

Modelos de Optimización II

Al desperdicio, ¡ni un tantico así!

Dalianys Pérez Pereira
Grupo C411

A.UNO@LAB.MATCOM.UH.CU

Dayany Alfaro González
Grupo C411

D.ALFARO@MATCOM.UH.CU

Gilberto González Rodríguez
Grupo C411

A.DOS@LAB.MATCOM.UH.CU

Antonio Jesús Otaño Barrera
Grupo C411

A.DOS@LAB.MATCOM.UH.CU

Resumen

El Resumen en Español debe constar de 100 a 200 palabras y presentar de forma clara y concisa el contenido fundamental del artículo.

Abstract

The English Abstract must have have 100 to 200 words, and present in a clear and concise form the essentials of the article content.

Palabras Clave: Separadas, Por, Comas.

Tema: Tema, Subtema.

1. Introducción

El problema de patrones de corte de piezas rectangulares pertenece a la familia de problemas de corte y empaquetamiento y sus aplicaciones se pueden observar en industrias de perfiles metálicos, corte de maderas, papel, plástico o vidrio en donde los componentes rectangulares tienen que ser cortados de grandes hojas de material. Para estas industrias es de gran importancia realizar este proceso de corte de una manera eficiente buscando minimizar el desperdicio y los demás costos asociados al proceso, teniendo en cuenta las restricciones técnicas y de demanda.

El problema de patrones de corte es un problema de gran complejidad tanto por las características y variables que involucra como por las técnicas que se utilizan para abordarlo, es una temática en constante evolución y muchos investigadores han desarrollado diversos modelos para resolverlo. El interés en este problema puede ser sustentado por su aplicación práctica y el reto que representa, pues, en general, es computacionalmente difícil de resolver ya que es un problema de tipo NP-completo, dado que los patrones de empaquetamiento incrementan exponencialmente con el número de rectángulos que deben ser empaquetados.

Este trabajo tiene como objetivo resolver un problema de corte en dos dimensiones, asociado a la industria del papel, presente en la empresa Geocuba ubicada en la provincia Pinar del Río, Cuba. Se propone el diseño e implementación de un algoritmo que permita determinar qué patrones de corte deben usarse para cortar un conjunto de hojas de forma que se satisfaga

una demanda (de hojas más pequeñas) solicitada por el usuario de forma que el desperdicio resultante de los cortes sea el menor posible. En este caso se permite la rotación de las piezas a colocar y se requiere que los cortes sean de tipo guillotina, es decir, que el corte, ya sea horizontal o vertical, vaya de un extremo a otro de la hoja a picar y que a su vez produzca dos nuevas hojas que satisfagan este tipo de corte.

La solución propuesta está implementada haciendo uso de Python como lenguaje. Además se brinda como parte de la solución una aplicación de escritorio con una interfaz de usuario para introducir una instancia del problema en cuestión.

[Texto explicando lo que aborda cada sección]

2. Antecedentes y Enfoques de Solución

El Problema de patrones de corte (CSP, por sus siglas en inglés) fue formulado por primera vez en 1939 por el economista ruso Kantorovich.

Han surgido numerosas investigaciones que abordan diferentes problemas según el tipo de dimensión (1D y 2D) y desde diversos enfoques tales como los métodos exactos, heurísticos y meta heurísticos, pero aún no existe un método global establecido para dar solución a este tipo de problemas, debido a la complejidad asociada.

2.1 Programación Lineal Entera

Casi todos los procedimientos basados en la programación lineal para resolver el problema de patrones de

corte se remontan a Gilmore y Gomory, [1], para lo cual, proponen la relajación de la restricción de integridad para la solución de problemas de programación lineal logrando minimizar el desperdicio a través de la generación de columnas evitando el conocimiento explícito o enumeración de todos los patrones desde el principio, ya que bajo este esquema las columnas (patrones) son generadas cuando se requieran [2]. La idea consiste en utilizar el método simplex revisado para resolver el problema de la entrada del patrón de corte siguiente a la base mediante la resolución de un problema de la mochila asociado. Este método es denominado en la literatura como *delayed column generation technique*, y permite resolver este tipo de problemas en un tiempo computacional mucho menor [3].

2.2 Procedimientos Heurísticos Secuenciales

Los procedimientos heurísticos secuenciales pertenecen a la clase de heurísticas de búsqueda local. La solución se construye mediante la generación de patrones uno a uno hasta que todos los requerimientos de demanda se hayan satisfecho, donde los patrones inicialmente seleccionados deben tener un nivel de desperdicio bajo, un nivel de utilización alto y dejar una serie de requerimientos para poder combinar bien los patrones futuros, evitando así incurrir posteriormente en desperdicios excesivos [3]. La ventaja principal de este método es que puede controlar otros factores aparte del desperdicio y elimina el problema del redondeo al trabajar sólo con valores enteros.

2.3 Procedimientos Heurísticos Híbridos

Este procedimiento consiste en combinar los dos procedimientos descritos anteriormente, de tal forma que se utilice el procedimiento heurístico secuencial para generar una solución, la cual es guardada y utilizada como base inicial en el procedimiento de programación lineal. Posteriormente, el desperdicio es reducido si es posible a través iteraciones adicionales, tal y como lo realiza [6].

Independientemente de la forma como se combinen estos dos métodos, lo más importante del éxito de la unión entre el procedimiento heurístico secuencial y el redondeo de problemas de programación lineal es la selección del criterio apropiado para resolver el problema [7].

2.4 Metaheurísticas

Ante el problema que presenta la búsqueda local y las heurísticas constructivas de quedar atrapadas en óptimos locales, surgen las metaheurísticas a mediados de 1970 pues tienen la capacidad de guiar la búsqueda local para que se escape de los óptimos locales. Muchos de estos algoritmos se han utilizado para resolver el problema de patrones de corte, entre los cuales se destaca *Tabu Search* (TS), *Greedy Randomized Adaptive Search Procedure* (GRASP) [8], Algoritmos genéticos [9, 10] y *Ant Colony Optimization* (ACO) [11, 12], entre otros algoritmos evolucionarios

[13].

3. Definición del Problema

Sea I el conjunto de n hojas, una hoja i está definida por su ancho w_i , su altura h_i y su demanda d_i . Sea J el conjunto de m patrones de corte con las mismas dimensiones $H \times W$, el número m es desconocido. Sea p_{ji} el número de hojas i presentes en el patrón j , se tiene que $p_j = (p_{j1}, \dots, p_{jn})$. Sea π_j una configuración de las hojas del patrón j , la cual incluye información acerca de la posición exacta de las hojas y si se encuentran rotadas o no. Por tanto, un patrón j va a estar definido por (p_j, π_j) . Sea x_j el número de veces que es necesario aplicar el patrón de corte j , se tiene que las variables de decisión son: m, p_j, x_j y $\pi_j \forall j \in J$.

El objetivo que se persigue es minimizar el desperdicio de papel:

$$f = \sum_{j \in J} c_j x_j \quad (1)$$

donde c_j es el área no usada del patrón j .

Para satisfacer la demanda d_i solicitada para cada hoja i van a aparecer n restricciones:

$$\sum_{j \in J} p_{ji} x_j \geq d_i \quad \forall i \in I \quad (2)$$

Es necesario también tener en cuenta la sobreproducción a la hora de analizar el espacio desperdiciado dado que cuando una solución sobrecumple la demanda se tiene que esas hojas sobreproducidas también van a ser desechadas. Por tanto para tener lo anterior en consideración hay que modificar la función objetivo que se muestra en la ecuación (1). Sea a_i el área total de la hoja i se tiene que:

$$sobreProd = \sum_{i \in I} sobreProd_i$$

donde $sobreProd_i = (\sum_{j \in J} p_{ji} x_j a_i) - d_i a_i$ va a ser el área total que ocupa la sobreproducción de hoja de tipo i sobreproducidas. La nueva función objetivo sería la siguiente:

$$f = \sum_{j \in J} c_j x_j + sobreProd \quad (3)$$

Esta función objetivo y las n restricciones presentadas antes definen un problema de programación lineal entero. Sea S el conjunto de todas las posibles soluciones, una solución $s \in S$ va a estar formada por $((p_1, \pi_1), \dots, (p_m, \pi_m))$ y (x_1, \dots, x_m) que es la solución de dicho problema lineal entero.

4. Propuesta de Solución

La solución que se propone en este trabajo es una adaptación de la solución brindada por [4]. El problema de patrones de cortes va a ser dividido en los siguientes 3 subproblemas de optimización:

1. El problema de programación lineal descrito en la sección anterior, el cual consiste en hallar (x_1, \dots, x_m) .

2. Un problema de empaquetamiento en 2 dimensiones (2D Bin Packing) que consiste en encontrar las configuraciones (π_1, \dots, π_m) de las hojas en los m patrones.
3. Un problema combinatorio que consiste en hallar los adecuados (p_1, \dots, p_m) . Para resolverlo se utiliza un algoritmo genético, el cual a su vez se va apoyando en los subproblemas anteriores.

4.1 Problema de Programación Lineal

Como ha sido explicado en la Sección 3 es necesario resolver un problema de programación lineal entero donde se quiere minimizar la función que se muestra en la ecuación (3) sujeta a las restricciones en la ecuación (2).

La eficiencia al resolver este subproblema resulta crucial debido a que va ser necesario resolverlo múltiples veces para evaluar y saber cuan buenas son las soluciones al problema de patrones de corte que se van obteniendo. Por esto se resuelve el problema relajado correspondiente por lo que la solución obtenida (x_1^*, \dots, x_m^*) se redondea de forma que el número de veces que es necesario aplicar el patrón de corte $x_j = \lceil x_j^* \rceil \quad \forall j \in J$.

Dado que en el presente trabajo se usó Python como lenguaje, para resolver esta fase del problema se exploraron las herramientas que brinda dicho lenguaje para resolver problemas de programación lineal y se decidió hacer uso de la biblioteca `cvxopt`.

4.2 2D Bin Packing

El problema 2D Bin Packing se puede enunciar de la siguiente manera: Dada una lista de rectángulos (R_1, \dots, R_n) determinar la forma de colocarlos todos sin que se solapen y usando el menor número posible de contenedores rectangulares (*bins*). En este caso los rectángulos serían las hojas que se demandan y un *bin* sería la hoja a picar. Como resultado se obtendría una lista de patrones $\pi = (\pi_1, \dots, \pi_k)$.

Existen diversas alternativas para resolver este problema [5]. En este caso se propone la variante *Guillotine*, que como su nombre indica cumple la restricción de cortes tipo guillotina. Este algoritmo en todo momento mantiene actualizada una lista "rectángulos libres" que representan el área disponible en el rectángulo principal (*bin*). Estos rectángulos libres cumplen que son disjuntos tomados dos a dos, es decir $F_i \cap F_j = \emptyset \quad \forall i \neq j$

y el área libre total del *bin* se puede definir como $\bigcup_{i=1}^m F_i$

donde $F = \{F_1, \dots, F_m\}$ es la lista de rectángulos libres. El algoritmo comienza con un único rectángulo libre, lo cual se representa como $F = \{F_1 = (W \times H)\}$. En cada iteración del algoritmo se selecciona un rectángulo libre F_i donde coloca el rectángulo $R = (w \times h)$ en la esquina inferior izquierda, por tanto F_i se sustituye por dos nuevos rectángulos F', F'' que cumplen que $F' \cap F'' = \emptyset$ y $F' \cup F'' \cup R = F_i$. Este procedimiento continúa hasta que se hayan colocado todos los rectángulos.

En la implementación del algoritmo *Guillotine* se usaron las siguientes heurísticas:

- **BSSF: Best Short Side Fit**

A la hora de escoger un rectángulo libre de $F = \{F_1, \dots, F_m\}$ para colocar el rectángulo $R = (w \times h)$ se selecciona el $F_i = (w_i \times h_i)$ tal que $\min(w_i - w, h_i - h)$ tiene el menor valor.

- **BFF: Bin First Fit**

Como se pueden tener varios *bins* abiertos a la vez se decide colocar $R = (w \times h)$ en el primer *bin* en el que quepa.

- **SAS: Shorter Axis Split**

Cuando se va a sustituir F_i por F', F'' se realiza un corte paralelo al lado más pequeño de F_i .

- **DESCSS: Sort by Shorter Side First in Descending Order**

Ordenar la entrada (R_1, \dots, R_n) de acuerdo al siguiente criterio:

$$R_i < R_j \iff \min(w_i, h_i) < \min(w_j, h_j)$$

- **RM: Rectangle Merge**

La principal limitante del algoritmo es que a causa de mantener una lista de rectángulos libres disjuntos es posible que no permita colocar un rectángulo aún cuando hay suficiente área libre para ello. Una manera en la que se puede mejorar el algoritmo es después de colocar un rectángulo encontrar todos los pares de rectángulos libre vecinos que pueden ser mezclados en uno solo y mezclarlos. Repetir este procedimiento mientras exista al menos un par de rectángulos libres F_i, F_j que cumplan dicha condición.

Se probó el algoritmo tanto con esta heurística como sin ella y de manera general el algoritmo produce mejores salidas aplicando la heurística pero para algunos casos produce una salida peor, por lo que se decidió que a la hora de ejecutar el algoritmo se decide si utilizar o no produciendo un número aleatorio entre 0 y 1. Si ese número es menor que un valor fijo (con 0.75 se obtuvieron buenos resultados) entonces se usa la heurística en esa ejecución del algoritmo. De esta manera se garantiza que se use la heurística en la mayor parte de las ejecuciones del algoritmo pero también permite explorar otras soluciones que solo se obtienen sin usarla y que podrían ser mejores.

En el Algoritmo 1 se muestra un pseudocódigo de la variante de solución propuesta para este problema, incluidas las heurísticas utilizadas.

4.3 Algoritmo Genético

Los algoritmos genéticos son llamados así porque se inspiran en la evolución biológica y su base genético-molecular. Estos algoritmos hacen evolucionar una población de individuos sometiéndola a acciones aleatorias semejantes a las que actúan en la evolución

Algorithm 1 Guillotine-BSSF-BFF-SAS-DESCSS-RM

Input: $R \leftarrow \{R_1 = (w_1 \times h_1), \dots, R_n = (w_n \times h_n)\}$
 $M \leftarrow (W \times H)$ ▷ Dimensión de la hoja principal
1: $F \leftarrow \{(W \times H)\}$
2: Ordenar R de acuerdo al criterio DESCSS
3: **for all** $R_k \leftarrow (w_k \times h_k) \in R$ **do**
4: Elegir rectángulo libre F_i y bin B_j donde colocar R_k
5: **if** F_i, B_j no existen **then**
6: Abrir nuevo bin B
7: $F \leftarrow F \cup \{(W \times H)\}$
8: $F_i \leftarrow (W \times H)$
9: $B_j \leftarrow B$
10: **end if**
11: Decidir orientación del rectángulo
12: Colocar rectángulo en la esquina inferior izquierda de F_i
13: Dividir espacio sobrante de F_i en F' y F''
14: $F \leftarrow F \cup \{F', F''\} \setminus F_i$
15: **while** Existan rectángulos F_a y F_b que se puedan mezclar **do**
16: Mezclar F_a y F_b en F^*
17: $F \leftarrow F \cup F^* \setminus \{F_a, F_b\}$
18: **end while**
19: **end for**

biológica (mutaciones y recombinaciones genéticas), así como también a una selección de acuerdo con algún criterio, en función del cual se decide cuáles son los individuos más adaptados, que sobreviven, y cuáles los menos aptos, que son descartados.[14]

El funcionamiento de un algoritmo genético básico se puede resumir en los pasos siguientes:

- **Inicialización:** Se genera aleatoriamente la población inicial, que está constituida por un conjunto de cromosomas los cuales representan las posibles soluciones del problema. En caso de no hacerlo aleatoriamente, es importante garantizar que dentro de la población inicial, se tenga la diversidad estructural de estas soluciones para tener una representación de la mayor parte de la población posible o al menos evitar la convergencia prematura.
- **Evaluación:** A cada uno de los cromosomas de esta población se aplicará la función de aptitud para saber cómo de "buena" es la solución que se está codificando.
- **Condición de término:** El AG se deberá detener cuando se alcance la solución óptima, pero esta generalmente se desconoce, por lo que se deben utilizar otros criterios de detención. Normalmente se usan dos criterios: correr el AG un número máximo de iteraciones (generaciones) o detenerlo cuando no haya cambios en la población. Mientras no se cumpla la condición de término se hace lo siguiente:

- **Selección:** Después de saber la aptitud de cada cromosoma se procede a elegir los cromosomas que serán cruzados en la siguiente generación. Los cromosomas con mejor aptitud tienen mayor probabilidad de ser seleccionados.
- **Recombinación o cruzamiento:** La recombinación es el principal operador genético, representa la reproducción sexual, opera sobre dos cromosomas a la vez para generar dos descendientes donde se combinan las características de ambos cromosomas padres.
- **Mutación:** Modifica al azar parte del cromosoma de los individuos, y permite alcanzar zonas del espacio de búsqueda que no estaban cubiertas por los individuos de la población actual.
- **Reemplazo:** Una vez aplicados los operadores genéticos, se seleccionan los mejores individuos para conformar la población de la generación siguiente. [14]

4.3.1 INICIALIZACIÓN

Para explicar el proceso seguido a la hora de crear una población inicial para el algoritmo lo primero sería definir qué sería el vecindario de una solución. Sea $s \in S$ una solución formada por la configuración (π_1, \dots, π_m) correspondiente a (p_1, \dots, p_m) y (x_1, \dots, x_n) . Un vecino de s se va a construir usando unos de los siguientes operadores: *añadir*, *eliminar*, *mover*, *intercambiar*.

Sea $L_{max}(i)$ el máximo número posible de hojas de tipo i en un patrón:

$$L_{max}(i) = \lfloor \frac{H \times W}{h_i w_i} \rfloor$$

Aplicar el operador *añadir* a una solución $s \in S$ añade una hoja i a un patrón j lo que se refleja como un aumento de p_{ij} si $p_{ij} < L_{max}(i)$. El operador *eliminar* consiste en quitar una imagen i del patrón j lo cual disminuye p_{ij} si el número total de hojas i en toda la solución s se mantiene estrictamente positivo. El operador *mover* mueve una hoja de un patrón a otro y el operador *intercambiar* intercambia dos hojas entre dos patrones diferentes.

Un vecino de una solución va a ser generado seleccionando aleatoriamente un operador, uno (*añadir, eliminar*) o dos (*mover, intercambiar*) patrones y una (*añadir, eliminar, mover*) o dos (*intercambiar*) hojas en esos patrones. La aplicación de dichos operadores puede dar como resultado una solución no válida porque sea imposible lograr una configuración para las cantidades p_j obtenidas. Para comprobar la validez de la solución obtenida se utilizó una versión del algoritmo 1, descrito en la Sección 4.2, donde solo va a existir un *bin* disponible y se tratará de construir cada una de las configuraciones solicitadas en la nueva solución, la cual solo pertenecerá a la vecindad de s si este proceso pudo llevarse a cabo exitosamente.

Para crear la población inicial es necesario generar un conjunto de soluciones válidas. Primero se va a generar una solución s_0 . Usando el algoritmo 1 solicitando solo una vez cada una de las hojas que se demandan se obtiene las configuraciones (π_1, \dots, π_k) y los correspondientes (p_1, \dots, p_k) . Luego resolviendo el problema lineal descrito en la Sección 3 se obtendrían los (x_1, \dots, x_n) y aplicando la ecuación 3 se asocia a s_0 su *fitness*, que es el valor que describe cuán bien cumple el objetivo propuesto.

Para obtener otra solución a partir de s_0 se va a seleccionar una solución s_1 que pertenezca al vecinadario de s_0 y luego se selecciona una solución s_2 que pertenezca al vecinadario de s_1 y así sucesivamente hasta que se halla realizado un número fijo de iteraciones $NWalk$. Este proceso está encapsulado en la función $RandomWalk(s)$.

La población inicial va a ser generada aplicando $NPop$ veces la función $RandomWalk(s_0)$. Esto va a constituir la función $CreateInitialPopulation()$.

4.3.2 SELECCIÓN

En la etapa de selección se utiliza una combinación de dos métodos: Selección Elitista (*Elitist Selection*) y Selección basada en Rango (*Rank Based Selection*).

- **Selección Elitista:**

Este método consiste en seleccionar los mejores individuos de una población (aquellos con mejores valores de *fitness*) para que formen parte de la próxima generación. En este caso se seleccionan los primeros $NBest$ individuos.

Este proceso se realiza en la función $ElitistSelection(P)$, donde P es la población de la que se quiere seleccionar los individuos.

- **Selección basada en Rango:**

Este método se basa en un rango que otorga a cada uno de los individuos de la población de forma que aquel que tenga mejor *fitness* va a tener rango n mientras que al peor se le otorga rango 1. La probabilidad de que el individuo i sea seleccionado va a estar dada por la expresión:

$$p(i) = \frac{rango(i)}{\sum_{j=1}^n rango(j)} = \frac{rango(i)}{n * (n + 1) / 2}$$

En el Algoritmo 2 se presenta el pseudocódigo que describe este método. Este proceso va a estar encapsulado en la función $RankSelection(P)$

4.3.3 CRUZAMIENTO

4.3.4 MUTACIÓN

El operador genético Mutación, como muestra el pseudocódigo en el Algoritmo 3, va a seleccionar aleatoriamente un individuo en la población P para que actúe como padre, o sea, como base para formar una nueva solución. Luego se aplica $RandomWalk(s)$ al padre seleccionado como fue explicado en la Sección 4.3.1 para crear la población inicial.

Algorithm 2 RankSelection

Input: P

```

1: Selección  $\leftarrow []$ 
2: for  $k \leftarrow 1, NSel$  do
3:    $r \leftarrow Random(0, 1)$ 
4:   for all  $i \in P$  do
5:     if  $r < \sum_{j=1}^i p(j)$  then
6:       Añadir individuo  $i$  a Selección
7:       break
8:     end if
9:   end for
10: end for
11: return Selección

```

Algorithm 3 Mutation

Input: P

```

1: Parent  $\leftarrow$  Elegir una solución de  $P$ 
2: Offspring  $\leftarrow RandomWalk(Parent)$ 
3:  $HillClimbing(Offspring)$ 
4: return Offspring

```

4.3.5 ALGORITMO

Algorithm 4 GeneticAlgorithm

```

1:  $P_0 \leftarrow CreateInitialPopulation()$ 
2: BestKnown  $\leftarrow UpdateBestSolution(P_0)$ 
3: for  $k \leftarrow 1, NGen$  do
4:    $P_{k-1}^* \leftarrow RankSelection(P_{k-1})$ 
5:    $P_k \leftarrow ElitistSelection(NBest, P_{k-1})$ 
6:   for  $i \leftarrow NBest + 1, NPop$  do
7:     if  $Random(0, 1) < ProbCross$  then
8:       Añadir  $Crossover(P_{k-1}^*)$  a  $P_k$ 
9:     else
10:      Añadir  $Mutation(P_{k-1}^*)$  a  $P_k$ 
11:    end if
12:  end for
13:  BestKnown  $\leftarrow UpdateBestSolution(P_k)$ 
14: end for
15:  $HillClimbing(BestKnown)$ 
16:  $CleanSolution(BestKnown)$ 
17: return BestKnown

```

5. Desarrollo

En esta sección (o secciones) incluya el contenido fundamental del artículo. No es necesario tener una sección nombrada *Desarrollo*, por el contrario, nombre las secciones según el contenido que tratan.

5.1 Organización del Documento

Puede agregar secciones y subsecciones según sea necesario para organizar de manera más coherente su artículo. Tenga en cuenta que un documento más plano es más fácil de navegar y entender, pero las subsecciones relacionadas deberían estar agrupadas en una sección común.

Los nombres de las secciones deben ir en mayúsculas,

excepto para las preposiciones, conjunciones, y otros vocablos auxiliares.

Empiece un nuevo párrafo cada vez que vaya a comenzar una idea nueva.

5.2 Listas y Descripciones

Para producir listas enumeradas, use el siguiente estilo:

1. Primer Elemento
2. Segundo Elemento
 - (a) Segundo Elemento - Subitem Uno
 - (b) Segundo Elemento - Subitem Dos

Para producir descripciones, use el siguiente estilo:

Primer Elemento con su respectiva descripción.

Segundo Elemento también con su respectiva descripción.

5.3 Figuras

Para producir cuerpos flotantes (figuras ó tablas), asegúrese de numerar y etiquetar correctamente cada figura. Las referencias a las figuras deben estar también correctamente etiquetadas. Por ejemplo, en la Fig. 1 se muestra....

Aquí va el contenido de la figura ...

Figure 1: Figura de ejemplo

5.4 Código Fuente

Para producir código fuente, envuélvalo en una figura flotante y etiquételo correctamente. Por ejemplo, en la Fig. 2 se muestra un código bastante conocido....

```
int main(int argc, char** argv)
{
    // Imprimiendo "Hola Mundo".
    printf("Hello, _World");
}
```

Figure 2: Código fuente de ejemplo.

5.5 Referencias

Las referencias deben estar agrupadas en una sección al final del artículo, y las citas numeradas correctamente, por ejemplo [15] ó [16]. Incluya toda la información importante de cada referencia, incluídos autor, título, y notas de la edición. En caso de citar sitios web, además de la URL, incluya la fecha en que fue consultado, como en [17].

6. Conclusiones

En esta sección puede incluir las conclusiones de su investigación y las ideas sobre la continuidad del trabajo, en el caso que aplique.

7. Recomendaciones

En esta sección puede incluir recomendaciones sobre posibles formas de continuar la investigación u otros temas relacionados.

References

- [1] P. Gilmore, and R. Gomory, "A linear programming approach to the Cutting Stock Problem-Part II," *Operations Research*, 11(6), 863-888, 1963.
- [2] H. Hideki, and M.J. Pinto, "An integrated cutting stock and sequencing problem," *European Journal of Operational Research* (183), 1353-1370, 2007.
- [3] J. Karelaiti, "Solving the cutting stock problem in the steel industry". Department of Engineering Physics and Mathematics. Helsinki University of Technology, 2-5, 2002.
- [4] Stéphane Bonnevey, Philippe Aubertin, and Gérald Gavin, "A Genetic Algorithm to Solve a Real 2-D Cutting Stock Problem with Setup Cost in the Paper Industry". Genetic and Evolutionary Computation Conference, 2015
- [5] J. Jylanki. A thousand ways to pack the bin - a practical approach to two-dimensional rectangle bin packing. research report, 2010.
- [6] Cui, Y.-P., Tang, T.-B. Parallelized sequential value correction procedure for the one-dimensional cutting stock problem with multiple stock lengths. *Engineering Optimization*, 46 (10), 1352-1368, 2014.
- [7] R. Haessler, and P. Sweeney, "Cutting stock problems and solution procedures," *European Journal of Operational Research*, 54, 141-150, 1991.
- [8] MirHassani, S.A., Jalaeian Bashirzadeh, A. A GRASP meta-heuristic for two-dimensional irregular cutting stock problem. *International Journal of Advanced Manufacturing Technology*, 81 (1-4), 455- 464, 2105.
- [9] Wenshu, L., Dan, M., Jinzhuo, W. Study on cutting stock optimization for decayed wood board based on genetic algorithm. *Open Automation and Control Systems Journal*, 7 (1), 284-289, 2015.
- [10] Lu, H.-C.a , Huang, Y.-H.b. An efficient genetic algorithm with a corner space algorithm for a cutting stock problem in the TFT-LCD industry. *European Journal of Operational Research*, 246 (1), 51-65, 2015.

- [11] Lu, Q., Zhou, X. GPU parallel ant colony algorithm for the dynamic one-dimensional cutting stock problem based on the on-line detection. *Yi Qi Yi Biao Xue Bao/ Chinese Journal of Scientific Instrument*, 36 (8), pp. 1774-1782, 2015.
- [12] Díaz, D., Valledor, P., Areces, P., Rodil, J., Suárez, M. An ACO Algorithm to Solve an Extended Cutting Stock Problem for Scrap Minimization in a Bar Mill. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8667, 13-24, 2014.
- [13] Ben Lagha, G.a , Dahmani, N.b , Krichen, S.a. Particle swarm optimization approach for resolving the cutting stock problem. 2014 International Conference on Advanced Logistics and Transport, 2014.
- [14] Wikipedia. URL: https://es.wikipedia.org/wiki/Algoritmo_genetico. Consultado el 15 de Noviembre, 2020.
- [15] Donald E. Knuth. *The Art of Computer Programming*. Volume 1: Fundamental Algorithms (3rd edition), 1997. Addison-Wesley Professional.
- [16] Kurt Gödel. *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I*. *Monatshefte für Mathematik und Physik* 38.
- [17] Wikipedia. URL: <http://en.wikipedia.org>. Consultado en November 20, 2020.