UNIVERSIDAD SIMÓN BOLÍVAR
Ingeniería de la Computación
Diseño de Algoritmos II - CI-5652

One-Dimensional Cutting Stock Problem (CSP)

3ra. Entrega

Juan García 05-38207
Federico Flaviani 99-31744

19 de julio de 2011

Índice

1. Introducción **3**
1.1. Breve descripción del problema 3

2. Diseño **4**
2.1. Modelo utilizado para representar el problema 4
2.2. Estructuras y algoritmos involucrados en la aplicación 4
2.2.1. Clase Piece 5
2.2.2. Clase Pattern 5
2.2.3. Clase Chromosome 7
2.2.4. Algoritmo de Colonia de Hormigas 9
2.3. Representación de la solución 12
2.4. Función Objetivo 12
2.5. Operadores 13

3. Detalles de implementación **14**
3.1. Pseudocódigo de las Metaheurísticas 14
3.1.1. Algoritmo de Local Search 14
3.1.2. Algoritmo de Local Search - Mejor Mejor 14
3.1.3. Algoritmo de ILS 15
3.1.4. Algoritmo de GRASP 15
3.1.5. Algoritmo de Simulated Anneling 16
3.1.6. Algoritmo Genético 17

4. Instrucciones de operación **19**

5. Experimentos y análisis de resultados **19**
5.1. Instancias 19
5.2. Resultados y analisis de las Metaherísticas 20

6. Estado actual **21**
6.1. Estado final de la aplicación 21

7. Conclusiones y recomendaciones **22**

8. Referencias bibliográficas **23**

1. Introducción

1.1. Breve descripción del problema

Cutting Stock Problem (CSP) o Problema de Corte y Empaquetamiento es un problema de optimización orientado al área de la programación entera, en donde el objetivo es minimizar el desperdicio generado al cortar una serie de patrones en un área dada. Este problema surge regularmente en el área de la industria siderúrgica o del papel, en donde se busca disminuir las pérdidas monetarias por desperdicios de material.

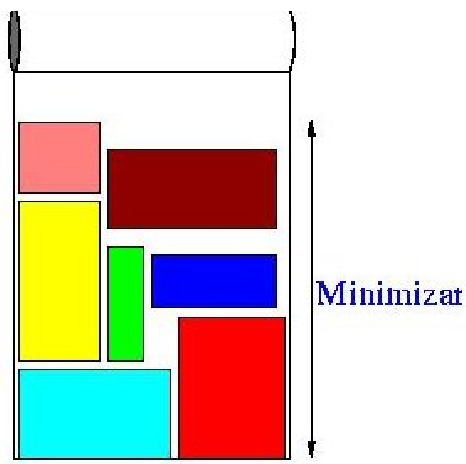

Figura 1: Forma general de CSP en dos dimensiones

En CSP se dispone de un área la cual deseamos cortar y existen muchas maneras de representarla. Puede ser una tela de tamaño finito en todos los sentidos, o una tela de ancho fijo con altura infinita, que es el caso que trataremos en este proyecto. Luego se tienen una serie de formas o patrones que queremos cortar de la mencionada tela. Dado que nuestro problema es la versión unidimensional (Figura 2), las piezas que se quieren cortar de la tela seran rectangulos con base constante igual a uno y con alturas variadas. Además no se permitirá el cambio de orientación de las piezas. Instancias del problema con mas nivel de dificultad se encuentra con la versión de dos dimensiones y piezas de tamaño y formas variables, asi como en tres dimensiones, en donde el problema se basa en optimizar la forma de empaquetar piezas en un volumen dado, tipo un contenedor de mercancia.

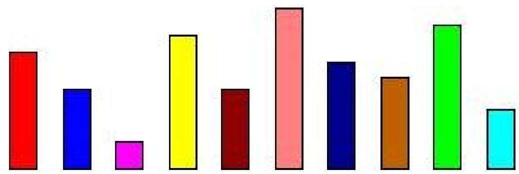

Figura 2: Generalización a cortes unidimensionales

2. Diseño

2.1. Modelo utilizado para representar el problema

Como se mencionó anteriormente la forma en que representamos el problema es usando una tela ïnfinitaçon ancho fijo. Para representar la infinitud de la tela hallamos una cota superior que cubra todos los posibles cortes que se pueden realizar, de esta forma nuestra cota superior CS sera:

$$CS = \sum_{i=1}^{N} L_i$$

donde N es el número de piezas a cortar y L_i es el largo de la pieza i.
En nuestro programa crearemos una serie de estructuras para representar cada parte del problema, comenzando con las piezas que queremos cortar, pasando por patrones de corte ya establecidos, los cuales representaran en nuestro caso una solución factible.

2.2. Estructuras y algoritmos involucrados en la aplicación

Dentro de las principales estructuras que utilizamos, estan aquellas que nos permiten representar un estado o solución factible. Para ellos necesitamos piezas y un patrón de corte, por lo tanto se definirán las estrucutras *Piece* y *Pattern* en conjunto con su tanda de atributos y diferentes métodos.

En cuanto a los algoritmos utilizados, encontramos en primera instancia un algoritmo greedy que nos permitira crear una solución inicial. Luego se aplicarán una serie de procedimientos con el fin de determinar una vecindad en un cierto espacio. Para finalmente aplicar los algortimos de las diferentes meta-heuristicas que se van a implementar para resolver el problema.

En primera instancia el algoritmo greedy que genera una solución inicial consistia en ir aplicando cortes sucecivos a la tela, de la manera mas eficiente posible. Primero se ordenaban los cortes a realizar según el tamaño de mayor a menor, para comenzará por la base colocando las piezas por fila. El problema de esta solucion inicial es que siempre generaba un optimo local, por lo tanto los algoritmos de busqueda no lograban optmimizar mas alla de lo que generaba la solucion.

Para resolver esta problemática se decidió no ordenar las piezas por el tamaño, sino ir realizando los cortes en el orden en que se iban aplicando las demandas de los clientes. Y luego de colocar lor cortes, se aplica una rutina de perturbación, que permite degerar mas la solución, y así ayudar a las metaheurísticas a encontrar una mejor solución.

2.2.1. Clase Piece

Esta clase representa una pieza a ser cortada de la tela. Posee un atributo, *large* que representa el largo del rectangulo a cortar. Además, como todo objeto tiene un constructor que recibe como parámetros un entero que representa el largo del bloque para inicializar el tamaño del mismo en la tela. Finalmente posee un método *clone* que permite crear una copia de una pieza.

```
class Piece {
 public:
  int large;
  Piece(int);
  Piece clone();
};
```

2.2.2. Clase Pattern

La clase *Pattern* representa un patrón de corte de una isntancia del problema del CSP. Es decir es una solucion factible, en donde estan posicionadas todas las piezas de corte sobre la tela. A partir de una instancia de patter se obtienen otras instancias de la misma aplicacando los operadores de vecindad, asi como se puede obtener el nivel de calidad de la misma, para hacer comparaciones entre un patrón y otro.

La clase tiene una serie de atributos y métodos, pero dentro de los mas importantes encontramos a *width* y *height* que representan el alto y ancho de la tela a cortar. *num_pieces* y *pieces* son el número de piezas y la lista que contiene a las mismas, repectivamente. Además otros atributos que modelan el area ocupada y área libre, el número de lineas concretadas en el patrón y variables enteras que soportan la generación de las vecindades.

Dentro de los métodos mas importantes encontramos los constructores de clase, que permiten inicializar un patrón cualquiera, también el método *vicinityNext* que permite ir generando la vecindad dinamincamente, basado en los métodos *vicinityFirstLevel*, *vicinitySecondLevel* y *vicinityThirdLevel*, el método *perturbar* que permite pertubar una solución iniciada para aplicar la metaheuristica *ILS*, entre otros métodos de apoyo para hacer revision de calidad y de la función objetivo.

```
int **paper;
int width;
int height;
int heightMax;
int num_pieces;
list<Piece *>* pieces;
int area_ocup;
int area_no_ocup;
int lines;

Pattern();
Pattern(int, int);
Pattern(list<Piece *>, int, int);

list<Pattern *> genVicinity();
Pattern* vicinityOperator(int, int, int);
void swap(int *, int, int);
void updateRemovePaper(Piece *);
void updateAddPaper(Piece *);
void deleteList(list<Pattern *> *);
Pattern* perturb();
Pattern* clone();
int calcHeight();
int quality();
void actualizar();
void print();
void swap();
Pattern* addRequest(list<Piece *>);
void addPiece(Piece *, int);
```

2.2.3. Clase Chromosome

Esta clase representa un Cromosoma en la implementación del algoritmo Genético. Basicamente posee una matriz que almacena las demandas de los clientes por fila, y por columna se representa una torre de piezas ubicadas por columna a lo ancho de la tela.

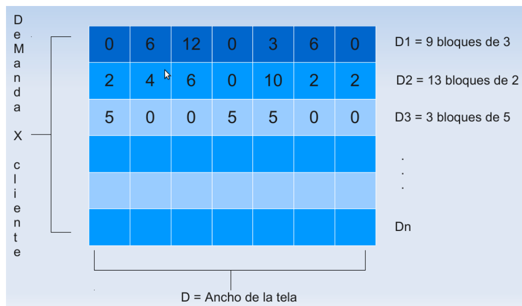

Figura 3: Representación de un cromosoma

Posee otros atributos que permiten terminar de representar un cromosoma y obtener una solución factible del mismo, en referencia al problema que estamos atacando. Variables que almacenan el ancho y alto de la matriz, la altura máxima alcanzada, la calidad del cromosoma, el área ocupada por el patrón y las lineas generadas. De igual forma posee métodos que permiten construir un nuevo cromosoma y aleatoriamente llenarlo con valores de acuerdo a la demanda especificada, asi como las principales funciones de Crossover y Mutación, indispensables en los algoritmos genéticos.

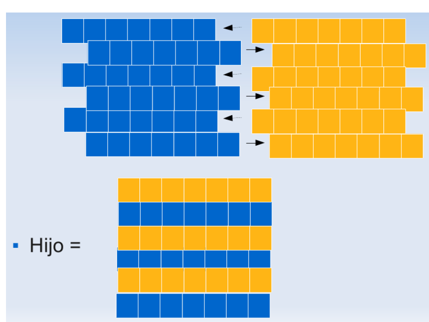

Figura 4: Operador de Crossover

El operador de Crossover realiza un cruce por filas de dos cromosomas dados. Del primer padre toma las filas pares y del segundo la filas impares, y a partir de ellas genera un nuevo hijo, que gracias a la representación que se realizó, permite siempre obtener soluciones factibles.

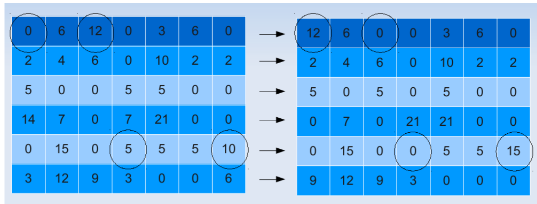

Figura 5: Operador de Mutación

El operador de mutación se ejecuta con una probabilidad de 0.3, y cuando se ejecuta selecciona aleatoriamente el número de hijos que se mutaran. Y el efecto que tiene es que para cada fila del cromosoma, correspondiente a un cliente, realiza el equivalente a mover un grupo entero de piezas de una columna a otra, de igual manera aleatoriamente. En el ejemplo de la figura 5 queda mejor reflejado.

```
class Chromosome {

  int **genes;
  int quality;
  int req;
  int width;
  int *tops;
  int heightMax;
  int area;
  int lines;

  Chromosome(int, int);
  ~Chromosome();
  void updateQuality();
  Chromosome* crossover(Chromosome*);
  void mutation();
  void fillChrom(int*, int*);
  void print();
};
```

2.2.4. Algoritmo de Colonia de Hormigas

Para entender el algoritmo de hormigas que hemos implementado restringiremos la altura maxima que nuestros patrones pueden tener por L fijo y seguidamente indexemos el conjunto de las piezas de 1 hasta n, con n igual al total de piezas distintas:

Definición. *Definiremos las variables l_i como la longitud de la pieza i*

Definición. *Una columna factible es un multiconjunto de enteros $\{i_1, i_2, \ldots, i_h\}$ tales que $\Sigma_{k=1}^{h} l_{i_k} \leq L$.*

Dado un numero de tipos de piezas fijas y sus respectivas longitudes, se tiene que el número total de columnas factibles es finito, denotemos a este número finito por m e indexemos el conjunto de columnas factibles.

Definición. *Sea $\{i_1, i_2, \ldots, i_h\}$ la columna factible j, denotemos entonces por S_i como el número $L - \Sigma_{k=1}^{h} l_{i_k}$ y a O_{ij} como la cantidad de veces que se repite el entero i en la columna factible j*

A continuación describiremos el algoritmo:

El concepto de columna factible es fundamental para nuestro algoritmo de hormigas, por lo tanto es importante destacar que representaremos una columna factible j como un arreglo de enteros de tamaño igual al total de piezas distintas y donde cada entero será un O_{ij}.

La primera etapa del algoritmo consiste en calcular todas las posibles columnas factibles y colocar cada una de ella en un arreglo, de modo que este arreglo implementa la indexación necesaria que se mensionaba en los párrafos anteriores para poder definir las variables S_j y O_{ij}.

La idea del algoritmo es que en cada iteración, cada hormiga escoja probabilisticamente una columna factible hasta el punto que el total de las piezas de todas estas columnas sobrepase el total de las piezas del problema. El exeso de piezas será penalizado en la función objetivo y las probabilidades de escogencia de columna será menor en la medida que de estas escogencias resulte un crecimiento del número extras de piezas.

La función objetivo según lo dicho anteriormente será $\Sigma_{j=1}^{m} S_j X_j + \Sigma_{i=1}^{n} V_i$ donde X_i es el número de veces que se usó la columna factible j en la solución y V_i es el total de las piezas extras.

Las probabilidades de escogencia de las columnas se realizan en dos etapas: Primero se escoje una pieza probabilisticamente y luego se escoje (probabilisticamente tambien) una columna donde se encuentre la pieza escojida.

Para asignar las probabilidades antes mencionadas necesitamos la siguiente.

Definición. *Para una solución (no necesariamente factible) fija definimos a P_i como la cantidad de piezas del tipo i que se encuentran en dicha solución y a D_i como la cantidad máxima de piezas disponibles del tipo i*

Definición.

$$M_i := \begin{cases} D_i - P_i & si \quad P_i > 0 \\ \\ 0 & si \quad D_i - P_i \leq 0 \end{cases}$$

Definición. *Denotaremos por TM a la suma $\Sigma_{\forall i} M_i$*

En la primera iteración de nuestro algoritmo la probabilidad de escoger la pieza i será igual a $\frac{M_i}{\Sigma_{\forall k} M_k}$ y una vez escogida i la probabilidad de escoger una columna factible j que contenga la pieza i será $1 - \frac{S_j}{\Sigma_{k \in \Omega_i} S_k}$, donde $\Omega_i := \{j | O_{ij} > 0\}$.

Como es posible con estas probabilidades que se obtenga una solución no factible con mayor cantidad de piezas de algún tipo de la permitida, entonces en las siguientes iteraciones las probabilidades penalizan estas situaciones. Para definir estas probabilidades necesitamos las siguientes

Definición. *Denotaremos por P'_i el número total de piezas de la solución de la iteración anterior y definamos*

$$V_i^{\prime aux} := (P'_i - D_i) * l_i$$

No nos interesa que $V_i^{\prime aux} = 0$ y que piezas que no tiene una cantidad execiva en la iteración anterior tenga mucha mas probabilidad de escogerce que otras, de esta forma damos valores mas uniformes de V' en la siguiente

Definición.

$$V'_i := \begin{cases} V_i^{\prime aux} & si \quad V_i^{\prime aux} > 0 \\ \\ (0{,}5) min\{V_j^{\prime aux} | V_j^{\prime aux} > 0\} & si \quad V_i^{\prime aux} = 0 \end{cases}$$

Definición. *Denotemos por TOV' a $\Sigma_{\forall i} V'_i$*

De esta forma la probabilidad de escogencia de la pieza i para iteraciones distintas a la primera es $\frac{\frac{M_i}{TM}(1 - \frac{V'_i}{TOV'})}{Sum}$ donde $Sum = \Sigma_{\forall i} \frac{M_i}{TM}(1 - \frac{V'_i}{TOV'})$, y por lo tanto mientras mas se sobrepase la cantidad de piezas del tipo i en la solución de la iteración anterior, menos probabilidad tiene de ser escogida en la actual iteración.

Una vez que se escoge la pieza i las probabilidades para escoger una columna factible j es $\frac{P'_j}{\Sigma_{\forall j \in \Omega_i} P'_j}$ con $\Omega_i = \{j | \forall O_{ij} > 0\}$ y P'_j dada por la siguiente definición

Definición. *Sean $X_{0(k)}$ y $X_{0(k-1)}$ los valores de la función objetivo de la solución de la iteración k y $k-1$ (suponemos que la iteración actual es la $k+1$) respectivamente, y sea N'_j el número de veces que se escogio la columna factible j en la solución de la iteración anterior, entonces definimos:*

$$P'_j := \begin{cases} \frac{1}{N'_j S_j} O_{ij} & si \qquad X_{0(k)} > X_{0(k-1)} \\ \\ \frac{N'_j}{S_j} O_{ij} & si X_{0(k)} \leq X_{0(k-1)} \end{cases}$$

Esta escogencia del valor de P'_j en función de si las soluciones mejoran o no en cada iteración es una simulación de la evaporación de la feromona.

2.3. Representación de la solución

La solución al problema de CSP se encuentra representada mediante la clase *Pattern*, en donde se encuentra una lista de piezas que fueron cortadas y poseen mediante la clase *Piece* una serie de atributos que denotan la posición en la tela, y forman el patrón como fue cortada. Además podemos saber la altura y el número de lineas creadas con el corte, y analizar una función objetivo.

Esto es de manera general, pero la estructura que en concreto representará nuestra solución se basa en un arreglo de listas de piezas. El tamaño del arreglo viene definido por el ancho fijo de la tela a cortar, y cada una de las listas se compone por la piezas colocadas al estilo de una pila, en la columna i del arregla o tela. Esta representación nos será de conveniencia al momento de realizar un movimiento para generar un estado vecino.

2.4. Función Objetivo

La función objetivo que vamos a utilizar esta definida por diferentes parámetros. De manera general definimos una métrica de calidad de los estados solución basada en la máxima altura generada por el corte de cada una de las piezas. Es decir que si comparamos dos estados solución, en donde ambos tengan un patrón con el mismo número de piezas cortadas, es decir, la misma área ocupada, aquel patrón que tenga la altura menor sera la mejor. Indirectamente al hablar de altura mínima, esto se refiere a el mínimo de área o material desperdiciado. En primera instancia esa es la métrica que vamos a utilizar en nuestro problema.

Otra función objetivo que podemos usar esta basada en el número de lineas completas generadas en el patrón, es decir, todas las lineas completas de que tengan el ancho total de la tela, y como altura una unidad. El problema de esta métrica es que podemos encontrar dos patrones con el mismo número de lineas, pero con alturas muy diferente, en donde el malgasto sea excesivo.

Una idea mejor es crear una función objetivo que relacione las dos métricas anteriores, aunque actualmente no hemos definido esta estrategia para ser usada en las metaheuristicas a implementar.

2.5. Operadores

Al inicio del proyecto, para generar la vecindad para un estado solución utilizabamos un conjunto de tres operadores, basados en el movimiento de piezas. El operador de primer nivel genera una vecindad moviendo una pieza seleccionada de los niveles superiores. El siguiente operador, de segundo nivel, fija una pieza seleccionada y realiza una llamada recursiva al operador de primer nivel, al final se recolocará la pieza fijada, resultando asi un movimiento de dos piezas. El tercer operador es una llamada al operador de tercer nivel, en donde se fija una pieza y se llama a el operador de segundo nivel.

Pero al analizar estos operadores nos dimos cuenta que resultaban ineficientes y la vecinandad que generaban era excesivamente grande. Por lo tanto, siguiendo recomendaciones de compañeros de clase, decidimos implementar un nuevo operador. Este nuevo operador se basa en mover cualquier pieza seleccionada entre todo el conjunto, a alguna posición en el tope del patrón de corte. De esta manera el tamaño de la vecindad será siempre finita, definida por

$$V = (\sum_{i=1}^{N} L_i) \times W$$

que es solo multiplicar el número total de piezas por el ancho fijo de la tela. Entonces para aquellas metaheurísticas donde sea necesario generar toda una vecindad, en general el uso de memoria sera siempre fijo, y la diversidad de los patrones sera considerable.

3. Detalles de implementación

3.1. Pseudocódigo de las Metaheurísticas

3.1.1. Algoritmo de Local Search

Require: Patron inicial : initial
  $k \leftarrow 0$
  Patron $actual \leftarrow initial$
  Patron $next$
  Lista Patrones $vicinity$
  while $k < 50$ **do**
    $vicinity \leftarrow actual.genVicinity()$
    while $!vicinity.empty()$ **do**
      $next = v$
      if $(next.quality() < actual.quality()$ **then**
        $actual.destroy()$
        $actual \leftarrow next$
        $break$
      end if
    end while
    $k + +$
    $deleteList(vicinity)$
  end while
  return return actual;

3.1.2. Algoritmo de Local Search - Mejor Mejor

Require: Patron inicial : initial
  $k \leftarrow 0$
  Patron $actual \leftarrow initial$
  Patron $next \leftarrow actual$
  Lista Patrones $vicinity$
  while $k < 50$ **do**
    $best \leftarrow actual$
    $vicinity \leftarrow actual.genVicinity()$
    while $!vicinity.empty()$ **do**
      $next = v$
      if $(next.quality() < best.quality()$ **then**
        $actual.destroy()$
        $best \leftarrow next$

end if
end while
if $(best.quality() < actual.quality()$ **then**
$actual.destroy()$
$actual \leftarrow best$
end if
$k{+}{+}$
$deleteList(vicinity)$
end while
return return actual;

3.1.3. Algoritmo de ILS

Require: Patron inicial: s_{mejor}
$k \leftarrow 0$
Patron $s1$,$s2$
Patron $s_{mejor} \leftarrow localSearch(s_{mejor})$
while $k < 10$ **do**
$s1 \leftarrow s_{mejor}.perturbar()$
$s2 \leftarrow localSearch(s1)$
if $s2.quality() < s_{mejor}.quality()$ **then**
$s_{mejor}.destroy()$
$s_{mejor} \leftarrow s2$
end if
$s2.destroy()$
$k{+}{+}$
end while
return s_{mejor}

3.1.4. Algoritmo de GRASP

Require: Lista Piezas $piecesPart$, int $demanda$
Patron pat
Lista Patron RCL
int ran, $count$
while $i < demanda$ **do**
while $j < 20$ **do**
$RCL.insert(pat.addRequest(piecesPart[i]))$
end while
$RCL.sort()$

$ran \leftarrow random()$
$count \leftarrow 0$
while $!RCL.empty()$ **do**
$count++$
if $count = ran$ **then**
$pat.destroy()$
$pat \leftarrow RCL(r)$
$break$
end if
end while
$pat \leftarrow localSearch(pat)$
end while
return pat

3.1.5. Algoritmo de Simulated Anneling

Require: Patron $Initial$
int $i \leftarrow 0$
int $a \leftarrow 0$
int $maxAccept \leftarrow 20$
double $k \leftarrow 0{,}0$
double $maxIter \leftarrow 200{,}0$
double $ro \leftarrow 1{,}05$
double $temp \leftarrow 1000{,}0$
double $alfa \leftarrow 0{,}8$
double $delta$
int $from, pieceNum, to$
double $suerte$
Patron $actual \leftarrow initial.clone()$
Patron $next$
while $i < 50$ **do**
while $k < maxIter \wedge a < maxAccept$ **do**
while $true$ **do**
$from \leftarrow random(1, width)$
if $actual.pieces[from].size()! = 0$ **then**
$pieceNum \leftarrow random(0, numPiecesFrom)$
$to \leftarrow random(1, width)$
$break$
end if
end while

{Se genera el vecino proximo a evaluar}
$next \leftarrow actual.vicinityOperator(from, pieceNum, to)$
{Se calcula el delta con la comparacion de calidades}
$delta \leftarrow next.quality() - actual.quality()$
if $delta < 0$ **then**
$actual.destroy()$
$actual \leftarrow next.clone()$
$a++$
else
$suerte \leftarrow random(0, 1)$
if $temp > 0{,}0$ **then**
if $suerte < e^{-delta/temp}$ **then**
$actual.destroy()$
$actual \leftarrow next.clone()$
$a++$
end if
end if
end if
$k \leftarrow k + 1$
end while
{Disminuye la temperatura}
$temp \leftarrow temp * alfa$ {Aumenta el numero de iteraciones}
$maxIter \leftarrow maxIter * ro$
$k \leftarrow 0{,}0$
$a \leftarrow 0$
$i++$
end while
return actual;

3.1.6. Algoritmo Genético

Require: int $request$, int $width$, int[] $numPieces$, int[] $largePieces$
int $sizePop \leftarrow 400$
Lista de Cromosomas $population$, $best$, $worst$, $child$
Iterador sobre lista de Cromosomas it, itB, itW
Iterador reverso sobre lista de Cromosomas rit
Cromosoma ch
int $count \leftarrow 0$
double $suerte$
double $probMut \leftarrow 0{,}3$

int ran
{Se genera la Poblacion inicial}
while $i < sizePop$ **do**
  $ch \leftarrow newChromosome(request, width)$
  $ch.fillChrom(numPieces, largePieces)$
  $population.insert(ch)$
  $i++$
end while
while $g < 1000$ **do**
  $population.sort()$ {Se ordena la poblacion por calidad}
  {Se toman los mejores}
  $count \leftarrow 0$
  while $count < sizePop/4$ **do**
    $it \leftarrow population.getBest()$
    $best.insert(it)$
  end while
  {Se toman los peores}
  $count \leftarrow 0$
  while $count < sizePop/4$ **do**
    $it \leftarrow population.getWorst()$
    $worst.insert(it)$
  end while
  {Se cruzan los mejores con los peores}
  $count \leftarrow 0$
  while $count < sizePop/4$ **do**
    $itB \leftarrow population.getBest()$
    $itW \leftarrow population.getWorst()$
    $child.insert(croosover(itB, itW))$
  end while
  {Se cruzan los mejores entre ellos}
  $count \leftarrow 0$
  while $count < sizePop/4$ **do**
    $itB1 \leftarrow population.getBest()$
    $itB2 \leftarrow population.getAnotheBest()$
    $child.insert(croosover(itB1, itB2))$
  end while
  {Se decide si se realiza mutacion}
  $suerte \leftarrow random(0, 1)$
  if $suerte < probMut$ **then**
    $ran \leftarrow random(1, sizeChild)$

$count \leftarrow 0$
while $count < ran$ **do**
$it \leftarrow child.get()$
$it.mutation()$
end while
end if
{Se realiza la repoblacion}
$count \leftarrow 0$
while $count < child.size()$ **do**
$itW \leftarrow population.getWorst()$
$population.erase(itW)$
$it \leftarrow child.get()$
$population.insert(it)$
end while
end while
$population.sort()$
return $population.front()$

4. Instrucciones de operación

Para compilar el programa debe realizar la llamada:

```
$> make
```

Y para realizar la corrida:

```
$> ./CSP Metaheuristica Instancia -- Ej: ./CSP ILS 100
```

5. Experimentos y análisis de resultados

En la siguiente seccion detallaremos cuales fueron las instancias que fueron corridas, y un analisis de las diferentes metaheuristicas que fueron implementadas.

5.1. Instancias

En la literatura y en las librerias en la red, se nos fue imposible encontrar un set de problemas para correr nuestros algoritmos. En general tuvimos que diseñar nuestros propios casos de prueba, los cuales se listan a continuacion en un formato definido como un vector $\{C_1/L_1, C_2/L_2, ..., C_n/L_n\}$, donde cada C_i representa la cantidad de bloques de tamaño L_i:

1. Instancia de 55 piezas: tela de 10 de ancho con de 8 clientes con la siguiente demanda: $\{3/8, 5/4, 7/6, 9/2, 5/8, 15/5, 6/9, 5/7\}$
2. Instancia de 66 piezas: tela de 8 de ancho con de 7 clientes con la siguiente demanda: $\{7/6, 5/2, 11/3, 20/5, 5/2, 8/5, 10/4\}$
3. Instancia de 100 piezas: tela de 8 de ancho con de 5 clientes con la siguiente demanda: $\{20/5, 20/4, 20/3, 20/2, 20/7\}$
4. Instancia de 150 piezas: tela de 10 de ancho con de 14 clientes con la siguiente demanda: $\{13/8, 5/4, 17/6, 9/2, 15/8, 15/5, 6/9, 5/7, 10/2, 24/8, 8/7, 9/9, 10/2, 4/10\}$

5.2. Resultados y analisis de las Metaherísticas

Como se ha mencionado a lo largo del informe, las metaheurísticas que fueron implementadas y corridas fueron las de trayectoria LS, LS-MM, ILS y Simulated Anneling, las constructivas de GRASP y Hormiga y finalmente un algoritmo Genético en la parte de poblacionales.

En general las corridas de los diferentes algoritmos generaban buenos resultados, a excepción de algunos que ofrecian normalmente malas soluciones, o aquellos algoritmos mas estables que generalmente ofrecian buenas soluciones. Para los algoritmos de trayectoria se obtiene un mejor resultado para Simulated Anneling, el cual resultó ser el que mejor se adapta al problema y ofrece los optimos para todas las instancias. Con relación a los de búsqueda, solo la búsqueda local iterada (ILS) logro obtener buenos resultados, pero adicionando un costo de tiempo y memoria que pueden resultar vitales. Para la búsqueda local sencilla lo mas importante es la dependencia con la solución inicial, en general para una solución inicial "buena" el algoritmo se quedaba en un optimo local, pero al realizar una gran perturbación a la misma, y obtener una solución inicial "peor", el algoritmo lograba converger a una mejor solución cerca del optimo del problema.

Para los algoritmos constructivos no pudimos obtener buenos resultados. Especificamente para la metaheurística de GRASP no fue posible encontrar una solución buena que se acercara al optimo de la instancia. La opción constructiva no aporta un beneficio para esta clase de problema, porque aunque se elijan los mejores siempre de una lista de candidatos, la solución greedy te va adaptando el problema a una convergencia siempre a un optimo local, a veces definido en la etapa de optimización del algoritmo.

Otros buenos resultados se obtuvieron del algoritmo Genético, el cual gracias a la representación realizada permitió diversificar mucho el espacio de resultados que se pueden obtener, y con los operadores de cruce y mutacion fue posible ir generando nuevas generaciones de soluciones factibles que por lo general siempre lograban converger a una buena solución.

6. Estado actual

6.1. Estado final de la aplicación

La aplicación se encuentra totalmente operativa con relación a la representación del problema, una solución factible, los operadores y la función objetivo. Además estan implementadas y funcionales las metaheurísticas de LS, LS-MM, ILS, GRASP, Simulated Anneling y Algoritmo Genético. Y aunque se realizó una implementación de Hormiga, no se encuentra del todo operativa.

7. Conclusiones y recomendaciones

En el desarrollo de esta tercera etapa del proyecto pudimos reafirmar la complejidad que tiene este problema en cuanto a su representación, y de la cual depende en gran cabida el funcionamiento de las diferentes metaheurísticas. Sin embargo, fue posible solventar los diferentes problemas que encontramos en la 2da entrega, y ahora tenemos una representación sólida y más fácil de utilizar, la cual consigui representar de manera efectiva el problema que estamos atacando.

Ya para finalizar pudimos determinar que son dos metaheuristicas las que mejor se adecuan a nuestro problema y que ofrecen buenas soluciones en un tiempo reducido y con poca necesidad de recursos computacionales. Las mismas fueron la metaheurística de trayectoria Simulated Anneling, que es la mas estable de todas, en donde para toda corrida siempre ofrece muy buenos resultados iguales o cercanos al optimo de la instancia. Y por otro lado el Algoritmo Genético, que para soluciones pequeñas y medianas resulta ser bastante estable y genera buenas soluciones. Con un detalle, que para problemas mas grandes se deben realizar algunas adaptaciones, aunque sigue teniendo un mejor desempeño que otras metaheurísticas probadas.

8. Referencias bibliográficas

Referencias

[1] **Solving the Cutting Stock Problem in the Steel Industry**
KARELAHTI J.

[2] **A Genetic Solution for the Cutting Stock Problem**
ANDRAS P., ANDRAS A, SZABO Z.

[3] **A Progressive Heuristic Search for the Cutting Stock Problem**
ONAINDIA E., BARBER F., BOTTI V., CARRASCOSA C., HERNANDEZ M., REBOLLO M.

[4] **An ACO Algorithm for One-Dimensional Cutting Stock Problem**
ESHGHI K., JAVANSHIR H.

[5] **Using Genetic Algorithms in Solving the One-Dimensional Cutting Stock Problem in the Construction Industry**
SHAHIM A., SALEM O.

[6] **A Simulated Aneeling Approach for a Standard One-Dimensional Cutting Stock Problem**
JAHROMI M.H.M.A., TAVAKKOLI-MOGHADDAM R., GIVAKI E., REZAPOUR-ZIBA A.

[7] **An ACO Algorithm for One-Dimensional Cutting Stock Problem**
ESHGHI K., JAVANSHIR H.