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

One-Dimensional Cutting Stock Problem (CSP)

2da. Entrega

Juan García 05-38207
Federico Flaviani 99-31744

30 de junio de 2011

Índice

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.

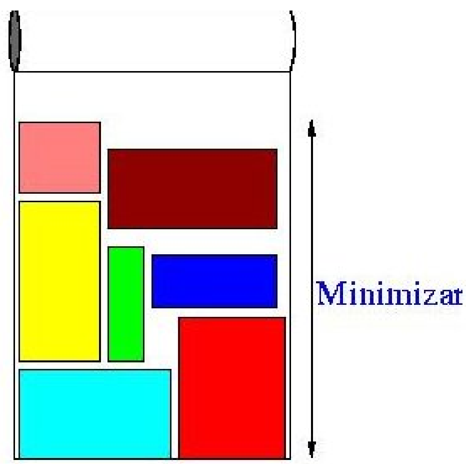

Figura 1: Forma general de CSP en dos dimensiones

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.

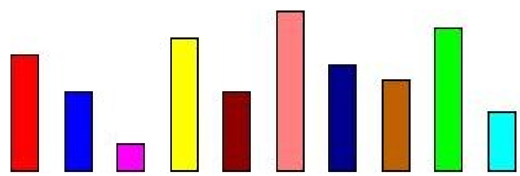

Figura 2: Generalización a cortes unidimensionales

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 dos atributos, *large* que representa el largo del rectangulo a cortar y *pos* que permite conocer la posición de la pieza cortada en la tela total. Además, como todo objeto tiene un constructor que recibe como parámetros un entero que representa el largo del bloque y un arreglo bidimensional para inicializar la posición del mismo en la tela. Finalmente posee un método *clone* que permite crear una copia de una pieza.

```
class Piece {
 public:
  int large;
  int pos[2][2];
  int id;
  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. 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.

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

En la entrega anterior, 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. Pseudocodigo de las estructuras

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 ← best
  end if
  k++
  deleteList(vicinity)
end while
return  return actual;
```

3.1.3. Algoritmo de ILS

```
Require: Patron inicial: s_mejor
  k ← 0
  Patron s1,s2
  Patron s_mejor ← localSearch(s_mejor)
  while k < 10 do
    s1 ← s_mejor.perturbar()
    s2 ← localSearch(s1)
    if s2.quality() < s_mejor.quality()  then
      s_mejor.destroy()
      s_mejor ← 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

4. Instrucciones de operación

Para compilar el programa debe realizar la llamada:

```
$> make
```

Y para realizar la corrida:

```
$> ./CSP
```

5. Estado actual

5.1. 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 e ILS. Y aunque se realizó una implementación de GRASP y Hormiga, no se encuentran del todo operativas.

5.2. Errores

Existe un problema en la implementación de la metaheurística de GRASP, relacionado con la actualización de la posición de las piezas. Para la entrega final, ya que este campo será eliminado, podrá terminar de realizarse la implementación de la metaheurística. Con relación al algoritmo Hormiga, se realizó una implementación, pero que no se encuentra operativa.

6. Conclusiones y recomendaciones

En el desarrollo de esta segunda 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 1ra entrega, y ahora tenemos una representación sólida y más fácil de utilizar, para seguir implementando los algoritmos que restan.

Además es importante destacar el cambio de enfoque que se le da al problema dependiendo del tipo de metaheurística que se desea implementar, sobre todo al momento de comenzar con la implementación del algoritmo de hormiga, el cual enfrenta el problema de una manera diferente a los algoritmos anteriormente implementados.

6.1. Mejoras

Es necesario realizar ligeros cambios en la representación de las piezas. Por recomendaciones de los compañeros de la materia, debemos suprimir el campo de posición, ya que no aporta ninguna ventaja sobre el modelo, porque debido a la representación de listas de piezas por columna, y porque lo que queremos es minimizar alturas, el orden o posición de las piezas es irrelevante. De esta forma realizar movimiento de piezas resultará más faćil, porque solo se debera hacer operaciones de suma o resta pocas veces.

7. Tablas

Parametro	Valor
k = Nro. iteraciones LS	100
k = Nro. iteraciones LS Mejor-Mejor	100
k = Nro. iteraciones ILS	100

Cuadro 1: Parametros

La instancia que se presenta a continuacion tenia 66 piezas, ancho de 8 y un optimo de 5. Ademas la demanda es de 7 clientes, ordenado de la siguiente manera: $\{7/6, 5/2, 11/3, 20/5, 5/2, 8/5, 10/4\}$. En donde la notacion N/L representa N piezas de tamaño L.

Heuristica	Mejor Tiempo (seg)	Peor Tiempo (seg)	Tiempo Promedio (seg)
Local Search	3.52	4.01	3.747
LS Mejor-Mejor	3.75	3.99	3.87
ILS	43.6	45.21	44.421

Cuadro 2: Tiempo de las heuristicas para instancia de 66 piezas

Heuristica	Mejor Solución (seg)	Peor Solución (seg)	Solución Promedio (seg)
Local Search	13	53	21.8
LS Mejor-Mejor	13	53	21.4
ILS	5	21	13

Cuadro 3: Soluciones de las heuristicas para instancia de 66 piezas

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.