageFont
import random
from time import time

def imprimir(solucion, x, y):
    operadores=[]
    itms=[]
    font = ImageFont.truetype('Roboto-Regular.ttf', 60)
    def item_to_img(item):
        im=Image.new('RGBA', (int(item.largo*10),int(item.alto*10)),
(random.randint(0, 255),random.randint(0, 255),random.randint(0, 255)))
        draw = ImageDraw.Draw(im)
        draw.text((20, 50), "Itm: "+str(item.id), fill='black',font=font)
        im2=im.rotate(180, expand=True)
        draw = ImageDraw.Draw(im2)
        draw.text((20, 50), "Itm: "+str(item.id), fill='black',font=font)
        return im2

    def analisis(s_arbol):
        if(type(s_arbol)==C.Item):
            itms.append(item_to_img(s_arbol))
        else:
            operadores.append(s_arbol.pop())
            tmp=s_arbol.pop()
            if(type(tmp)!=C.Item):
                analisis(tmp)
            else:
                itms.append(item_to_img(tmp))
            tmp=s_arbol.pop()
            if(type(tmp)!=C.Item):
                analisis(tmp)
            else:
                itms.append(item_to_img(tmp))
                while(len(itms)>1):
                    im=[]
                    operacion=operadores.pop()
                    tmp1=itms.pop()
                    tmp2=itms.pop()
                    if(operacion=='A'):
                        im = Image.new('RGBA',
(tmp1.width+tmp2.width,max(tmp1.height,tmp2.height)), 'white')
```

```
                        im.paste(tmp1,(0,0))
                        im.paste(tmp2,(tmp1.width,0))
                    elif(operacion=='B'):
                        r2=tmp2.rotate(90, expand=True)
                        im = Image.new('RGBA',
(tmp1.width+r2.width,max(tmp1.height,r2.height)), 'white')
                        im.paste(tmp1,(0,0))
                        im.paste(r2,(tmp1.width,0))
                    elif(operacion=='C'):
                        r1 =tmp1.rotate(90, expand=True)
                        im = Image.new('RGBA',
(r1.width+tmp2.width,max(r1.height,tmp2.height)), 'white')
                        im.paste(r1,(0,0))
                        im.paste(tmp2,(r1.width,0))
                    elif(operacion=='D'):
                        r1 =tmp1.rotate(90, expand=True)
                        r2=tmp2.rotate(90, expand=True)
                        im = Image.new('RGBA',
(r1.width+r2.width,max(r1.height,r2.height)), 'white')
                        im.paste(r1,(0,0))
                        im.paste(r2,(r1.width,0))
                    elif(operacion=='E'):
                        im = Image.new('RGBA',
(max(tmp1.width,tmp2.width),tmp1.height+tmp2.height), 'white')
                        im.paste(tmp1,(0,0))
                        im.paste(tmp2,(0,tmp1.height))
                    elif(operacion=='F'):
                        r2=tmp2.rotate(90, expand=True)
                        im = Image.new('RGBA',
(max(tmp1.width,r2.width),tmp1.height+r2.height), 'white')
                        im.paste(tmp1,(0,0))
                        im.paste(r2,(0,tmp1.height))
                    elif(operacion=='G'):
                        r1 =tmp1.rotate(90, expand=True)
                        im = Image.new('RGBA',
(max(r1.width,tmp2.width),r1.height+tmp2.height), 'white')
                        im.paste(r1,(0,0))
                        im.paste(tmp2,(0,r1.height))
                    elif(operacion=='H'):
                        r1 =tmp1.rotate(90, expand=True)
                        r2=tmp2.rotate(90, expand=True)
                        im = Image.new('RGBA',
(max(r1.width,r2.width),r1.height+r2.height), 'white')
                        im.paste(r1,(0,0))
                        im.paste(r2,(0,r1.height))
                    itms.append(im)
    lista=deepcopy(solucion)
    analisis(lista)
    if(itms[0].width<=x*10):
        lamina = Image.new('RGBA', (int(x*10),int(y*10)), 'yellow')
    else:
        lamina = Image.new('RGBA', (int(y*10),int(x*10)), 'yellow')
    lamina.paste(itms[0],(0,0))
    return lamina

def grupo_items(items, area_estandar, relacion_util):
```

```
    def ordenamientoBurbuja(unaLista):
        for numPasada in range(len(unaLista)-1,0,-1):
            for i in range(numPasada):
                if (unaLista[i].area>=unaLista[i+1].area):
                    temp = unaLista[i]
                    unaLista[i] = unaLista[i+1]
                    unaLista[i+1] = temp
    suma_areas=0
    limite=relacion_util*area_estandar
    grupo_t = []
    ordenamientoBurbuja(items)
    residuo=[]
    while(suma_areas<=limite):
        m=items.pop()
        if(suma_areas+m.area<=limite):
            grupo_t.append(m)
            suma_areas=suma_areas+m.area
        else:
            residuo.append(m)
        if(len(items)==0):
                break
    return {'grupo':grupo_t, 'residuo':residuo}

def main(eAncho, eAlto, pA, tP,pM, pF, orden_corte):
    A_Respuesta=[]
    tiempo_inicial = time()
    ex_estandar = eAncho
    ey_estandar = eAlto
    ajuste_min = pF
    area_est_uni=ex_estandar*ey_estandar
    tam_poblacion = tP
    relacion_util = pA
    porcentaje_mutacion = pM
    i_1=deepcopy(orden_corte)
    dat=""
    items=deepcopy(i_1)
    contador_estandar=1
    num_iteraciones=1
    while(len(items)>0):

        grupo=grupo_items(items,area_est_uni, relacion_util)
        if(len(grupo['grupo'])==1):
            s = AG.algoritmo_uno(grupo['grupo'], ex_estandar, ey_estandar)
            xcv=C.area_usada(s.valor)
            x= imprimir(s.valor, ex_estandar, ey_estandar)
            dat+="Unidad estandar No. "+str(contador_estandar)+"\n"
            dat+="Fitnes: "+str(s.ajuste)+"\n"
            dat+="Ancho: "+str(xcv.largo)+"\tAlto: "+str(xcv.alto)+"\tArea:
"+str(xcv.area)+"\n"
            dat+="Area Items: "+str(s.area_util())+"\n\n"
            dat+="Items obtenidos:\nID\tAncho\tAlto\n"
            for i in grupo['grupo']:
                dat+=str(i.id)+"\t"+str(i.largo)+"\t"+str(i.alto)+"\n"
            dat+="\n"
            items=grupo['residuo']
            contador_estandar+=1
```

```
            A_Respuesta.append({'data': dat, 'img':x})
            dat=""
            num_iteraciones=1
        elif(len(grupo['grupo'])==2):
            s = AG.algoritmo_par(grupo['grupo'], ex_estandar, ey_estandar)
            xcv=C.area_usada(s.valor)
            x= imprimir(s.valor, ex_estandar, ey_estandar)
            dat+="Unidad estandar No. "+str(contador_estandar)+"\n"
            dat+="Fitnes: "+str(s.ajuste)+"\n"
            dat+="Ancho: "+str(xcv.largo)+"\tAlto: "+str(xcv.alto)+"\tArea:
"+str(xcv.area)+"\n"
            dat+="Area Items: "+str(s.area_util())+"\n\n"
            dat+="Items obtenidos:\nID\tAncho\tAlto\n"
            for i in grupo['grupo']:
                dat+=str(i.id)+"\t"+str(i.largo)+"\t"+str(i.alto)+"\n"
            dat+="\n"
            items=grupo['residuo']
            contador_estandar+=1
            A_Respuesta.append({'data': dat, 'img':x})
            dat=""
            num_iteraciones=1
        else:
            Respuesta = AG.algoritmo(grupo['grupo'], ex_estandar, ey_estandar,
tam_poblacion,ajuste_min, porcentaje_mutacion)
            if(Respuesta['generaciones']>0):
                s = Respuesta['solucion']
                xcv=C.area_usada(s.valor)
                x= imprimir(s.valor, ex_estandar, ey_estandar)
                dat+="\n=====================================\n\n"
                dat+="Unidad estandar No. "+str(contador_estandar)+"\n"
                dat+="El algoritmo se ejecutó "+str(num_iteraciones)+"
veces\n"
                dat+="Fitnes: "+str(s.ajuste)+"\n"
                dat+="Ancho: "+str(xcv.largo)+"\tAlto:
"+str(xcv.alto)+"\tArea: "+str(xcv.area)+"\n"
                dat+="Generacion No."+str(Respuesta['generaciones'])+"\n"
                dat+="Area Items: "+str(s.area_util())+"\n\n"
                dat+="Items obtenidos:\nID\tAncho\tAlto\n"
                for i in grupo['grupo']:
                    dat+=str(i.id)+"\t"+str(i.largo)+"\t"+str(i.alto)+"\n"
                dat+="\n=====================================\n\n"
                items=grupo['residuo']
                contador_estandar+=1
                A_Respuesta.append({'data': dat, 'img':x})
                dat=""
                num_iteraciones=1
            else:
                num_iteraciones+=1
                if(num_iteraciones==20):
                    dat+="\nNo se pudo encontrar una solucion para las
condiciones dadas\n"
                    A_Respuesta=[]
                    break
                dat+="\nEl algoritmo presenta una convergencia prematura en la
generación No: "+str(Respuesta['generaciones']*(-1))+"\n"
                dat+="El algorimo procederá a ejecutarce nuevamente.\n\n"
```

```
                items=grupo['grupo']+grupo['residuo']

    tiempo_final = time()
    tiempo_ejecucion = tiempo_final - tiempo_inicial
    dat+='\nEl tiempo de ejecucion fue:'+str(tiempo_ejecucion)+"\n\n"
    A_Respuesta.append(dat)
    return A_Respuesta
```

Anexo 3: Script interfaz gráfica

```
# -*- coding: utf-8 -*-
"""
Created on Thu Nov 22 13:46:42 2018

@author: Henry Trujillo
"""

import tkinter as tk
from tkinter.filedialog import askopenfilename
from tkinter.messagebox import showerror
import csv
import Clases as C
import Principal as M
import PIL.ImageDraw as ImageDraw,PIL.Image as Image, PIL.ImageShow as
ImageShow, PIL.ImageFont as ImageFont
from concurrent.futures import ThreadPoolExecutor

orde_corte=[]
last_id=0
ejecucion=True
class FullScreenApp(object):
    def __init__(self, master, **kwargs):
        self.master=master
        pad=3
        self._geom='200x200+0+0'
        master.geometry("{0}x{1}+0+0".format(
            master.winfo_screenwidth()-pad, master.winfo_screenheight()-pad))
        master.bind('<Escape>',self.toggle_geom)
    def toggle_geom(self,event):
        geom=self.master.winfo_geometry()
        self.master.geometry(self._geom)
        self._geom=geom


def load_file():
    global orde_corte
    global last_id
    fname = askopenfilename(filetypes=(("Template files", "*.csv"),
                                       ("All files", "*.*") ))
    if fname:
        try:
            dt_infoTxt.insert(tk.INSERT,"Archivo abierto: "+str(fname)+'\n')
            s="ID\tAncho\tAlto\n"
            with open(fname, newline='') as csvfile:
                reader = csv.reader( csvfile, delimiter=';')
                orde_corte = []
                dt_lista.delete(1.0,tk.END)
                for row in reader:
                    s = s + str(row[0])+'\t'+str(row[1])+'\t'+str(row[2])
+'\n'
                    orde_corte.append(
C.Item(float(row[1]),float(row[2]),int(row[0])))
```

```
                dt_lista.insert(tk.INSERT, s)
                last_id = orde_corte[-1].id
        except:                     # <- naked except is a bad idea
            showerror("Open Source File", "Failed to read file\n'%s'" % fname)
            dt_infoTxt.insert(tk.INSERT,"Open Source File: Failed to read
file"+str(fname))
        return

#Ejecutar algoritmo
def ejecutar_algoritmo2():
    eAn=float(eAncho.get())
    eAl=float(eAlto.get())
    pA=float(pAreaUsada.get())
    tP=float(tPoblacion.get())
    pM=float(pMutacion.get())
    pF=float(pAjuste.get())
    dt_infoTxt.insert(tk.INSERT,"Ejecutando Algoritmo... \n\n")
    r=M.main(eAn, eAl, pA, tP, pM, pF, orde_corte)

    dt_infoTxt.insert(tk.INSERT,r.pop())

    for i in r:
        dt_infoTxt.insert(tk.INSERT,i['data'])
        i['img'].show()

#asincronico
def _do(eAn, eAl, pA, tP, pM, pF, orde_corte):
    global ejecucion
    r=M.main(eAn, eAl, pA, tP, pM, pF, orde_corte)
    time = r.pop()

    if(len(r)>0):
        for i in r:
            dt_infoTxt.insert(tk.INSERT,i['data'])
            i['img'].show()
    dt_infoTxt.insert(tk.INSERT,time)
    ejecucion=False

def ejecutar_algoritmo():
    global ejecucion
    ejecucion=True
    eAn=float(eAncho.get())
    eAl=float(eAlto.get())
    pA=float(pAreaUsada.get())
    tP=float(tPoblacion.get())
    pM=float(pMutacion.get())
    pF=float(pAjuste.get())

    executor = ThreadPoolExecutor(max_workers=1)
    executor.submit(_do, eAn, eAl, pA, tP, pM, pF, orde_corte)
    dt_infoTxt.insert(tk.INSERT,"Ejecutando Algoritmo...")

def reset():
    global orde_corte
    dt_lista.delete(1.0,tk.END)
    dt_infoTxt.delete(1.0,tk.END)
```

```
    eAncho.set(200)
    eAlto.set(200)
    pAreaUsada.set(0.7)
    tPoblacion.set(30)
    pMutacion.set(0.3)
    pAjuste.set(0.8)
    iAncho.set("")
    iAlto.set("")
    orde_corte=[]
    dt_lista.insert(tk.INSERT, "ID\tAncho\tAlto\n")

def add_item():
    global orde_corte
    global last_id
    last_id+=1
    new_item=C.Item(float(iAncho.get()),float(iAlto.get()), last_id)
    orde_corte.append(new_item)
    s = str(last_id)+'\t'+str(iAncho.get())+'\t'+str(iAlto.get()) +'\n'
    dt_lista.insert(tk.INSERT, s)
    iAncho.set("")
    iAlto.set("")

#interfaz

root = tk.Tk()
root.title('Corte de Materiales')
root.geometry('{}x{}'.format(1980, 700))

# crea contenedores principales
top_frame = tk.Frame(root, bg='white', width=1980, height=50, pady=3)
center = tk.Frame(root, bg='gray72', width=1980, height=250, padx=5, pady=5)
btm_frame = tk.Frame(root, bg='gray84', width=1980, height=300, pady=3)

# layout contenedores principales
root.grid_rowconfigure(1, weight=1)
root.grid_columnconfigure(0, weight=1)

top_frame.grid(row=0, sticky="ew")
center.grid(row=2, sticky="ew")
btm_frame.grid(row=3, sticky="ew")

#Varibles
eAncho=tk.StringVar()
eAlto=tk.StringVar()
listaI=tk.StringVar()
pAreaUsada=tk.StringVar()
tPoblacion=tk.StringVar()
pMutacion=tk.StringVar()
pAjuste=tk.StringVar()
iAncho=tk.StringVar()
iAlto=tk.StringVar()

#layout top
top_frame.grid_columnconfigure(0, weight=1)
# widgets top
```

```
titulo_app = tk.Label(top_frame, text='Algoritmo Genético para el Corte de
Material Bidimensional Rectangular')
titulo_app.config(font=("Courier", 20))
titulo_app.config(bg="white")
dsc_label=tk.Label(top_frame,text='Descripcion/información
aplicacion',bg='white')
# layout widgets top
titulo_app.grid(row=0,column=0, columnspan=3)
dsc_label.grid(row=1,column=0)
# create the center widgets
center.grid_rowconfigure(0, weight=1)
center.grid_columnconfigure(3,weight=1)

ctr_left = tk.Frame(center, bg='white', width=250, height=250, padx=15,
pady=30)
ctr_mid = tk.Frame(center, bg='white', width=300, height=250, padx=15,
pady=30)
ctr_right = tk.Frame(center, bg='white', width=510, height=250,padx=15)
ctr_last = tk.Frame(center, bg='white', width=200, height=250,padx=15,
pady=40)

ctr_left.grid(row=0, column=0, sticky="ns")
ctr_mid.grid(row=0, column=1, sticky="ns")
ctr_right.grid(row=0, column=2, sticky="ns")
ctr_last.grid(row=0, column=3, sticky="ns")

#unidad estandar
dt_estandar = tk.Label(ctr_left, text='Dimensiones Unidad Estandar',
bg='white', font=("Courier", 12))
estandar_ancho = tk.Label(ctr_left, text='Ancho:', bg='white',
font=("Courier", 15))
estandar_alto = tk.Label(ctr_left, text='Alto:', bg='white', font=("Courier",
15))
input_eAncho = tk.Entry(ctr_left, background="ivory3",font=("Courier", 15),
width=10, textvariable=eAncho)
input_eAlto = tk.Entry(ctr_left, background="ivory3",font=("Courier", 15),
width=10, textvariable=eAlto)

dt_estandar.grid(row=0, column=0, columnspan=2)
estandar_ancho.grid(row=1, column=0)
estandar_alto.grid(row=2, column=0)
input_eAncho.grid(row=1, column=1)
input_eAlto.grid(row=2, column=1)
#parametros algoritmo
dt_algoritmo = tk.Label(ctr_mid, text='Parámetros del Algoritmo', bg='white',
font=("Courier", 12))
p_area_usada = tk.Label(ctr_mid, text='% Area Usada:', bg='white',
font=("Courier", 15))
t_poblacion = tk.Label(ctr_mid, text='Tamaño Población:', bg='white',
font=("Courier", 15))
p_mutacion = tk.Label(ctr_mid, text='% de Mutación:',
bg='white',font=("Courier", 15))
p_ajuste = tk.Label(ctr_mid, text='Ajuste esperado:',
bg='white',font=("Courier", 15))
input_pAreaUsada = tk.Entry(ctr_mid, background="ivory3",font=("Courier", 15),
width=10, textvariable=pAreaUsada)
```

```
input_tPoblacion = tk.Entry(ctr_mid, background="ivory3",font=("Courier", 15),
width=10, textvariable=tPoblacion)
input_pMutacion = tk.Entry(ctr_mid, background="ivory3",font=("Courier", 15),
width=10, textvariable=pMutacion)
input_pAjuste= tk.Entry(ctr_mid, background="ivory3",font=("Courier", 15),
width=10, textvariable=pAjuste)

dt_algoritmo.grid(row=0, column=0, columnspan=2)
p_area_usada.grid(row=1, column=0)
t_poblacion.grid(row=2, column=0)
p_mutacion.grid(row=3, column=0)
p_ajuste.grid(row=4, column=0)
input_pAreaUsada.grid(row=1, column=1)
input_tPoblacion.grid(row=2, column=1)
input_pMutacion.grid(row=3, column=1)
input_pAjuste.grid(row=4, column=1)

#ingreso items
dt_items = tk.Label(ctr_right, text='Ingreso Items', bg='white',
font=("Courier", 12))
item_ancho = tk.Label(ctr_right, text='Ancho:', bg='white', font=("Courier",
15))
item_alto = tk.Label(ctr_right, text='Alto:', bg='white', font=("Courier",
15))
input_iAncho = tk.Entry(ctr_right, background="ivory3",font=("Courier", 15),
width=10, textvariable=iAncho)
input_iAlto = tk.Entry(ctr_right, background="ivory3",font=("Courier", 15),
width=10, textvariable=iAlto)
btn_iCrear = tk.Button(ctr_right,text='Agregar Nuevo
Item',background="SkyBlue3", font=("Courier", 15), command=add_item)
btn_iLeer = tk.Button(ctr_right,text='Cargar Archivo',background="SkyBlue3",
font=("Courier", 15), command=load_file)

dt_items.grid(row=0, column=0, columnspan=2)
item_ancho.grid(row=1, column=0)
item_alto.grid(row=2, column=0)
input_iAncho.grid(row=1, column=1)
input_iAlto.grid(row=2, column=1)
btn_iCrear.grid(row=3, column=0, columnspan=2, pady=10)
btn_iLeer.grid(row=4, column=0, columnspan=2, pady=10)

#botones ejecucion
btn_Iniciar = tk.Button(ctr_last,text='Ejecutar
Algoritmo',background="yellow3", font=("Courier", 15),
command=ejecutar_algoritmo)
btn_Reset = tk.Button(ctr_last,text='Reset',background="yellow3",
font=("Courier", 15), command=reset)

btn_Iniciar.grid(row=0, column=0, columnspan=2, pady=10)
btn_Reset.grid(row=1, column=0, columnspan=2, pady=10)

#salida datos
btm_frame.grid_rowconfigure(0, weight=1)
btm_frame.grid_columnconfigure(1,weight=1)
```

```
btm_left = tk.Frame(btm_frame, bg='white', width=250, height=250, padx=15,
pady=15)
btm_right = tk.Frame(btm_frame, bg='white', width=510, height=250,padx=15,
pady=15)

btm_left.grid(row=0, column=0, sticky="ns")
btm_right.grid(row=0, column=1, sticky="ns")

dt_listaItems = tk.Label(btm_left, text='Lista Items', bg='white',
font=("Courier", 12))
dt_lista = tk.Text(btm_left, bg='white', font=("Courier", 12),
width=40,height=18)

dt_listaItems.grid(row=0, column=0)
dt_lista.grid(row=1, column=0)

dt_info = tk.Label(btm_right, text='Información', bg='white', font=("Courier",
12))
dt_infoTxt = tk.Text(btm_right, bg='white', font=("Courier", 12),width=90,
height=18, fg='black')

dt_info.grid(row=0, column=0)
dt_infoTxt.grid(row=1, column=0)

eAncho.set(200)
eAlto.set(200)
pAreaUsada.set(1)
tPoblacion.set(30)
pMutacion.set(0.1)
pAjuste.set(0.8)
dt_lista.insert(tk.INSERT, "ID\tAncho\tAlto\n")

#lanzar aplicacion
root.mainloop()
```

Bibliografía

Álvarez, L., Jiménez, N., & Lanzagorta, O. De. (2012). *Introducción a los Algoritmos genéticos*. Madrid.

Ben Amor, H., & Valério de Carvalho, J. (2005). *Cutting Stock Problems*. Boston: Springer. https://doi.org/10.1007/0-387-25486-2

Berkey, J., & Wang, P. (1987). Two-Dimensional Finite Bin-Packing Algorithms. *The Journal of the Operational Research Society*, *38*(5), 434–429.

Deep, K., & Mebrahtu, H. (2012). Variant of Partially Mapped Crossover for the Travelling Salesman Problems. *International Journal of Combinatorial Optimization Problems and Informatics*, *3*(1), 47–69.

Dyson, R. G., & Gregory, A. S. (1974). The Cutting Stock Problem in the Flat Glass Industry. *Operational Research Quarterly*, *25*(1), 41–53.

Gestal, M., Rivero, D., Rabuñal, J., Dorado, J., & Pazos, A. (2010). *Introducción a los Algoritmos Genéticos y la Programación Genética* (1st ed.). Coruña: Universidad de Coruña, Servicio de Publicaciones.

Gilmore, P. C., & Gomory, R. (1965). Multi-Stage Cutting Stock Problems of Two or More Dimensions. *Operations Research*, *13*(1), 94–120. https://doi.org/10.1287/opre.13.1.94

Gilmore, P. C., & Gomory, R. E. (1961). A Linear Programming Approach to the Cutting-Stock Problem Author (s): P . C . Gilmore and R . E . Gomory Published by : INFORMS Stable URL : http://www.jstor.org/stable/167051 REFERENCES Linked references are available on JSTOR for this article : You m. *Operations Research*, *9*(6), 849–859.

Goldberg, D. E. (1989). *Genetic Algorithms in Search , Optimization , and Machine Learning*. Boston: Addison-Wesley Longman Publishing.

Haupt, R. L., & Haupt, S. E. (2004). *ALGORITHMS PRACTICAL GENETIC ALGORITHMS*. Hoboken: John Wiley & Sons, Inc.

Kuhlman, D. (2009). A Python Book: Beginning Python, Advanced Python, and Python Exercises, 1–227. Retrieved from http://96.243.233.8:5190/get/pdf/A Python Book_ Beginning Python, Advanced Python, and Python Exercises - Dave Kuhlman_14890.pdf

Lodi, A. (1999). *Algorithms for Two-Dimensional Bin Packing and Assignment Problems*.

Lodi, A., Martello, S., & Monaci, M. (2002). Two-dimensional packing problems: A survey. *European Journal of Operational Research*, *141*(2), 241–252. https://doi.org/10.1016/S0377-2217(02)00123-6

Lodi, A., Martello, S., & Vigo, D. (1999). Heuristic and Metaheuristic Approaches for a Class of Two-Dimensional Bin Packing Problems Heuristic and Metaheuristic Approaches for a Class of Two-Dimensional Bin Packing Problems. *Informs Journal on Computing*, *11*(4), 345–357.

Oliveira, J. F., Neuenfeldt Júnior, A., Silva, E., & Carravilla, M. A. (2016). a Survey on Heuristics for the Two-Dimensional Rectangular Strip Packing Problem. *Pesquisa Operacional*, *36*(2), 197–226. https://doi.org/10.1590/0101-7438.2016.036.02.0197

Rao, S. S. (2009). *Engineering Optimization: Theory and Practice*. *Theory and Practice* (4th ed.). Hoboken: John Wiley & Sons, Inc. https://doi.org/10.1002/9780470549124

Sheppard, C. (2016). *Genetic Algorithms with Python*.

Suliman, S. M. A. (2001). Pattern generating procedure for the cutting stock problem. *International Journal of Production Economics*, *74*(1–3), 293–301.

The Spyder Website Contributors. (2018). SPYDER. Retrieved from https://www.spyder-ide.org